Understanding the Apple II

by Jim Sather

Foreword by Steve Wozniak



QS QUALITY SOFTWARE

$22.95

Understanding the Apple II

A Learning Guide and Hardware Manual for the Apple II Computer

Quality Software is pleased to present the definitive source of information about how the Apple works. Jim Sather has conducted an exhaustive analysis of the inner workings of the Apple II computer. Now he has documented his findings in a way that will benefit everyone interested in microcomputer technology.



Understanding the Apple II—

- Documents all motherboard circuits, including some discussed nowhere else.
- Describes disk controller operation, including previously undocumented details of the logic state sequencer.
- Explains RAM and ROM card operation.
- Reveals previously unnoticed features of Apple graphics.
- Contains 23 software and hardware Application Notes including shift key mod, disk write protect mod, and EPROM mods.
- Includes a chapter on maintenance that provides simple troubleshooting steps.

If you are at all curious about how the Apple II works, you are sure to find *Understanding the Apple II* very valuable. It is an ideal book for a microcomputer fundamentals course based on the Apple.

About the Author

James Fielding Sather, a former electronics field technical representative for ITT Gilfillan, is an independent author, programmer, and designer of circuits for microcomputers, specializing in the Apple II.

Understanding the Apple II describes the Apple II and Apple II Plus. However, some information, including that on disk controller operation, also applies to the Apple IIe.

QS QUALITY SOFTWARE

ISBN 0-912985-01-1

Printed in USA

Understanding the Apple II

by James Fielding Sather

QS QUALITY SOFTWARE
21601 Marilla Street
Chatsworth, California 91311

K

861 204 15

10
QC550
.S253

Univ.
Bibliothek
Bielefeld

Production Editor: Kathryn M. Schmidt
Original Schematics and Diagrams: James Fielding Sather
Art Director and Cover Design: Vic Grenrock
Cover Art: George Garcia
Schematic Art: Ron Widman
Photography: Gainsforth Studios
Compositor: American Typesetting, Inc.
Printed By: California Offset Printers

Copyright © 1983 by Quality Software. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing 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: 0-912985-01-1
Printed in the United States of America

BI 100/384105+01



foreword

Exhausted after addressing an Applefest audience in Anaheim, I was offered a ride to the airport by a stranger who was writing a book on something Apple related and who wanted to talk to me. The traffic congestion, missed exits, and airport confusion have deeper meaning to a computer designer, but I shall never forget the conversation we had for the next couple of hours. Jim Sather proceeded to explain details, anomalies, oversights, and paradoxes of the Apple II hardware as we drove the LA freeways. Designers like myself find it very rewarding to encounter others who understand and appreciate what we feel are the tricks and magic of our circuits. I was able to add to the magic by explaining the unusual framework in which the computer was designed.

The Apple II was designed as an interesting hobby project to "show off" at the local computer club. Only one or two were intended to be built, and then for a market of one (myself). Component selection was biased by what was easy to obtain and affordable. Features such as game I/O, color, and graphics were added to impress friends with specific applications. The driving motivation was to accomplish a lot with very few components. In this framework the system features were largely influenced by the technical environment—which ICs occupying how much board space, the bandwidth of normal home TVs, etc. It is more common today to specify features based on market considerations.

Since the hardware and software tasks were not partitioned in advance for separate groups to work on, a tremendous synergy evolved. **Understanding the Apple II** *respects this close interdependence. It contains discussion and examples throughout. Hardware and software developers alike (and enthusiasts and students too) will gain greatly from this manual. This is especially true for Chapter 9 on the floppy disk and controller, one of the last individual computer projects and certainly my personal favorite.*

Design considerations must be resolved in new, creative ways when they are not commonplace. Appreciate the challenge of striving for low component count yet including many features of today's personal computers for the first time ever on a low cost computer. These features include dynamic RAMs, video terminal appearance, plastic case, BASIC in ROM, color, hi-res graphics, paddles, speaker, switching power supply, and more. Normally, large successful companies generate good products. This time it happened the other way around. The Apple II is a product which generated one of the greatest companies ever! It's about time the story of its innards was presented so well.

Steve Wozniak

Woz

Dedication

On behalf of my father, Fredrick, and my brothers and sisters, Lee, Jenny, Tim, Mary, Mike, and Joe,
to my beloved mother,
Frances Janet Nalder Sather.

Acknowledgements

Deborah A. Sather entered the original manuscript to word processor and offered numerous suggestions that made descriptions more clear. She also tolerated the disruption of a wageless author cluttering up the household with his presence and demands.
ITT Gilfillan carried me on leave of absence for seven months before we mutually agreed to terminate my employment.
Jim Aalto provided suggestions and corrections which improved Chapters 4 and 9. Don Worth did the same for Chapters 2 and 9.

preface

The Apple II computer is an adult toy of wonderful depth which brings forth the pride and creativity that resides in all of us. It is a gymnasium for the mind which tests and develops our mental agility. It teaches us about itself, and there is much to learn. Learning about the Apple through creative usage is an exciting and fulfilling, if sometimes frustrating, experience. The frustration comes from the difficulty of the puzzles with which the Apple confronts us. The fulfillment comes from learning to manipulate the powerful machine.

The front line source of information about the Apple is the documentation provided with the computer: the *Apple II Reference Manual*, the *Applesoft II BASIC Programming Reference Manual*, *The Applesoft Tutorial*, and *The DOS Manual*. Magazines and other books provide a wealth of information for the inquiring user. The great self-teaching experience is programming, and persons who choose to program the Apple in any of the various languages quickly increase their familiarity with the features of their computer. Another important source of knowledge is the study of professionally written programs such as the system monitor, listed in the *Apple II Reference Manual*. Gradually, the student acquires a working knowledge of the various hardware features of the Apple. This is important, because it is the hardware features which ultimately define the capabilities and limitations of the machine.

The purpose of *Understanding the Apple II* is to assist the student of the Apple in accelerating the process of assimilating knowledge about his or her computer. This book is hardware oriented, and, therefore, fills a longstanding information gap in Apple literature. Emphasis is placed on achieving an operational knowledge of the Apple, based on an understanding of how the hardware works. Operational knowledge consists of knowing what the Apple can do, knowing how to make it do it, and knowing what a controlling program is making the Apple do. By way of assisting the reader in achieving his goals, the goals of this book are:

1. To provide clear descriptions of microcomputer fundamentals and of the operational features of the Apple.
2. To provide examples of how a knowledge of the operational features of the Apple can be applied.
3. To provide the most thorough hardware reference material available for the benefit of technically oriented readers and for those readers who wish to delve very deeply into the secrets of the Apple.
4. To serve as a text book for Apple based high school or university courses teaching computer fundamentals.
5. To fill information gaps in Apple literature by describing previously undocumented operational features.

Because of the great differences in hardware between the Apple IIe and previous Apple IIs, no attempt is made in this book to document the operation of the Apple IIe. The hardware differences between the two computers are of such consequence that the Apple IIe would itself be the subject of a book similar in scope to this one.* All mention of the Apple or Apple II in this book is meant to refer to Apple IIs prior to the IIe. Yet it is only prudent to

*The book *Understanding the Apple IIe* by Jim Sather is scheduled for release by Quality Software in 1984.

acknowledge here that the Apple IIe is a considerable improvement over the older Apple IIs. In particular some Apple weaknesses mentioned in this book do not pertain to the IIe. This includes most keyboard and text display deficiencies and incompatibility between motherboard ROM and equivalent EPROM.

Persons who will benefit from reading *Understanding the Apple II* are inquiring people who want to spend some time learning about this machine. Generally speaking, this refers to those persons who program the Apple in any language. It is recognized that different people will carry their investigation to different depths. For those who have not the time or desire to reach the greater depths, the Overview, Bus Structure, and Address Decoding chapters (Chapters 1, 2, and 7), as well as the Application Notes at the end of every chapter, are recommended as providing a good foundation for understanding the Apple. As a textbook for students or a learning guide to hard core enthusiasts, cover to cover reading is recommended.

While an inquiring mind is the only qualification required of a reader of this book, certain sections will be difficult for those readers without some background knowledge. In order of descending importance, helpful background knowledge includes understanding of BASIC programming language, hexadecimal and binary number systems, 6502 assembly language, and technical illustrative aids such as timing diagrams, truth tables, and schematic diagrams. It should be noted by all readers that except for the technical aids, they will eventually have to acquire the listed background knowledge if they are to achieve a real understanding of the Apple computer. It is hoped that the nontechnical aids and language in *Understanding the Apple II* are sufficiently descriptive, and that a technical background, although helpful, is not necessary. In general, the later chapters contain more detailed and technical information than the earlier chapters, and the earlier sections in each chapter are less technically oriented. Appendices E and F contain some basic information on number systems and circuit symbols for those readers who come to this book with no previous knowledge of those subjects.

Even though *Understanding the Apple II* is not a programming instruction manual, many programming examples are given in the body of the text which illustrate applications of principles being discussed. Where possible, these examples are written in BASIC so that the clearest possible level of illustration results. In addition, a number of Software Application Notes are included at the end of various chapters which further demonstrate the application of principles. These programming notes are included because understanding the Apple includes a combination of programming knowledge and hardware knowledge. Unless noted otherwise, all software examples are creations of the author and are hereby placed in the public domain. The author requests that he be given credit as the programmer in all reproductions of these programs.

A number of Hardware Application Notes are also included at the ends of chapters. Some of these Notes are hardware projects which demonstrate relevant principles. Other Notes are simple descriptions of modifications to the Apple which enhance operation in some way, such as the SHIFT key modification. Figures 3.11, 3.12, 3.13, 3.15, 4.8, and 4.10 are all original designs of the author. Readers are encouraged to study or build them or integrate them into their own designs. The author requests that he be given credit as the designer in any reproductions or other use of these schematics. D MAnual Controller (Figure 4.8) is being manufactured by the Southern California Research Group, and is available for purchase as noted in Chapter 4.

Several Hardware Application Notes detail modifications to part of the Apple or Apple peripherals. Please read the NOTE OF CAUTION following the Table of Contents before performing any modifications to your equipment. It is recommended that readers unskilled in electronics workmanship who desire a modification have the work performed at a computer dealership or by a skilled friend. Persons who modify their Apples should be able, or know someone who is willing and able, to repair the modified assembly if it should fail.

Understanding the Apple II is the result of an intensive investigation of the Apple II computer by the author. There is no other source of much of the information covered here, and the possibility of error exists on the part of the author. For those errors which do exist, the author is truly sorry.

The Apple II is not a perfect computer, and, in this book, less than perfect features are reported along with the more admirable features. There are many opinions of the author in the body of the text and the reader must rely on his own judgment to evaluate these opinions. Lest the reader get a mistaken impression from harsh comments about certain Apple features, let the record be set straight here. The author highly admires the Apple design and considers this computer to be in a class by itself. Furthermore, he respects the newer products being designed by Apple. To Steve Jobs, Mike Markkula, Rod Holt, and Mike Scott: "Thanks fellas." To Steve Wozniak: "How about an encore, Woz?"

Table of Contents

Chapter 1	THE APPLE II—AN OVERVIEW
Chapter 2	BUS STRUCTURE OF THE APPLE II
Chapter 3	TIMING GENERATION AND THE VIDEO SCANNER
Chapter 4	THE 6502 MICROPROCESSOR
Chapter 5	RAM IN THE APPLE II
Chapter 6	ROM IN THE APPLE II
Chapter 7	ADDRESS DECODING AND INPUT/OUTPUT
Chapter 8	VIDEO GENERATION
Chapter 9	THE DISK CONTROLLER
Chapter 10	MAINTENANCE AND CARE OF THE APPLE II

Glossary

Appendices

A	References
B	Trademarks
C	6502 data
D	BASIC program listings
E	A logic circuits primer
F	A number systems primer
G	Apple II revisional information
H	Historical notes
I	A technical conversation with Steve Wozniak
J	Baseplate and motherboard removal
K	List of Figures and Tables

Schematic Diagrams

Index

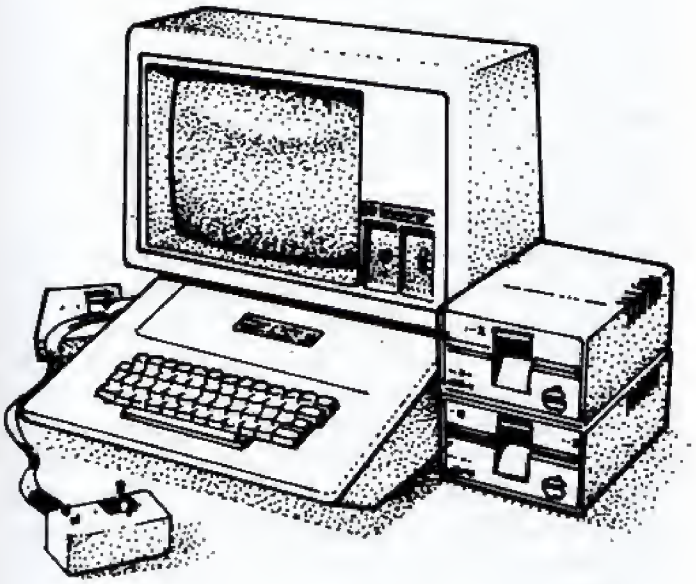
Foldouts

Note of Caution

Several of the Application Notes in *Understanding the Apple II* contain procedures for modifying the Apple II computer and peripheral cards. Modification of your Apple or peripherals may void your warranty if the warranty period has not yet lapsed. It may also increase your out of warranty repair costs should the modified unit fail in the future. The decision to perform any of the modifications described in *Understanding the Apple II* rests solely with the owner of the hardware concerned. Neither Quality Software nor the author bears responsibility for any negative consequence of the owner's decision to perform such modifications.

chapter 1

The Apple II— An Overview



Understanding of any subject is a relative thing. You may, for instance, say that you understand rain and mean that you know to go inside when it is raining. When a meteorologist says he understands rain, however, he speaks of a more significant level of understanding. Understanding of the Apple II can likewise mean various things. You may understand how to use it. More significantly, you may understand how to program it and recognize what programs are making it do. More significantly yet, you may understand how it works. Inquiring reader, you hold in your hands a book which tells you how the Apple II computer works.

Understanding the Apple II is a book of explanations and applications. It contains explanations of how the hardware works and how to make the hardware work. It contains applications which show the advantages of understanding Apple II fundamentals. It contains answers for those who come to it with a less significant level of understanding, because they surely have many unanswered questions.

It is natural to begin a book of explanations with an overview, and so it is done here. But before all else, the author wishes to discuss the first step he had to take in reaching a significant understanding of the Apple II. That step was distinguishing the functions of the BASIC language from the functions of the Apple II hardware.

THE FIRST STEP

The Apple II is a programmable computer. With a program in memory it is an amazing tool, but without a program it is lifeless hardware. The Apple, however, is never without a program, because it has programs built into non-erasable memory (ROM). The most significant program in ROM is a BASIC interpreter through which most people learn to communicate with the Apple. Communication is accomplished very naturally via the keyboard and screen in the easily grasped BASIC language. Persons who choose to can quickly learn to write fairly sophisticated BASIC programs just by reading the BASIC tutorial that comes with the Apple and doing what it says. Good for BASIC. It is a human oriented program which opens computing doors, but...

BASIC is not the computer. BASIC controls the computer. BASIC is a program written in the fundamental language of the Apple computer, 6502 machine code. This concept of one language written in an other is one of the most important in the field of computers. BASIC is a "high level" language written in "low level" 6502 machine code. The Apple can execute programs in any language written in 6502 code, but in fact, during every moment it is turned on, the Apple is executing 6502 code.* These points are made to help the reader take the first step of escaping from BASIC and realizing that the Apple is capable of doing so much more than executing BASIC. BASIC is so easy to use that the Apple owner is held at arm's length from his machine. When an application cannot be performed in BASIC, the owner needs to understand the real features of the Apple.

Familiarity only with BASIC can create misconceptions, because the BASIC language does have similarities and ties to the hardware structure of the Apple. BASIC programs are stored sequential programs which the computer executes. BASIC programs do manipulate the hardware features of the Apple. BASIC in ROM is, in itself, a hardware feature of the Apple. Yet there is an architecture to the Apple which is of a different nature from that of BASIC. This is the architecture of fixed address locations, accessed sequentially by a central processing unit. The BASIC concepts of line numbers for sequential control and variable names for storage of data are very powerful, but they are features of BASIC, not the Apple.

Not just BASIC, but other sophisticated operating systems like the DOS, Pascal, and the system monitor become so much a part of the Apple that it is difficult to distinguish software or firmware from hardware features. These operating systems represent major engineering efforts and they are very much a part of the Apple's power, but no program can make the Apple go beyond its hardware capabilities. If you understand the hardware, you will have crossed a very important threshold for a computerist, awareness of what you can or cannot do with your computer.

*There are actually three instances when 6502 code is not being executed: when a peripheral card performs DMA, when a peripheral card deactivates the READY signal, and when the RESET key is held down or the RESET' line is otherwise held low. These special cases are explained in detail in later chapters.

APPLE II OVERVIEW

The following overview is a brief statement of the hardware features of the Apple II computer. It is not meant to be a description of everything programs can make the Apple do. Rather, it is a description of the basic capabilities with which computer programmers and peripheral designers work. An attempt is made to explain the technical terms that are used, but newcomers to microcomputers should not be discouraged if some points are not absolutely clear to them. The following chapters expand on all topics covered here, and Chapter 2 in particular contains information which will clarify much of Chapter 1.

The Apple II is made up of five physical units: the baseplate and case, the keyboard, the power supply, the speaker, and the motherboard. The speaker, power supply and keyboard are all utility units which plug into the motherboard. It is the motherboard which contains all the uniqueness of the Apple. The motherboard is the Apple, and the Apple is consequently referred to as a single board computer. It has, on one board, a microprocessor, memory, video text and graphics output circuitry, eight peripheral expansion slots, and circuitry for communications with a variety external devices. These features are part of an organized structure centered around the microprocessor.

The Microprocessor and Bus Structure

The brains of the Apple is a 6502 **microprocessor**. A microprocessor, or **MPU** (MicroProcessing Unit), is a single chip logic device capable of executing stored sequential programs.* A microcomputer is a computer which uses an MPU as its fundamental logic processor.

Digital computers operate to a synchronizing beat known as a **clockpulse**, similar to the beat of music, but over ten thousand times as fast. The 6502 operates to a beat which occurs approximately 1,020,500 times a second. We say that the clockpulse frequency is 1.0205 MegaHertz (MHz) meaning 1.0205 million cycles per second. Actually there is a clock pulse jitter, which is described in the timing section of

*A chip is another name for an integrated circuit, or IC. It is a unit with a small body and a number of metal pins or leads, and it contains complex electronic circuitry inside. If you look inside the Apple, you will see many little black chips plugged into sockets on the motherboard. The biggest of these chips is the 6502 MPU.

Chapter 3. Until we get to that point, just say that the 6502 operates at about 1 MHz. This, incidentally, is slow by modern microprocessor standards. There are 3 MHz 6502 MPUs available now, and other MPUs have faster clock pulse rates than that. With a given MPU, the faster the clock, the faster the execution speed.

The structure of the Apple is that of multiple devices which can communicate with the MPU. Once every clockpulse, the MPU outputs the address of the location which is being communicated with, and it transmits data to or receives data from that location. The address which the MPU is putting out is distributed to all addressable devices in the Apple via the **address bus** and data is transferred between the MPU and the addressed location via the **data bus**. Associated and distributed with the address bus is the **read/write control** output of the MPU. Read/write control tells the addressed location whether data will be read from it or written to it.

The 6502 has 16 address outputs, each connected to one line (electrical conductor) of the address bus.* It controls the 16 address lines and the read/write line together by placing a high or a low voltage on each line. The simultaneous condition of the 16 address lines is the 6502 address. The 6502 address is a number between $0 and $FFFF (65535), and the 6502 can access any one of the $10000 (65536) addressed locations in that range.

The 6502 has eight data input/output lines, each connected to one line of the data bus. It controls the eight lines when writing and monitors the eight lines when reading, and the simultaneous condition of the eight lines is the 6502 data word. Like the address lines, each of the data lines is brought to a high or a low voltage when information is passed. Each line can be one of two states (high or low), so the information is said to be **two state**, or **binary**. Other common ways of referring to the two states of binary information are true/false, one/zero, and on/off.

A unit of binary information is a **bit**. Whether a line is high or low at a given instant is a bit of information. The 6502 reads or writes and manipulates information eight bits at a time and is therefore classified as an 8-bit MPU. A group of eight bits is a **byte**. The 6502 manipulates and transfers data, one byte at a time, to an addressed location in the Apple bus system.

Most locations which the MPU addresses are **memory** locations. Memory contains the stored program which the MPU is executing and about half of the MPU's time is spent fetching that program. The program is stored sequentially, so fetching the program by the MPU simply involves incrementing the address output while reading the data input and interpreting it as a sequential program. While not fetching the program, the MPU is executing it. This execution involves logical manipulation of data, storage of data at or loading of data from addressed locations determined by the program, changing the program fetching location to somewhere other than the next sequential address, or any combination of these and other functions.

Not all locations addressed by the MPU are memory locations. Program instructions fetched from memory may cause the MPU to address non-memory locations such as the speaker or keyboard. A memory location responds to a read at its address by placing data on the data bus. The speaker responds to a read or a write at its address with sound. The MPU thus controls the speaker via the address bus in an **address decoding** process. Address decoding is the only way a 6502 can control other devices, so all programmed control of Apple devices is via address decoding.

Memory

General purpose microcomputers require two types of memory—memory you can change (**RAM**) and memory you can't change (**ROM**).* RAM is necessary so you can store general programs and data. ROM is necessary so the computer has a program to run when it is first turned on.

Both ROM and RAM are random address memories meaning any specific memory location can be accessed at its specific address. Computer memory is like thousands of light bulbs, each of which may or may not be glowing. If the memory is random access, the microprocessor can communicate with any light bulb it chooses by calling its number. It can, for example, check if light bulb number 25,765 is glowing or not. This is analogous to reading memory. Telling light bulb number 7,682 to not glow is analogous to writing to memory; the MPU is altering the state of light bulb 7,682. RAM and ROM are functionally identical except that ROM is fixed

*As described in Chapters 2 and 4, the 6502 is not connected directly to the address bus and data bus. It is connected to the buses through isolating devices which give the Apple II a DMA (Direct Memory Access) capability and allow the 6502 to communicate with the large number of electronic devices connected to the address bus and data bus of the Apple II.

*ROM stands for Read Only Memory, which is accurate, and RAM stands for Random Access Memory, which is the most famous misnomer in all of computer jargon. Both read only memory and read/write memory in the Apple are random access memory, and this book refers to them by their conventional labels, ROM and RAM.

as if it was etched in stone. You can't turn the light bulbs on or off. You can only check to see if they are on or off.

The MPU can not really tell whether a light bulb is glowing or not, but it can tell whether the voltage on a line is high or low. RAM is capable of storing the high/low state of its data input when the MPU writes data to a RAM address. Both RAM and ROM are capable of bringing their data outputs high or low in accordance with stored data when the MPU reads data from a RAM or ROM address. In a **positive logic** system like that of the Apple, storing or reading a high voltage is thought of as storing or saving a "1". Storing or reading a low voltage is thought of as storing or saving a "0".

Since the 6502 is an 8-bit MPU, memory must be organized so that it is accessed eight bits, or one byte, at a time. The Apple motherboard has sockets for 49,152 bytes (393,216 bits) of RAM. This is normally referred to as 48K of RAM meaning 48 Kilobytes. The Apple uses **dynamic RAM** which must be **refreshed**. Memory refresh must occur on a periodic basis or dynamic RAM will not work. It's like a fire that goes out unless someone is constantly pumping the bellows. Dynamic RAM is nice because it's inexpensive, but it requires a lot of external circuitry to support the refresh requirement. The Apple motherboard fully addresses and supports 48K of RAM in every way including refresh.

The Apple motherboard has sockets for 12,288 bytes (98,304 bits) of ROM, commonly referred to as 12K. Each ROM chip contains 2K of information so there are six sockets. 2K are taken up by the **monitor** ROM. This chip tells the Apple what to do when it turns on and contains valuable utilities which make the Apple hardware accessible to its user. The rest of ROM is taken up by either **Applesoft** or **Integer BASIC**. Applesoft is the newer BASIC written by Microsoft Corporation for the Apple. Integer BASIC is the original BASIC written for the Apple. Integer BASIC is a smaller program than Applesoft so there are two empty ROM sockets permitting a total of 4K of user ROM on the Integer machine. The Apple with Applesoft on the motherboard ROM is referred to as the **Apple II Plus** to distinguish it from the Integer Apple. This book refers to both machines as the Apple or Apple II.

Peripheral Slots

The Apple peripheral slots are similar to a **card cage**. What is a card cage? A card cage is a very versatile physical package for microcomputers and other electronic circuits. It is a row of slots mounted close together into which printed circuit cards are plugged. Behind the slots are hundreds of wires connecting the slots together in accordance with the design purpose. Card cage architecture is like a house with an intercom system. Just as communication is possible between various rooms of the house, communication is possible between the various cards plugged into the card cage. Each slot in the card cage is a different station in the intercom system.

In a card cage microcomputer, part of the wiring which interconnects the slots is a multiline address bus and data bus, similar to the buses on the Apple II motherboard. A microprocessor board can be plugged into any slot, from where it can control communication in the card cage via the address bus. A very nice modern card cage micro would have a multifunction single board microcomputer in one slot and a variety of devices in the other slots. The Apple is exactly that computer, turned inside out. Instead of mounting the main logic board in the card cage, they mounted the card cage on the main board.

The Apple "card cage" consists of eight **peripheral slots** mounted on the back of the motherboard. The address bus and data bus are connected to all the slots, making them addressable extensions of the Apple's basic communication system. Each slot has a part of the 6502 address range assigned exclusively to it, so the programs can make the 6502 access a peripheral slot just as if it were a group of memory locations.

Some important 6502 input control signals are tied to pins on the peripheral slots. They are RESET', READY, NMI' (Non-Maskable Interrupt), and IRQ' (Interrupt ReQuest). These signals are all described in greater detail in the 6502 section of Chapter 4. Their connection to the peripheral slots means that the processor can be interrupted, stopped, started, and reset from any peripheral card. It also means that any peripheral card can be designed to respond to these control signals. For example, pressing RESET at the keyboard resets the 6502 and additionally turns off the floppy disk drive. The disk drive controller is designed to respond to the RESET' signal which occurs when RESET is pressed. RESET', incidentally, is read "reset prime." In this book, the prime behind the name of a logic term is used to signify that a signal is active or true when a low voltage is present.* It is an aid to understanding

*Most published computer literature will overscore a logic term, rather that placing a prime symbol behind it, to signify that it is active when low. In using the prime notation, *Understanding the Apple II* is following the convention used by Apple in their *Apple IIe Reference Manual*. In addition to signifying that a term is active when low, the prime symbol following a logic term can mean that the inversion of that logic term is being referred to. Please see Appendix E for further discussion of this subject.

the logic functions of a given signal. Knowing this, you could guess from the second sentence of this paragraph that the 6502 is interrupted and reset by low voltages on the NMI', IRQ', and RESET' line and enabled by a high voltage on the READY line.

Another peripheral slot signal which affects the 6502 but isn't connected directly to it is the DMA' signal. **DMA** stands for **Direct Memory Access** and refers to direct memory access from the peripheral slots. The DMA' line does a bit more than giving the slots access to memory, however. It allows a card in a slot to isolate the 6502 from the address bus and data bus and take control of communication in the bus system. This means that a peripheral card can control all hardware features of the Apple. It is as if you could plug a Suzy brain into Johnny and have the Suzy brain control Johnny's body, a concept much in vogue in some circles.

There are signals connected to the peripheral slots other than those that have been mentioned. They provide various capabilities so peripherals can be designed to be fully integrated into the Apple structure. These signals include timing and control inputs, power supply voltages, and control signals decoded from address ranges on the address bus. The purposes of these signals will be fully explained in later chapters.

Video Output

The primary output of the Apple II is video. The video is compatible with the video in an American TV set. Motherboard "Eurapple" jumpers can change the video for black and white compatibility with European TV systems, but a "Eurocolor" card in slot seven is necessary for European color compatibility.

The problem with video output is that the input to a television is not video but RF (Radio Frequency) modulated by video. This means that you can use the Apple with a television set, but the input to the television must be an RF signal modulated by Apple video. Generation of the RF signal and modulation is accomplished in a user supplied modulator. Another name for the user supplied modulator is a pain in the neck.

There are three basic Apple video modes: **TEXT**, **LORES** graphics (LOw RESolution colored blocks) and **HIRES** graphics (HIgh RESolution colored points). The displays are stored (mapped) in memory so that video is generated by processing data from memory.

All video modes use memory scanning to generate video. This means that certain areas of RAM are designated as display memory. The designated areas are:

TEXT/LORES	Page 1 - 400-7FF	(1K memory)
TEXT/LORES	Page 2 - 800-BFF	(1K memory)
HIRES	Page 1 - 2000-3FFF	(8K memory)
HIRES	Page 2 - 4000-5FFF	(8K memory)

As an example, assume that the computer is in TEXT mode, PAGE 1. Then memory in the range $400-$7FF will be scanned approximately 60 times a second and the data in that memory area will be processed for video output. Part of display memory is always being scanned while the computer is on. The Apple is designed so that this constant scanning satisfies the refresh requirement of the dynamic RAM.

An important consequence of the Apple's display implementation is that the video display steals memory from the user. The programmer must program around the display areas if he intends to use the associated displays.

Scanning for video output is not performed by the MPU. It is performed by the **video scanner**, also referred to as the video scan counter or video synchronizer. Signals from the video scanner are used to develop television sync which means that the memory scan in the Apple and the electron beam scan in the television tube are in sync. Memory is scanned one time for every time the picture is scanned.

The scanner accesses RAM in a way that is completely transparent to the MPU. During the first half of every 6502 cycle period, the video scanner addresses RAM. During the second half, the 6502 addresses RAM. The scanner access to RAM is always a read access and the data which comes from RAM during the scanner access is processed by the **video generator** to make video. The 6502 access can be either read or write and, on some cycles, the 6502 may not access RAM at all. When the 6502 does access RAM, RAM is accessed twice every 6502 cycle and is therefore accessed at 2 MHz.

The programming method for controlling the Apple display is to compute or look up the address of the desired screen location in the appropriate memory display area and modify that address to the

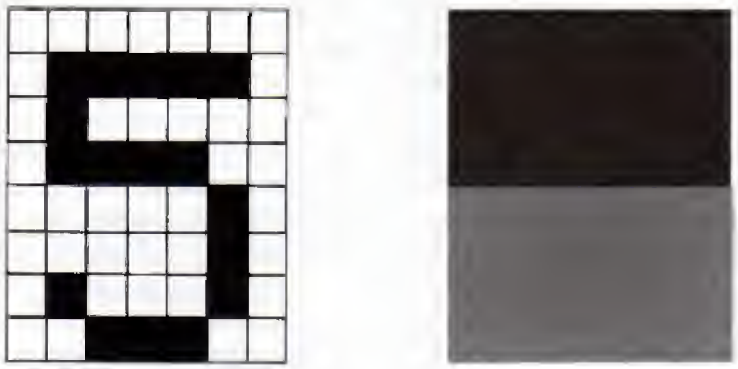


Figure 1.1 TEXT and LORES Graphics.

programmer's purpose. The video scanner scans the display area determined by the display mode and the resulting memory data is processed as text or graphics as determined by the display mode.

Text is stored in ASCII code (American Standard Code for Information Interchange). In addition to ASCII, code for normal display (black on white), inverse display (white on black), or flashing display (alternating normal and inverse) are stored for each text character. One character is stored per byte of display memory. As text is scanned, the coded data from memory is translated to 5x7 dot matrix video in normal, flashing, or inverse format. There are 64 displayable text characters (one is the space).

The TEXT display is 40 columns by 24 lines. Only uppercase is available on older Apple IIs but new ones have upper/lower case screen capabilities.* A variety of 80 column upper/lower case display boards are available for the peripheral slots.

LORES graphics is a programmable display of 40 columns and 48 rows of colored blocks. Each block can be any one of 15 colors including black and white. Apple claims 16 colors but the two grays are identical in color and luminance. LORES is mapped in the same display area as TEXT so memory scanning is identical in the two modes. In LORES, rather than converting ASCII to video, the video generator processes the bit pattern directly into video. The code for each LORES block requires four bits, thus there is code for two blocks in every byte of display memory. Also, there is a direct correspondence between the screen location of a pair of LORES blocks and one text character as shown in Figure 1.1.

HIRES graphics is a programmable array of 280 columns and 192 rows of dots. Because of the way video is generated in the Apple, the color of any dot is dependent on its horizontal position. To draw a violet horizontal line, for instance, every other dot in one row is turned on. To draw a violet figure, only half of the columns of dots can be turned on. This is also true of the other HIRES colors; green, orange, and blue. There is only 140 x 192 resolution when drawing these four colors. White (the absence of color) and black (the absence of luminance) can also be displayed. The 280 dots in any row are divided into 40 groups of seven dots. Each group of seven dots may be shifted together horizontally one half of a dot position, changing the colors of any colored dots in that group of seven. Thus, there are 560 horizontal dot positions in each row but only 280 dots are independently programmable. This brief statement of HIRES graphics capabilities is probably just enough information to let the reader know that the subject of HIRES is complex. Full understanding is possible in the light of more detailed analysis and HIRES is covered in greater detail in Chapter 8. For now, let us summarize the graphic resolution of the Apple II screen as follows. There is 140 x 192 dot resolution in six colors. There is 280 x 192 dot resolution if color is ignored. There is 560 x 192 dot resolution for some applications.

The HIRES memory display area is much larger than the TEXT/LORES area: 8192 bytes for a single display screen. This is the hardware cost of high resolution. Each byte of display memory contains the on/off conditions of seven dots plus the shift control bit which determines whether that group of

*In Revision 7 and later Apples, an upper/lower case screen display capability can be realized by replacing the A3 TEXT ROM. Please refer to the related Application Note at the end of Chapter 8.

seven dots is shifted half a dot or not. During the HIRES memory scan, the data from memory is essentially shifted straight to the monitor or television modulator via the VIDEO output line.

Apple programs select the video mode via four programmable toggle switches, or **soft switches.** The four video mode soft switches are:

GRAPHICS /	TEXT
LORES /	HIRES
PAGE 1 /	PAGE 2
NOMIX /	MIX

The first two switches select TEXT, LORES graphics, or HIRES graphics. PAGE 1/PAGE 2 simply selects between a primary memory display area and a secondary memory display area for each mode. PAGE 1 is normally used in all modes, but use of PAGE 2 may suit the programmer's purpose. The most common use of alternate page displays is to eliminate flicker in graphic animation. The programmer updates an animated picture on the page that isn't being displayed, then changes the display by switching pages so that the updated picture is displayed. PAGE 2 of TEXT is rarely used because it serves little purpose, is not supported by firmware, and interferes with BASIC. PAGE 2 of LORES graphics is rarely used for the same reasons.

The fourth programmable switch is MIX/NO MIX. MIX refers to mixed text and graphics. If MIX is selected and GRAPHICS is selected, there are four lines of text at the bottom of the screen and LORES or HIRES graphics on the rest of the screen. This is very useful, as far as it goes, because there are many times when the graphics programmer needs to enhance a display with text. Four lines at the bottom turn out to be inadequate for many purposes, though. The HIRES screen has good enough resolution to draw text, and several programs are available that make it relatively easy to place upper/lower case text on the HIRES screen via graphics. This type of text can be drawn alongside graphics to enhance graphic displays.

The Keyboard

The keyboard is the primary human input to the Apple II (as opposed to storage media input such as disk or cassette). Virtually all human alphanumeric input is via the keyboard, and the MPU of the Apple II spends the majority of its life cycling through a little firmware routine called KEYIN. This routine samples the keyboard to see if a key has been pressed and increments a random number counter. It checks the keyboard at a rate of about a quarter of a billion times an hour. If anyone asks you what an Apple does, you can answer "mainly, it checks to see if a key has been pressed."

Enough silliness. The keyboard input is uppercase only and there are a total of 52 keys on it. SHIFT keys shift between numerals and symbols. While SHIFT-M, SHIFT-N, and SHIFT-P produce codes which are interpreted as special characters, other alphabetic characters produce the same code whether SHIFT is pressed or not. The layout of the keys is like it is on a teletype. This means that the alphabetic and numeric characters are arranged like those on a typewriter, but some of the symbols are in rather strange places. Keyboard input to the computer is ASCII. Since the keyboard input and text output are both ASCII, it is fairly easy to display characters on the video as they are entered from the keyboard. This is done by keyboard input routines in the Apple firmware. There are, however, three screen displayable text characters which cannot be input from the keyboard. They are left bracket, backslash, and underscore.

Special function keys on the keyboard are ESC, CTRL, SHIFT, RESET, REPT, RETURN, left arrow, and right arrow. CTRL and SHIFT modify the ASCII produced by other simultaneously pressed keys. RESET is tied to the 6502 RESET' input. REPT causes any simultaneously held key to be repeated about 10 times a second. ESC, RETURN, left arrow, and right arrow produce ASCII which must be interpreted by the controlling program. The code for ESC is unique, but RETURN, left arrow, and right arrow are identical to CONTROL-M, CONTROL-H, AND CONTROL-U respectively.

In summary, the Apple II keyboard is notable only for its limitations. These are: upper case only alphabet, lack of dedicated cursor move keys, lack of user programmable function keys, and lack of a numeric entry keypad. It is a little league keyboard on a big league computer.*

*The keyboard of the Apple IIe is greatly improved over the earlier Apples. The author's negative overall opinion of the Apple II keyboard applies only to Apple II computers that predate the Apple IIe.

Other I/O

I/O is Input/Output. Our point of reference for this discussion is the motherboard, meaning that we speak of input to the motherboard and output from the motherboard. The peripheral slots give the Apple an extremely versatile I/O capability, but there is a good deal of additional I/O circuitry built into the Apple. The keyboard input and video output are the most significant motherboard I/O. Additionally, there are some useful serial I/O ports.

Serial data is data on one line. This is opposed to **parallel** data on more than one line (eight lines for instance). To transfer eight bits serially, each bit of information is placed on the same line one after the other. This takes eight times as long as an 8-bit parallel transfer but requires only one connecting wire. The keyboard is a parallel input. The video is not a simple digital output but a mildly complex signal output with a serial data component. In addition to these I/O capabilities there are **eleven serial I/O ports** and **four resistance sensitive timer inputs.**

The **speaker** output is a serial output port connected to a speaker through an audio amplifier. The **cassette** input and output are serial data transmitted via audio phone jacks on the motherboard accessible from the back of the case. They are designed to connect directly to the earphone output and microphone input of a common audio tape recorder. Firmware routines in the Monitor ROM read and write cassettes in Apple's storage format.

Usage of 5 1/4 inch floppy disks is so prevalent that cassette storage is rarely used by most Apple owners. Floppy disk I/O is not a built-in capability of the motherboard, so the disk electronics are contained in the drive and on a peripheral card called the disk **controller.** Disk data is transferred in parallel between the MPU and the controller and serially between the controller and the drive. Control of the disk I/O requires an extensive program, and the most commonly used program of this nature is the **DOS** (Disk Operating System) supplied by Apple Computer, Inc.

The other serial I/O signals are **TTL** (Transistor Transistor Logic) compatible. TTL is a very common logic family of integrated circuits used for digital logic. The logic devices on the Apple motherboard are either TTL or interface directly to TTL.* A TTL output port will communicate directly with TTL compatible devices. The cassette and speaker ports are not TTL ports and really are not very useful for transfering general digital information to anything but a tape recorder or other audio device. TTL devices operate with two voltages corresponding to the two states of digital logic. The TTL low voltage is 0 to 0.8 volts. The TTL high voltage is 2.4 to 5 volts. These are the two voltage levels which represent digital information throughout the Apple.

There is a 16-pin DIP (Dual In line Package) socket on the Apple motherboard which is generally called the **game I/O connector**. A set of two paddles or a joystick is normally connected here but there is a capability for multiple uses. Four of the pins are **annunciator** outputs. These are independently programmable TTL levels. The programmer can make each level go high or low at will. A fifth TTL output is called a **strobe**. This output is high unless a program triggers it. It then goes low for just 0.5 microseconds (half a 6502 cycle) then returns to its normal high state.

There are three TTL input ports on the game I/O connector which can be read by a program. Two of these are normally connected to **pushbuttons** on the joystick or game paddles.

The **paddles** themselves are just potentiometers (variable resistors). The **joysticks** are two potentiometers in a special mechanical arrangement. These potentiometers are connected through the game I/O connector to two of the four resistance sensitive timers. The way this works is that a program resets the four timers all at the same time. Under program control, the CPU can time and check when any or all of the timers have timed out. The time for any timer depends on its resistance input from the game I/O connector which is varied by the person massaging the joystick.

The game I/O connector can be used for a lot more than games, but not easily. The multiple I/O on one socket implies that any device which plugs into the socket should have a built-in extension socket so other devices can use the remaining I/O lines. Unfortunately, no one designs them this way (see Figure 1.2). Originally, Apples were supplied with a paddle set which was two paddles, each with a pushbutton, connected to a single 16-pin DIP plug. With these paddles plugged in, the remaining I/O—four annunciators, two timers, one TTL in, and one strobe out—becomes inaccessible. This seems to have set a "no extension" design precedent which has stifled the development of hardware for the game I/O connector. Such products as programs for four paddles or two joysticks don't exist. If you want to measure temperature at a timer input, you must

*Most of the TTL chips in the Apple are LSTTL (Low Powered, Schottky-Barrier diode clamped TTL). ROM, RAM, and the 6502 are TTL compatible MOS (Metal Oxide Silicon) chips.

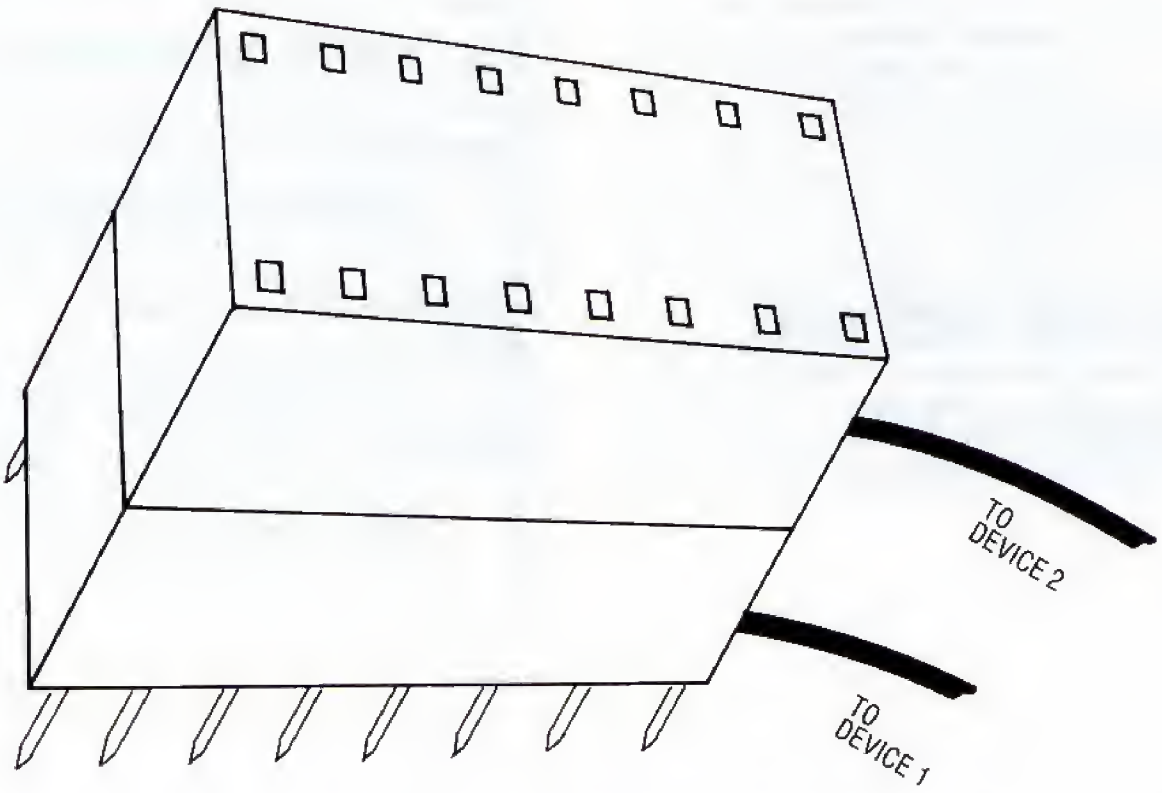


Figure 1.2 How the Game I/O Plug Should Be Designed.

disconnect your joystick and plug in your temperature sensitive resistor, or buy a game I/O extender. Thus, in most installations, the game I/O connector is merely a socket for two paddles or a joystick with two buttons.

The Power Supply

Household power in the United States is 115 Volts AC (Alternating Current). Most of the circuits in the Apple require +5 Volts DC (Direct Current). A power supply is a device which converts AC power to DC power. Televisions, stereos, computers and such all have power supplies built in.

The power supply in the Apple generates +5VDC, -5VDC, +12VDC, and -12VDC. These voltages are distributed throughout the motherboard to any device that needs them. Additionally, all four voltages are available at the peripheral slots to supply power to peripheral cards. Some Apple IIs were supplied with a 110VAC/220VAC switch for compatibility with foreign household power.

SUMMARY

The Apple II is a single board, 6502 based microcomputer with built-in memory and video generation circuitry. The board contains eight peripheral card slots which give the Apple II expansion capabilities comparable to more expensive card cage microcomputer designs.

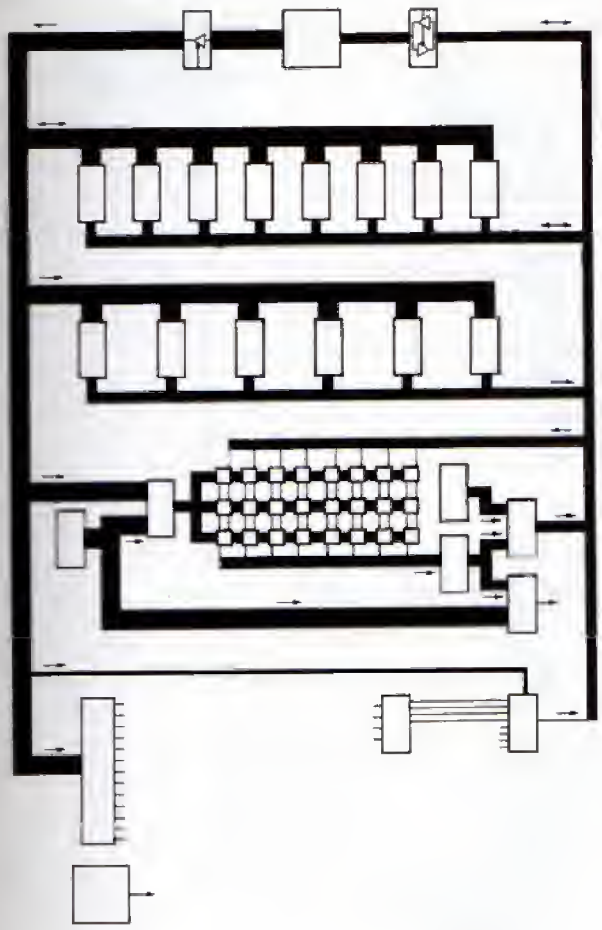
The 6502 operates at 1.0205 MHz. IRQ', NMI', RESET', and READY signals to the 6502 are connected to the peripheral bus. The DMA' signal enables peripheral cards to isolate the MPU from the rest of the motherboard. This enables control of the Apple II from secondary MPUs or other DMA devices in the peripheral slots. MPU control of the various hardware features is via address decode.

The motherboard contains sockets and addressing for 49152 bytes of dynamic RAM. 12288 bytes of firmware include Applesoft or Integer BASIC and a system monitor containing a number of important utilities.

The video output is compatible with a video monochrome or color monitor. It can be used with a home TV when connected through an inexpensive modulator. Text is upper case only, 40 characters by 24 lines, 5 x 7 dot matrix representation. Graphics modes include 40 by 48 LORES block mode in 15 colors, 140 by 192 HIRES point mode in six colors, and 280 by 192 HIRES point mode in black and white. Some capabilities exist for mixing text and graphics. The video display in all modes is mapped in certain areas of RAM. Video display circuitry continuously scans one of four possible memory areas while memory output is processed to generate video. RAM addressing is time shared between the system address bus and the video scanner. 6502 access to RAM alternates with video scanner access so, while the 6502 operates at 1 MHz, RAM is addressed at 2 MHz. In the process of scanning memory for video output, the memory is refreshed.

In addition to I/O capabilities inherent with the peripheral slots there are: a cassette input port, a cassette output port, an on board speaker, four TTL control outputs, one .5 microsecond TTL output strobe, four resistance sensitive timer inputs, three TTL inputs and a keyboard. The keyboard contains 52 key switches with uppercase only on alphabetic characters.

The built-in Apple power supply provides +12V, -12V, +5V, and -5V. These voltages are distributed throughout the motherboard and to the eight peripheral card slots.



chapter 2

The Bus Structure of the Apple II

There are many signals distributed throughout the Apple II, but the most fundamental data transfer takes place on the **data bus**, and the most basic control information is distributed via the **address bus**. To understand how the Apple and other microcomputers really work, it is very important to understand the bus structure. Fortunately, it's not that hard to understand. The basic concepts of the bus structure are within the grasp of nearly everyone who uses a microcomputer.

The bus structure is a natural starting point for learning what really goes on inside the Apple computer. Discussing the bus structure will lead naturally to the discussion of the other microcomputer elements that the bus is connected to. First, though, we need to find out what a bus is and how it is used.

COMPUTER BUSES AND THREE STATE LOGIC

Logic signals in the Apple are distributed electrically via conductive paths on the motherboard. When a number of signals are grouped functionally and distributed throughout a microcomputer, they are collectively referred to as a bus. Physically, then, a bus is an electrical distribution of multiline information. In the Apple, the **address bus** is a **sixteen** line electrically distributed information group, and the **data bus** is an **eight** line electrically distributed information group.

Some devices connected to a bus are strictly receivers of information. ROM is like this in its connection to the **address bus**. Receivers respond to the high/low information on the lines of the bus without appreciably affecting the bus information. Electrically speaking, the receiver input presents a high impedance to the bus which enables other devices to bring the bus lines high or low. If impedance is a new word to you, it may help to think of high impedance as high isolation.

Some devices on a bus must be information transmitters capable of bringing the bus lines high or low. If more than one information transmitter is connected to a bus, each transmitter must be able to disconnect itself from control of the bus by presenting a high impedance to the bus. Only one device can control the bus at a time. Instead of two state, the outputs of these devices are said to be three state or

LINE DRIVER
Information is transmitted to the bus by a device with tri-state outputs.

LINE RECEIVER
An information receiver presents a high impedance to the bus.

LINE TRANSCEIVER
A bidirectional connection to the bus must present a high impedance to the bus when in receive mode.

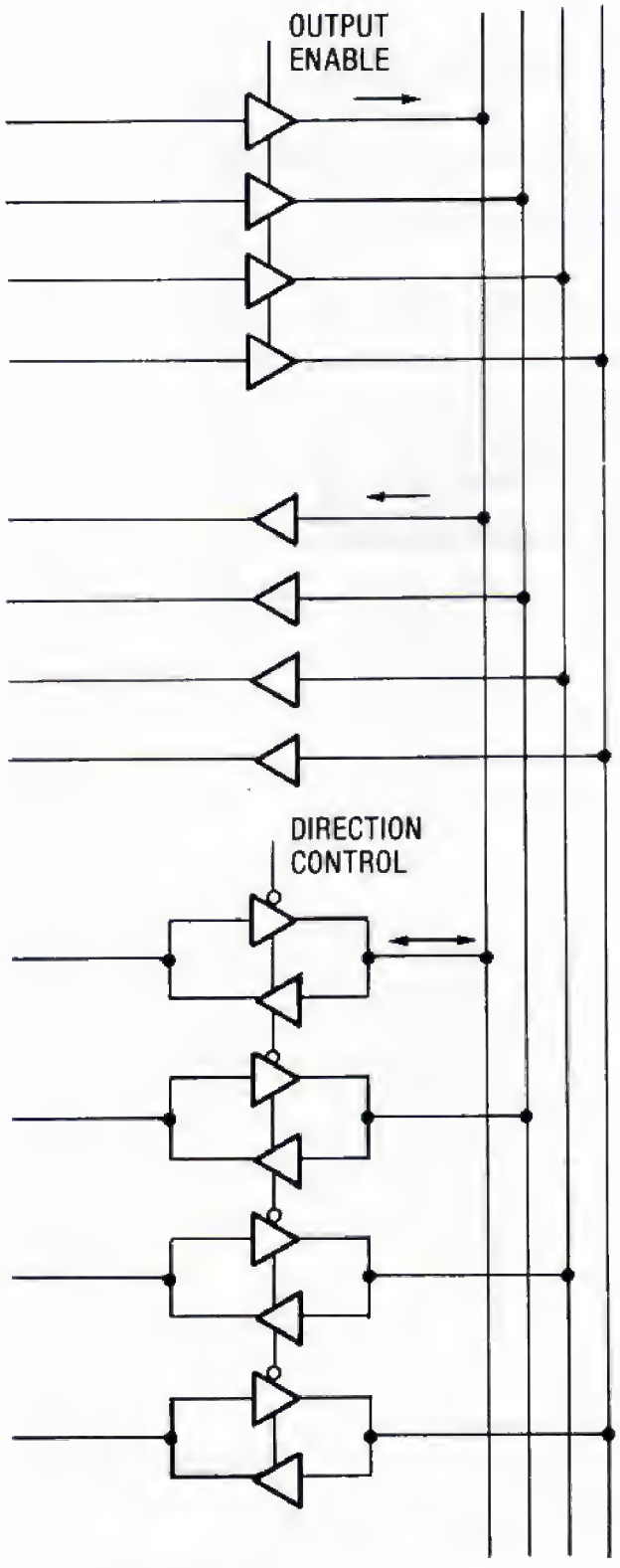


Figure 2.1 A Hypothetical Four-Line Bus.



Figure 2.2 Basic Microcomputer Building Blocks.

tri-state. The three states are high voltage, low voltage, and high impedance. All information transmitters to the data bus of the Apple are capable of presenting these states to their bus connections. The ROM output to the **data bus** is a typical three state output.

A third type of device, capable of transmitting to or receiving from a bus, is called a **transceiver** (**trans**mitter / **receiver**). The MPU, for instance, receives (reads) data from and transmits (writes) data to the data bus via an eight bit transceiver. While the MPU is reading, the transceiver presents a high impedance to the data bus so the addressed device can place data on the data bus. While the MPU is writing, the transceiver controls the data bus.

Figure 2.1 shows a hypothetical four line bus. The symbols shown are schematic representations of a tri-state line driver, a line receiver, and a line transceiver. A triangle represents a single line driver. Triangles with a control line coming in from the side are tri-state line drivers. A little circle at a control input to a triangle means that the input is active when its voltage is low. Here is a truth table for the tri-state line driver shown in Figure 2.1:

INPUT	OUTPUT ENABLE	OUTPUT
Any	Low	High Impedance
High	High	High
Low	High	Low

The control line either enables the high/low output or forces the output to high impedance. The high/-low output, when enabled, follows the input.

It can be seen that the **output enable** controls of the various information transmitters are the key to cohesive control of the bus. For a bus with many possible information transmitters, like the data bus of the Apple, there has to be some intelligent management of the various tri-state output enables. We will see shortly how this is accomplished. In the following discussions, remember that when a device like a ROM chip responds to an address prompt by placing data on the data bus, this is accomplished via an output enable to the tri-state outputs of the ROM chip.

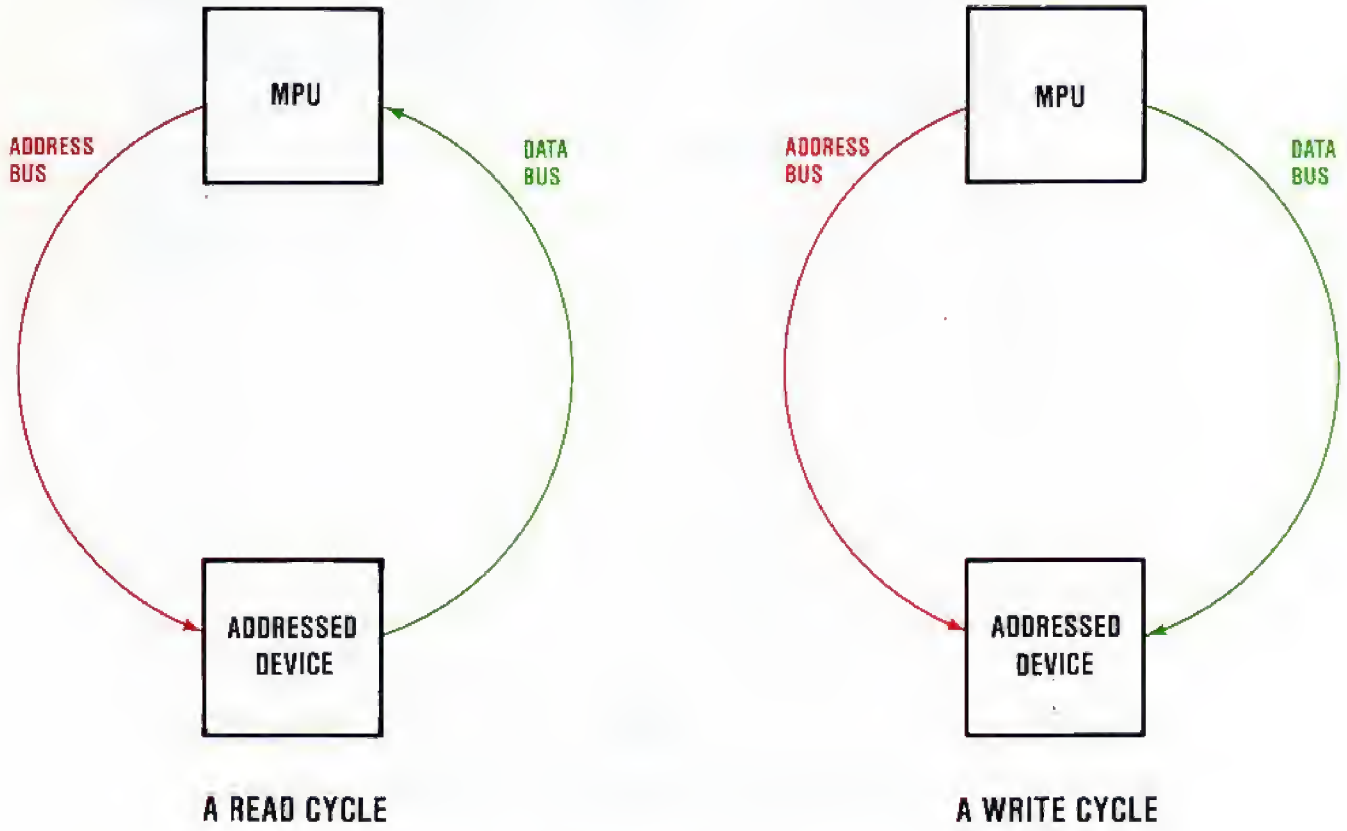


Figure 2.3 Communication on the Bus System.

Figure 2.2 shows a highly simplified diagram of the bus structure of the Apple II. There are two distinct multiline signal paths: the **address bus** and the **data bus**. The R/W' line (Read/Write control) is shown separate but in the same color as the address bus. R/W' can be thought of as an extension of the address bus controlling the direction of data flow on the data bus. Communication takes place on every 6502 cycle between the MPU and an addressed device. Data flows between the MPU and the device in a direction determined by the R/W' line. The MPU controls the R/W' line and the address bus.

Figure 2.3 shows the two types of bus access which occur in the Apple II. In a read access, the MPU places an address on the address bus and reads the data bus. In a write access, the MPU places an address on the address bus and places data on the data bus. This establishes a system of data bus control that had to be implemented in the design of the Apple. The control system works like this:

1. When the R/W' line is low (write access), all inputs to the data bus are disabled except the MPU.
2. When the R/W' line is high (read access), all inputs to the data bus are disabled except the device which is addressed.

This system concept keeps traffic flow orderly and is a basic feature of microcomputer design.

The only remaining points to be made about buses involve semantics. The **peripheral slots** are sometimes referred to as the **peripheral bus** or the **Apple bus**. In fact, the wiring of the slots fits our description of a bus as a functional group of distributed signals. The slots are a bus whose distributed signals include the address bus, the data bus, and other signals. Up to this point, the discussions have avoided calling the slots a bus only to avoid confusion between the card cage bus and the more basic address bus and data bus. The connections to the RAM and ROM chips form two more distributed signal groups that can accurately be referred to as the RAM bus and the ROM bus. This book will continue to use the word "bus" to refer to the address bus, the data bus, and the extensions of these two basic communications paths. The peripheral bus, RAM bus, ROM bus, and other distributed signals will be referred to using other terminology.

By this time the reader should understand the concept of the bus as a communication path. We will now move on to how microcomputers in general and Apples in particular perform their functions in a bus environment.

THE PIGEONHOLE COMPUTER

There is an old analogy for understanding digital computer operation which you don't see often enough in personal computer instruction literature. It possibly is not that helpful for understanding BASIC programming, but it is very much like the way a microcomputer works.

The analogy goes like this. A computer is like a gigantic row of pigeonholes with pieces of paper in them. Each piece of paper has an instruction on it. There is a man who goes to each pigeonhole, one after the other, reading the instructions and doing what they say. The man always gets the next instruction from the next pigeonhole in the row unless an instruction tells him to go to some other pigeonhole.

That's the pigeonhole computer. The man is executing a stored sequential program. The man is the microprocessor. The row of pigeonholes is computer memory. The instructions are the program. The microprocessor is smart enough to sequence through memory and do what it's told, but it **has** to be told. It has to have a program.

THE MPU, RAM, AND ROM

The microprocessor is the engineering marvel which made all the home computers possible. The 6502 MPU is what executes the programs in the Apple. Viewed from the outside, its capabilities include manipulation of the address bus and R/W' (Read/Write') control, writing data to the data bus, reading data from the data bus, logical and arithmetic manipulation of data, and response to various control inputs. These all add up to execution of a sequential program that comes from the data bus.

You see, the man from the pigeonhole computer resides inside the MPU. The little guy has got this control line called R/W' and he can put any address from 0 to 65535 on the address bus. He uses the R/W' line to tell the outside world whether he's reading from or writing to the data bus. He uses the address bus to tell the world where he wants the read data to come from and the write data to go to. There are plenty of things this man can do, but his most favorite thing in the whole world is to increment the address bus and read the results on the data bus. While he's reading, this little workaholic interprets

the data he reads as instructions. If there is an outside device that is responding to his address prompts with a valid sequential program, he will flat out execute the program. This means that you can exploit his insatiable reading appetite and get him to do what you want if you're smart enough. That's all any microcomputer designer ever really expects from an MPU.

The key requirement above was an outside device responding to the address prompts. This device is memory: ROM or RAM. All of the addressing on the Apple address bus is parceled out to various devices. RAM gets addresses $0-$BFFF. ROM gets $D000-$FFFF. The peripheral slots are controlled by $C080-$CFFF. $C000-$C07F is divided up among the keyboard and cassette and all the other built-in devices. If the 6502 happens to be executing a program in the $D000-$FFFF range, then **ROM** is responding to the addressing with a series of data which the 6502 is interpreting as a program. If the ROM program tells the MPU to store a byte of data at $400, the MPU takes a microsecond to bring R/W' low, set the address bus to $400, and place the pertinent data on the data bus. The data is accepted by address location $400 which is in **RAM**. That pigeonhole of RAM owns address $400 just as sure as your mailbox has a unique mailing address. Inside RAM, inside ROM, all along the address bus, address decoding takes place every 6502 cycle to enable only one of 65536 possible addresses.

The 6502 is continually executing a program while power is applied. If it gets lost and tries to execute a program where no program exists, it interprets whatever jibberish is appearing on the data bus as a program and executes it anyway. An unstoppable program-executing machine like this has to have a starting point when you turn the computer on. It also needs a way to start from scratch when it gets lost. This starting point is the RESET' input to the 6502.

The RESET' input to the 6502 goes low when the RESET key is pressed, when a peripheral card makes it go low, or when the computer is turned on. Any one of these occurrences makes the 6502 stop what it's doing, load the address of the next program step from locations $FFFC and $FFFD in ROM, and start executing at its special address. The contents of $FFFC and $FFFD are the low and high bytes of the reset vector.* The reset vector in the Autostart ROM is $FA62, the address of the reset routine. This is an essential feature of microcomputer design—the power-up routine in ROM. You might say it guarantees that the 6502 always gets out of bed on the right side.

The other routine which a microcomputer always has in ROM is a routine to load data from a storage device into RAM so that execution of saved programs is possible. The Apple, however, has much more than the bare necessities in its 12K of ROM space. The naked Apple is a cassette based system in which BASIC in ROM and a system monitor in ROM prevent unnecessary user aging while waiting for the computer to become operational at turn on. BASIC in ROM was an important development in the popularizing of cassette based personal computers.

*Two 8-bit RAM locations are required to store a 16-bit 6502 address. The 6502 fetches a 16-bit address from an adjacent pair of memory locations. The less significant byte of the address is fetched from the lower memory location, and the more significant byte is fetched from the higher memory location.

ADDRESS DECODING

Inside RAM and ROM, some pretty sophisticated address decoding goes on so that data communication is with the correct memory location. Each RAM chip in the Apple II has a capacity of 16384 individually accessible bits of information. Needless to say, much of the circuitry in the memory chips is devoted to decoding the address input.

Like memory, but on a much smaller scale, the Apple motherboard must decode addresses to control its various functions. As has been stated previously, the address bus and R/W' line are the way in which the 6502 commands the Apple devices to do things. There is a group of logic circuits on the motherboard which detects certain addresses or address ranges, then outputs control signals to various functional areas of the Apple. The following types of control are performed by **address decode**:

1. Gating (enabling) of information to the data bus, including data from serial inputs.*
2. Direct control of serial output lines.
3. Control of peripheral slots.
4. Video mode control.

Control by address decode gives cohesion to the bus structure.

The address and control functions of the address bus are not separate entities but different ways of looking at the same thing. Addressing memory location $95FF can be thought of as controlling that memory location. Similarly, control of the cassette output line may be thought of as addressing it. The **address bus** could be called the **control bus**.

*When a digital signal controls the passage of information in a logic device, it is said to **gate** that information. Gating of information is like opening or closing the gate of a fence to control passage through the gateway.

LOADING AND SAVING MEMORY DATA

Temporary programs and working data in a microcomputer reside in RAM while being used. An external storage device is required to keep this digital information while not in use. The two storage media in common use with the Apple II are 5 1/4" floppy disks and audio cassettes.

Comparing the two, floppy disks are faster, more versatile, easier to use, and more expensive. There are a few applications for which cassette may be better suited, but generally, the disk system is far more usable. The cassette interface is built into the motherboard, but disk drives plug into a peripheral slot (usually two drives per slot maximum).

Figure 2.4 is the partial diagram of the Apple's bus structure, expanded from Figure 2.2 to show cassette and disk input/output operations. The thick red and green lines represent the multiple lines of the address bus and the data bus. R/W' is considered to be distributed with the address bus. The MPU, as before, is in control of the address bus.

Data transfer between RAM and the cassette and disk devices is through the MPU. Data is loaded from the transfer source into the MPU, then stored at the transfer destination from the MPU. Disk I/O data and cassette input data are transferred via the data bus, but cassette output data is not.

Cassette I/O

As shown by Figure 2.4, the cassette input is connected to D7 of the data bus* via the **serial input multiplexor**. Multiplexing is the sharing of one line by more than one signal. All the motherboard serial inputs will soon be seen to be switched on to one data line, D7, by the serial input multiplexor. For now, suffice it to say that $C06X on the address bus is decoded to gate any of the serial inputs to D7, and $C060 in particular is used to select the cassette input.** This allows the MPU to read the cassette input like memory. The cassette input mechanization is similar to ROM. A device responds to its address on the address bus by placing data on the data bus. In this case, however, data is placed on only one line of the data bus. The MPU receives data from the data bus as it does when reading data from memory, and the controlling program ignores everything but D7. The program processes the D7 information, extracts the transfer data, and stores it in RAM.

The fact that the cassette output data is not transferred via the data bus is a surprise to many people. Most of us would expect a serial output to be written out on one of the lines of the data bus as if we were writing to memory. But addressing the cassette output port merely toggles the output line, meaning it changes the high/low state of the output line to the opposite state. In other words, the programmer does not write data to the cassette by telling the output line to go high or low. Rather, he either tells the line to change states or refrains from telling the line to change states at a timed interval. Any address bus state in the $C02X range can be used by programs to toggle the cassette output line, but the programming convention is to use $C020. A point to remember: a serial output can be controlled by the address bus, and the data bus doesn't have to be involved.

Reading or writing to the cassette output port is a control access as opposed to a data access. The MPU reads from the data bus or writes to it on every 6502 cycle, even in a control access. The programmer performs a control access with a normal read or write instruction, but the data that is read and written is irrelevant and ignored. This is why statements like "SPEAKER=PEEK(-16336)" are made in BASIC to control the speaker and the data is ignored. The programmer is making a control access to -16336 ($C030, the speaker port), and the data is irrelevant.

Disk I/O

The disk controller, which resides in a peripheral slot, responds to the address bus/data bus environment much like RAM. During disk input the controller responds to a read access from the MPU by placing a byte of data on the data bus. During disk output the controller responds to a write access by accepting a byte of data from the data bus. The addresses of the input port and output port depend on which slot the disk controller is in. If, as is normally the case, the disk controller is in slot 6, the input port address is $C0EC and the output port address is $C0ED. Besides $C0EC and $C0ED, other address commands perform the functions of motor control, drive selection, read/write configuration, and head positioning. These commands are decoded on the motherboard and controller. The motherboard circuits detect the $C0EX range on the address bus and activate a signal that tells Slot 6

*In *Understanding the Apple II* the lines of the data bus are referred to as D0 through D7, and the lines of the address bus are referred to as A0 through A15. D7 is the line which carries the most significant bit of data on the data bus. A15 is the line which carries the most significant bit of the address on the address bus.

**The $C06X notation is used to indicate the $C060-$C06F address range.

it is being accessed. The controller decodes A0 through A3 of the address bus to determine which of 16 possible commands it is being given.

The actual programming of disk I/O is very complex, requiring timed intervals, data encoding, and extensive software housekeeping. Regardless of this, all MPU control of the disk is via 16 address commands on the address bus, and all data transfer is over the data bus.

There is no motherboard ROM routine to load programs from a disk drive when the Apple is first turned on. A 256 byte program does exist on the controller card, accessible at addresses $C600-$C6FF (assuming slot 6), which loads the extensive Disk Operating System (DOS) from disk to RAM. After power up, the motherboard firmware turns control over to this controller firmware to get the DOS up and running.

THE SECONDARY BUSES

Up to this point only the address bus and data bus have been discussed, and the bus diagrams have shown functional areas connected to these two buses. In actuality, both the RAM chips and the MPU are connected to the address bus and data bus through isolating devices. This creates secondary buses which carry address (control) information and data, but which are not connected directly to the two primary buses. RAM circuitry, in particular, is much more complex than has been represented. The addressing and data flow shown in Figures 2.2 and 2.4 is accurate, but substantial processing details of the address input and data output of RAM were left out so that the essential points of MPU access to RAM could be clearly made. We are now at a point in our discussion where these complexities should no longer be ignored.

Figure 2.5 is the Apple's bus structure, expanded from Figure 2.4 to show the video scanner, video generator, and full I/O capability. It can be seen that the secondary buses are not major distribution networks like the address bus and data bus. They are relatively minor connecting paths for address and data information that must be distinguished from the address bus and data bus. Connections to MPU address and data, keyboard data, RAM output data, and latched RAM output data are basically extensions of the address bus and data bus. The video scanner output and multiplexed RAM address bus are more their own entities. The scanner output is addressing information unrelated to the address bus, and the multiplexed RAM address bus alternately contains information from the address bus and scanner output.

DMA and the MPU

The MPU address, R/W', and data connections are connected to the address and data bus via a 17 bit tri-state line driver and an eight bit line transceiver. The main purpose for these devices is to enable the CPU to **drive** (supply required signal voltages to) all the circuits on two buses, including a possible variety of peripheral cards. A second purpose of the address driver is to give the MPU a tri-state connection to the address bus. This is necessary to isolate the MPU from the address bus during **DMA** operation, because the 6502 address and R/W' outputs are not tri-state. DMA (**Direct Memory Access**) is achieved from a peripheral card when the card pulls the DMA' line low. This DMA capability is actually a **direct bus access** which gives the peripheral card command of the entire Apple. Pulling the DMA' line low forces the 17 bit line driver to high impedance, stops the clock to the MPU, forces the MPU data terminals to input mode, and forces the eight bit transceiver to input from the data bus. Unless it is stated otherwise, our following discussions assume that no peripheral card is performing DMA. This means that the normal situation exists in which the MPU controls the data bus during write cycles and always controls the address bus.

RAM Address Multiplexing and Data Distribution

Besides the MPU address and data connections, the other secondary buses shown in Figure 2.5 are concentrated around RAM. This complexity surrounding RAM is a result of two factors: the transparent video scan of memory and the natural complexity of addressing 16K dynamic RAM.

We don't want to get too steeped in RAM addressing right now, but the basic situation is that there are not enough pins on a 16K dynamic RAM chip to address 16K memory cells simultaneously.* The RAM is addressed with a one-two punch. First, half of the address information is input to RAM where it is saved. Then the second half of the address information is input and the data access takes place. Both

*Throughout this book the word "cell" will be used to refer to a unit of memory that stores one bit of data. The word "location" will be used to refer to eight associated memory cells that hold one byte of data in the Apple II.

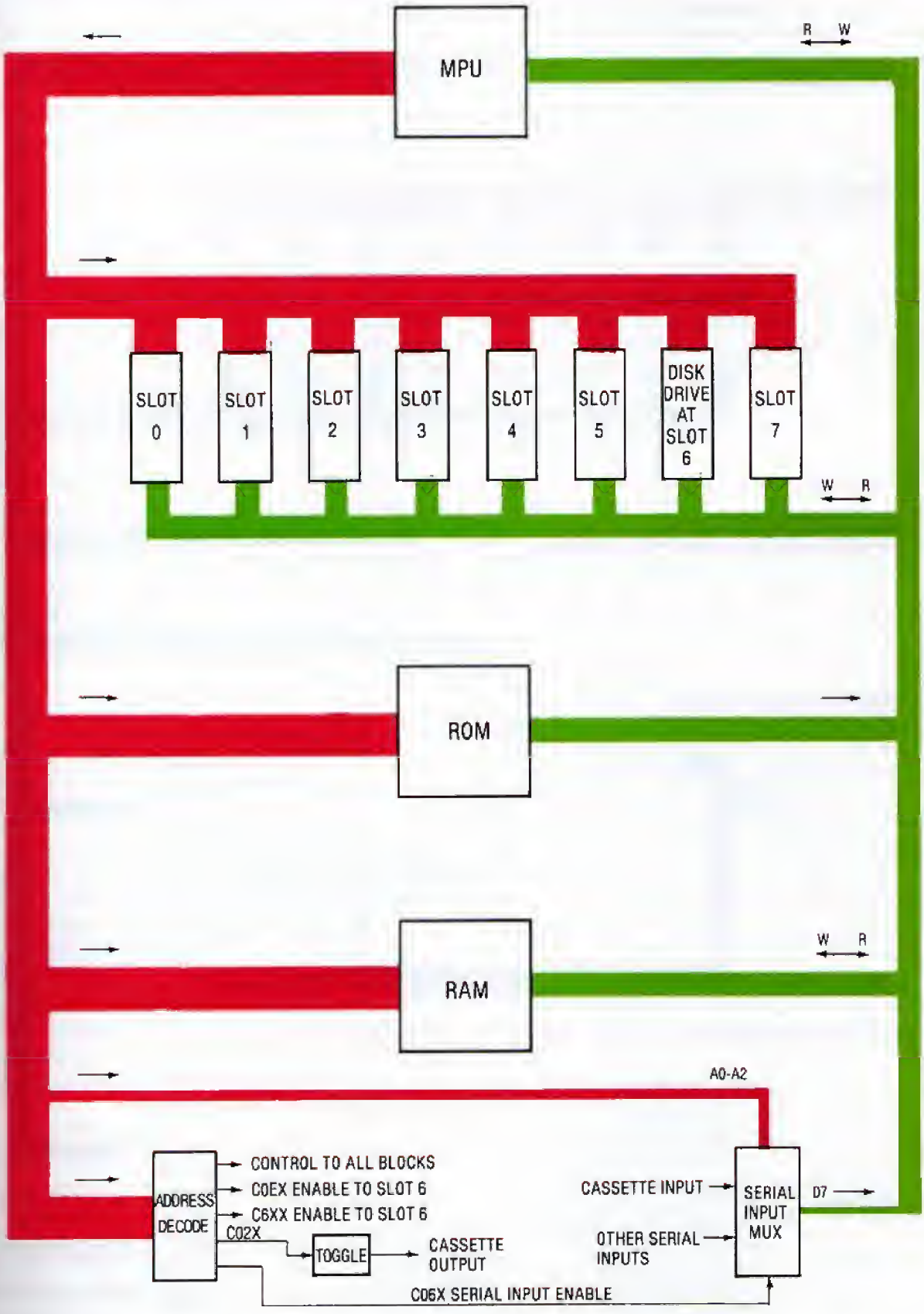


Figure 2.4 The Apple II Bus Structure Showing Disk, Cassette, and Serial I/O.

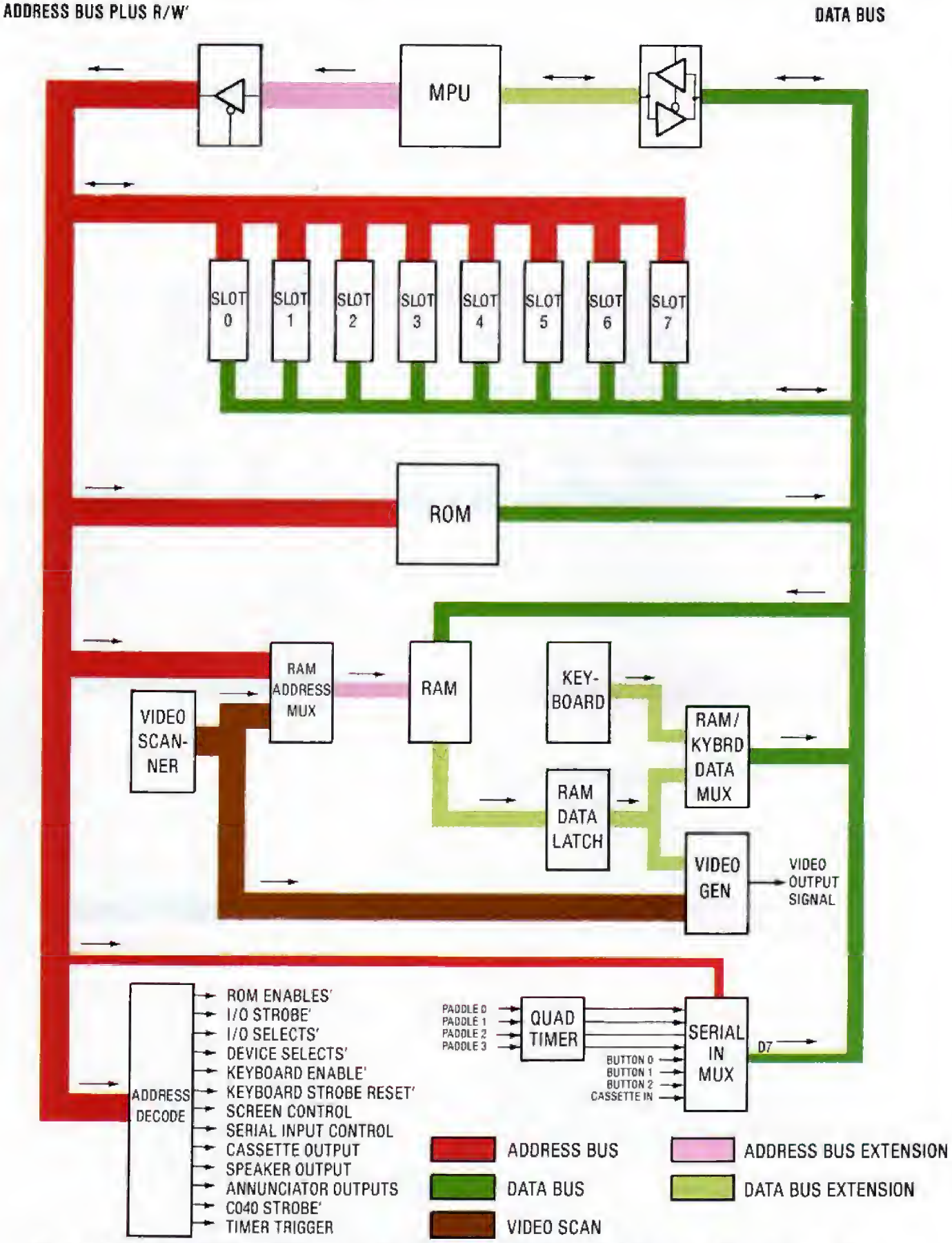


Figure 2.5 Secondary Address and Data Connections Extend the Address Bus and Data Bus.

the first half and the second half of the address are input on the same seven pins of RAM, so fourteen bits of information from the address bus must be multiplexed onto seven lines to effect the one-two punch. This multiplexing is accomplished in the **RAM address multiplexor** and the seven line output of the multiplexor is the multiplexed RAM address bus, shown in pink in Figure 2.5. Incidentally, the other two bits of information from the 16 bit address bus are used to select one of three possible rows of RAM chips.

The two halves of dynamic RAM addressing are referred to as the ROW address and the COLUMN address. This refers to conceptual rows and columns of memory cells inside the RAM chips, not the rows and columns of RAM chips on the motherboard.

The RAM addressing would be complex enough, but in the Apple, the RAM address lines are doubly multiplexed. Both the video scanner and the MPU must access RAM, so during every 6502 cycle, first the video scanner output, then the address bus must be switched on to the multiplexed RAM address bus. Each access is accomplished in two halves (the one-two punch). The RAM address multiplexing is cyclical, resulting in the following repeating pattern of access to the multiplexed address bus:

T1 - Video ROW address
T2 - Video COLUMN address
T3 - MPU ROW address
T4 - MPU COLUMN address

The data bus is connected to the RAM data inputs for writing to RAM, but the RAM output is routed through an eight bit latch and a tri-state data multiplexor to the data bus for reading by the MPU. The RAM data is latched (locked—held constant) twice every 6502 cycle, once from the scanner access and once from the MPU access. Latched data is routed to the video generator for video processing and to the data bus through the RAM/keyboard data multiplexor.

Keyboard Input

During a 6502 write cycle, the RAM data latch must be isolated from the data bus so the MPU can bring the lines of the data bus high or low without interference. The keyboard input, like the RAM latch, is not tri-state and must be isolated from the data bus when it is not being accessed. The dual function of multiplexing and isolation is performed by the RAM/keyboard data multiplexor which is a tri-state device. This multiplexor is switched off during 6502 write cycles and when neither RAM nor the keyboard is being accessed. The rest of the devices on the data bus—ROM, the serial input multiplexor, and the MPU transceiver—all have three state outputs.

The keyboard input, like the latched data from RAM, is eight bit parallel latched information. The keyboard responds to its address like ROM, by placing data on the data bus, so the latched input can be read any time just like memory. The MSB (Most Significant Bit—K7 of K0-K7) is the state of the keyboard **strobe flip-flop** that tells the MPU when any key has been pressed. The keyboard read address is $C00X, and the strobe flip-flop reset address is $C01X.

Video Scanning

The video scanner is not connected to the address bus and is therefore not controllable by the MPU. The scanner is completely isolated from program control and shares RAM on an equal footing with the 6502. The scanner is like a second MPU, but much simpler than an actual MPU. In microcomputer jargon, it is a built-in DMA device performing simultaneous direct memory access with the MPU.

Even though it is a read access as opposed to a write access, video scanner access to RAM is of a different nature than MPU read access. The MPU reads data from RAM, meaning that the MPU addresses RAM, and data from RAM comes back to the MPU. In contrast, when the video scanner addresses RAM, the data from RAM does not come back to the scanner. The data goes out, instead, to the video generator for video processing and to the data bus, when R/W' is high, for processing by peripheral cards. As a result, this book does not refer to the video scanner as **reading** data from RAM . Instead, the video scanner is said to **drive** data out of RAM to the video generator and the data bus.

Other than the fact that the video scanner and MPU both address RAM, their only operational tie is timing. Just as the 6502 executes a machine cycle once every microsecond, the video scanner changes its memory address and accesses RAM once every microsecond. Logically enough, the timing for the video scanner and MPU originate from the same source. In fact, all timing on the motherboard originates at the same source. The timing involved in the sharing of RAM is quite elaborate and is covered in the chapters on timing generation and RAM (Chapter 3 and Chapter 5).

The output of the video scanner is routed to the video generator, as well as the RAM address multiplexor. In the video generator, scanner data is used to make up television sync signals and to locate which part of a text letter or LORES block is being scanned.

Serial I/O and Address Decode

Figure 2.5 shows a full summary of the control functions decoded from the address bus of the Apple II. Most of the control functions are concerned directly or indirectly with I/O features.

It can be seen that the other serial I/O is implemented very much like the cassette I/O. Speaker, annunciator, and C040 STROBE' output lines are controlled directly by address decode in a process which ignores the data bus. The speaker is a toggle output like the cassette output. The programmer can toggle the high/low state, but he never knows whether the state is high or low. The annunciators are on/off outputs which can be brought high or low. For example, $C058 makes ANNUNCIATOR 0 go low, and $C059 makes ANNUNCIATOR 0 go high. The C040 STROBE' simply goes low for .5 microseconds any time $C040 is on the address bus, then returns high.

The serial inputs are all input through the serial input multiplexor to line D7 of the data bus. Any address in the $C06X range gates the serial input multiplexor to the data bus. A0, A1, and A2 from the address bus are connected directly to the serial input multiplexor and select from four timer inputs (paddles), three TTL inputs (pushbuttons) and the cassette input.

Other address decoded signals gate the keyboard and ROM to the data bus and control the peripheral slots and screen modes. Screen mode control is implemented by translating the video scanner output to appropriate RAM addressing in the RAM address multiplexor and by processing RAM data output as text, LORES, or HIRES in the video generator.

Some readers may have noticed that there is no RAM enable signal coming from the address decode block of Figure 2.5. There has to be a RAM enable signal which tells the RAM/keyboard data multiplexor when to enable RAM data to the data bus. There is a RAM enable signal, all right, called **RAM SELECT'**. It is not decoded directly from the address bus, however. RAM SELECT' is decoded in the RAM address multiplexor from the state of the address bus and R/W' during MPU access to RAM, and it is always active during scanner access to RAM when R/W' is high. As a result, data resulting from the scanner access to RAM is sometimes present on the data bus. This interesting feature is further discussed in Chapter 5, but for now, it is much more important to note that when the MPU performs a read access to RAM, the RAM SELECT' signal is activated to gate the output of the RAM/keyboard data multiplexor to the data bus. Thus, regardless of direct memory access by the video scanner, the basic response of memory to a read at its address range is intact. The MPU puts the address on the address bus, and memory responds with data on the data bus.

THE COMPLETED BUS STRUCTURE

This chapter has presented a series of diagrams of the bus structure, building in complexity and completeness as we progressed from basic ideas to detailed structure. The final diagram in this series is a foldout at the back of the book entitled "The Bus Structure of the Apple II." This diagram adds little to Figure 2.5 by way of explaining Apple operation, but it does show some final details about the address bus and data bus connections to ROM and RAM.

ROM is seen to be implemented in six chips, each storing 2048 bytes of firmware. Externally, ROM is much simpler than RAM with straightforward addressing from the address bus and tri-state outputs to the data bus. The 2048 bytes of each chip are addressed by the low order eleven bits (A0-A10) of the address bus. The eight bit output of each chip is gated to the data bus by a signal from address decode which signifies that an address in that ROM chip's address range was detected on the address bus. The six ROM chips are normally referred to by their base address, thus we have the D0 (for $D000), D8, E0, E8, F0, and F8 ROMs.

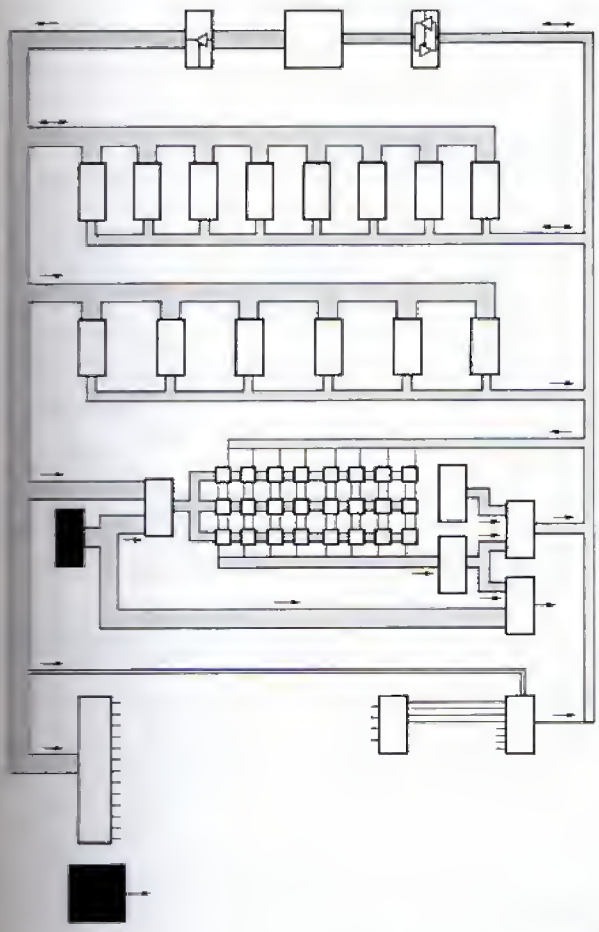
Each RAM chip is organized 16Kx1. This means that each RAM chip has 16,384 1-bit memory cells, one data input line, and one data output line. 6502 microprocessor structure requires that memory be organized for eight bit parallel data transfer, so eight chips provide 16,384 8-bit memory locations in a 6502 system.

To make up the 49,152 bytes of Apple II RAM, 24 chips are laid out on the motherboard in three rows of eight chips. Each row is the equivalent of a 16 kilobyte RAM device with eight input lines and eight output lines, and only one row is enabled during a RAM access depending on the address range of the access. Row C is $0000-$3FFF, row D is $4000-$7FFF, and row F is $8000-$BFFF.

Within each row, each chip is associated with a separate line of the data bus. For example, in row C, chip C10 is associated with line 7 of the data bus.

Chips D10 and E10 are also associated with line 7. Line 7 of the data bus is connected directly to the data input of chips C10, D10, and E10. The tri-state outputs of C10, D10, and E10 are tied together and routed to the bit 7 input of the RAM data latch. The latched bit 7 is routed to line 7 of the data bus when the MPU reads RAM and at other times detailed in the chapter on RAM. Distribution of the other lines of RAM data is identical to that of line 7.

This completes the discussion of the bus structure of the Apple. The author feels that study of the bus structure diagram at the back of the book is very important in the effort to understand the Apple II computer. It is hoped that the reader can become comfortable with the concepts of information flow within the bus structure, because this chapter is the foundation upon which all that follows is built. The remaining chapters are devoted to a more detailed discussion of the various functional areas of the Apple II, beginning with the important subject of timing. Understanding these detailed discussions will be much easier if the reader attempts to visualize how each area performs its functions within the bus structure.



chapter 3

Timing Generation and the Video Scanner

Most operational aspects of the Apple II have now been discussed within the context of the bus structure. However, this discussion has left out one of the Apple II's most important operational aspects—timing. Timing synchronizes everything that goes on in the Apple. To discuss it, we must get into real nuts and bolts detail about computer operation.

Up to this point, the subject matter of *Understanding the Apple II* has been of a general nature. No attempt was made in Chapters 1 and 2 to explain the finer points of Apple II operation. Having gained understanding of the Apple's bus structure, you are largely aware of the methods of communication and control that take place in this computer. The following chapters will build on this foundation of understanding, examining and discussing the detailed features of all functional areas of the Apple II.

The perceptive reader is probably getting the message that the going is about to become stickier. This book attempts to explain as much as possible about the operation of the Apple in understandable English. There comes a point, however, beyond which clear illustration is achieved only with such technical tools as timing diagrams, truth tables, logic diagrams and schematic diagrams. One of the goals of *Understanding the Apple II* is to assist those readers who desire to do so to analyze the operation of the Apple II in depth. For this reason, some technically oriented analysis aids are presented in this chapter and succeeding chapters. These technical aids will be accompanied by technical language. Every person reading these words is capable of understanding the technical sections, but some readers may not wish to, and others will find it a struggle. Every effort has been made to assist all readers in achieving fullest possible understanding from the least possible effort.

By way of warning, the details of some functional areas are just plain difficult, but most of the areas are pretty painless.* In particular, most of the complexity of the Apple is concentrated in the RAM and RAM address multiplexor circuitry. Some other complicated circuitry, like the internal workings of the MPU, will not be discussed at all. Besides the RAM circuitry, the most difficult topics probably

*Even though it is not part of the motherboard circuitry, disk I/O is the subject of a chapter of *Understanding the Apple II*. Readers intrepid enough to tackle this chapter will find disk I/O to be a complex but interesting area of study.

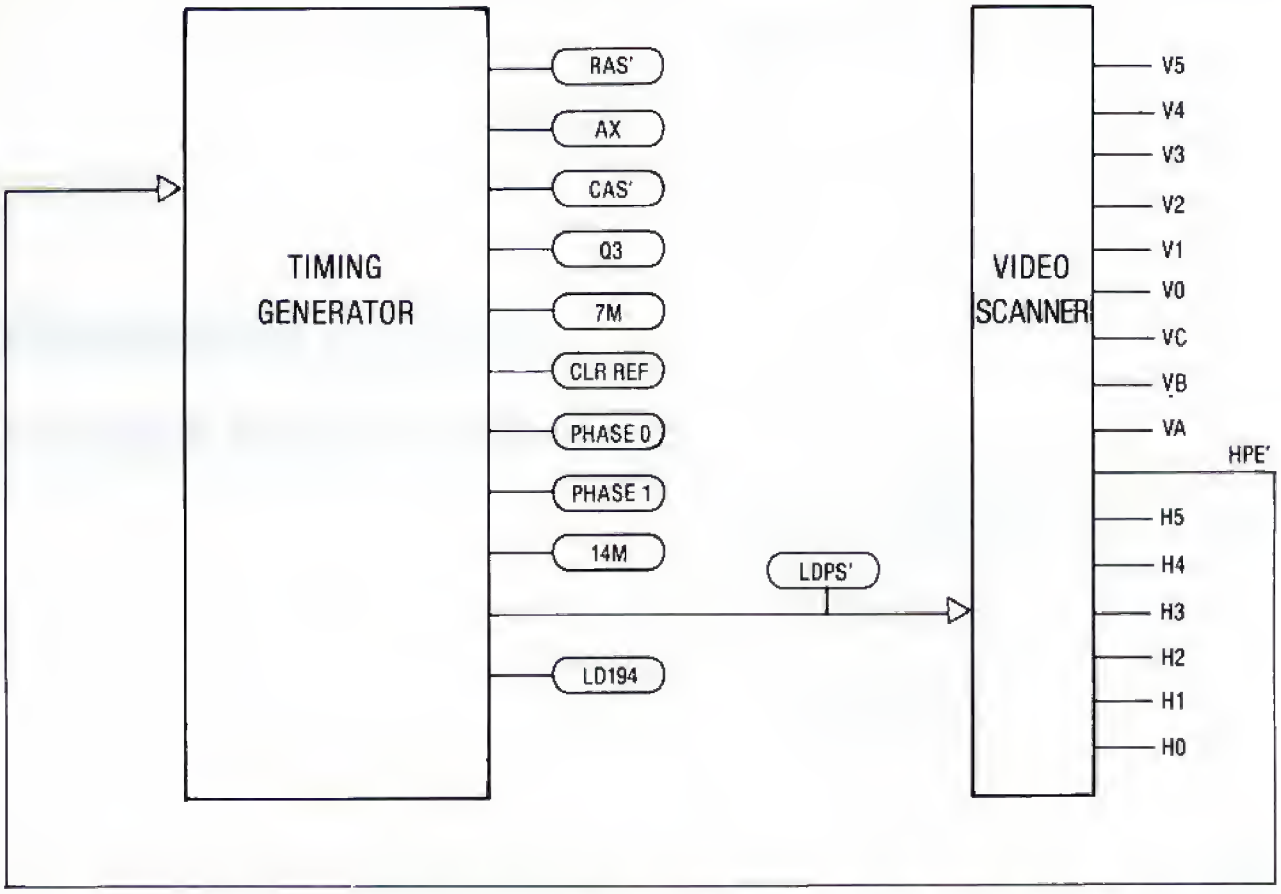


Figure 3.1 Functional Flow: The Timing Generator and the Video Scanner.

are the details of timing and video generation. Timing comes next, so put on your overshoes—we're going wading.

TIMING OVERVIEW

The important timing signals in the Apple II all originate at a small group of circuits called the **timing generator**. You should appreciate this when studying the Apple, because it makes a difficult job easier. Interrelated digital timing originating from multiple sources can scramble your brains. With a single timing source we can assimilate the timing sequences and then apply them to the various functional areas in the following chapters.

Timing signals are distributed to all areas of the Apple, but the Apple's timing requirements are determined primarily by RAM usage. RAM is accessed alternately by the 6502 processor and the video scanner. Executing a stored sequential program and generating a color television video signal are two entirely different tasks, but the two tasks are synchronized in the Apple. As we shall see, execution of this double task dictates certain facts of life about Apple timing.

The timing generator controls the timing and affects all areas of the Apple. By contrast only one external area affects the timing generator. That area is the **video scanner**. The timing feedback from the video scanner elongates one system clock period at the end of each horizontal television scan. This elongation is necessary to keep colors consistent from scan to scan. It also means the clock period of the 6502 is not constant but is elongated on every 65th cycle. This book will refer to this elongated machine cycle as the **long cycle**. Because of the feedback from the video scanner to the timing generator, the two areas are covered in this single chapter.

Apple timing originates with a 14.31818 MHz crystal oscillator. The output of the oscillator, referred to as 14M, is a voltage which switches from low to high and back very close to 14,318,180 times every second. The reason for using 14.31818 MHz instead of 14 MHz is that 14,318,180 Hz divided by four is 3,579,545 Hz, the exact frequency at which color information is passed in a television set. All of the distributed timing signals are clocked by low to high transitions of the 14M clock, so the exact frequencies at which events occur in the Apple are determined by a television signal specification. The

approximate frequencies at which some functions occur are:

FUNCTION	APPROXIMATE FREQUENCY
6502 Cycle	1 MHz
Video Scanner Increment	1 MHz
Address Bus Access	1 MHz
RAM Access	2 MHz
COLOR REFERENCE	3.5 MHz
Video Output	3.5 or 7 MHz

All of these frequencies are determined by outputs of the timing generator.

THE TIMING SIGNALS

This section is a very brief description of the timing signals which are the outputs of the timing generator. All these signals are described in detail later in this chapter.

PHASE 0 is the 1 MHz clock input to the 6502. It also is used to define when an MPU address is valid, and whether the MPU or the video scanner is addressing RAM.

PHASE 1 is PHASE 0 inverted or PHASE 0'.

COLOR REFERENCE is a 3.5 MHz clockpulse which is used to make up the color burst portion of the video output. The color of any Apple video is determined by its phase relationship with the COLOR REFERENCE signal.

7M is a 7 MHz clock used in the shifting of text and HIRES video. It is also available at the peripheral slots.

14M is the output of the Apple's 14 MHz clockpulse oscillator. It is used in timing generation and in the shifting of LORES video. As mentioned in the TIMING OVERVIEW, 14M is the ultimate source of Apple timing.

RAS' (Row Address Strobe) clocks ROW address information to RAM and clocks RAM data output to the data latch. It occurs twice every 6502 cycle, once for MPU access and once for video scanner access.

CAS' (Column Address Strobe) clocks COLUMN address information to RAM. CAS' is used in the Apple to select one of three rows of RAM chips with CAS' being routed to only one row per access. CAS' occurs twice every 6502 cycle.

AX (Address Multiplex) is used in the address multiplexor to select ROW or COLUMN information at the RAM address multiplexor. AX occurs twice every 6502 cycle.

Q3 is a 2 MHz clock used only in timing generation. It is available at the peripheral slots.

LDPS' (LoaD Parallel in/Serial out register) is a video timing term that increments the video scanner and loads text patterns into a register for the purpose of shifting the information serially to the video output. LDPS' occurs once every 6502 cycle.

LD194 (LoaD LS194) is a video timing term which loads graphics data from RAM to two 74LS194 universal shift registers for video processing. LD194 occurs once every normal 6502 cycle and twice during the long cycle.

APPLE FREQUENCIES

It is very hard to make precise statements about the frequencies of some signals in the Apple. This is because of the clockpulse elongation which occurs every 65th 6502 cycle. 14M, 7M, and COLOR REFERENCE are not affected by this elongation. PHASE 0, PHASE 1, Q3, RAS', CAS', and AX are affected.

If not for the long cycle, the frequencies of all timing signals could be computed by dividing 14,318,180 by 14, 7, 4, 2, or 1. In actuality, this works for computing the fixed frequencies. 14M occurs at 14.31818 MHz; 7M occurs at 7.15909 MHz; COLOR REFERENCE occurs at 3.579545 MHz. The 1 MHz and 2 MHz signals are less straightforward.

The period of time required for a 14.31818 MHz signal to go through a complete high/low cycle is 1/14318180 seconds or about 69.8 nanoseconds (69.8 billionths of a second). All synchronized durations in the timing generator are multiples of this time period which we will call the PERIOD for this discussion.

The normal 6502 machine cycle lasts 14 PERIODS or about .978 microseconds. The long cycle lasts 16 PERIODS or about 1.12 microseconds. There are three frequencies involved here: the primary frequency at which the 6502 is operated for 64 out of 65 cycles, 1.0227 MHz; the secondary frequency at which the 6502 operates for 1 out of 65 cycles, .8949 MHz; and the composite frequency which actually is the number of machine cycles per second, 1.0205 MHz.

The 2 MHz signals are similar to PHASE 0 except that only one of every 130 cycles is elongated. Their normal duration is seven PERIODS or about .489 microseconds. Their long duration is nine PERIODS or about .629 microseconds.

The durations and frequencies of the signals of the timing generator are shown in Table 3.1 below. The values are arithmetic derivations of 14.31818, carried to ten place accuracy. Actual frequencies will vary as the 14M oscillator varies from 14,318,180 Hz due to thermal environment and crystal tolerance.

It is reasonable to wonder why the exact frequencies in the Apple should be of any concern. In fact, for most purposes, the exact frequencies are not important. They are important when discussing television compatibility, because television signals require some specific frequencies which are not exact multiples of 1 MHz. Frequency is also important in so far as it affects MPU execution speeds. Knowledge of 6502 clock speed is very important for Apple programs with precision timing loops. For the most part, we will continue to refer to frequencies in very rough estimates such as 1 MHz or 3.5 MHz.

TIMING DIAGRAMS

Timing is usually summarized in **timing diagrams**. Figure 3.2 is a timing diagram showing the outputs of the timing generator and some related signals. The timing diagram is a series of line graphs of voltage as a function of time. Voltage changes vertically in the diagram as time passes from left to right.

In the following discussions of timing signals, the reader is encouraged to refer to Figure 3.2 as necessary to clarify relationships in his own mind. Time periods will be measured in millionths of a second (microseconds) and billionths of a second (nanoseconds).

Figure 3.2 is an idealized timing diagram in which voltages switch instantly and there is no delay between input change and output response (propagation delay). In an idealized timing diagram, cause and effect timing relationships are clearly illustrated instead of being obscured by the details of propagation delay. This is often desirable when studying theory of operation.

Figure 3.2 shows three 6502 machine cycles, two normal cycles, and one long cycle. For each normal machine cycle, there are one PHASE 0 cycle, two RAS', AX, CAS', and Q3 cycles, three and a half COLOR REFERENCE cycles, seven 7M cycles and fourteen 14M cycles. For reference, the period of 14M is about 70 nanoseconds and the period of a normal PHASE 0 cycle is about 978 nanoseconds.

For 64 PHASE 0 cycles, a repeating pattern of timing signals is generated. There is an alternating phase relationship between COLOR REFERENCE and PHASE 0. The H0 signal (see Figure 3.2) is the least significant bit of the video scanner. When H0 is low, COLOR REFERENCE is low when PHASE 0 makes its high to low transition. When H0 is high, COLOR REFERENCE is high when PHASE 0 makes its high to low transition. Other than COLOR REFERENCE the timing signals have identical relation to each other on every normal cycle, and the other H0'/H0 normal cycle pairs are identical to the one pictured in Figure 3.2.

Every 65th cycle, while HPE' is low, generation of the 1 MHz and 2 MHz timing signals is delayed for one half of a COLOR REFERENCE period. This keeps the phase relationship between COLOR REFERENCE and the slower signals correct. This relationship is required in the overall Apple scheme of

Table 3.1 Durations and Frequencies of Timing Signals.

SIGNAL	NORMAL DURATION (nsec)	LONG DURATION (nsec)	PRIMARY FREQUENCY (MHz)	SECONDARY FREQUENCY (MHz)	COMPOSITE FREQUENCY (MHz)
PHASE 0	977.7779019	1117.460459	1.022727143	.89488625	1.02048432
RAS',AX,CAS',Q3	488.888951	628.5715084	2.045454286	1.590908889	2.04096864
COLOR REFERENCE	279.3651148		3.579545		
7M	139.6825574		7.15909		
14M	69.84127871		14.31818		

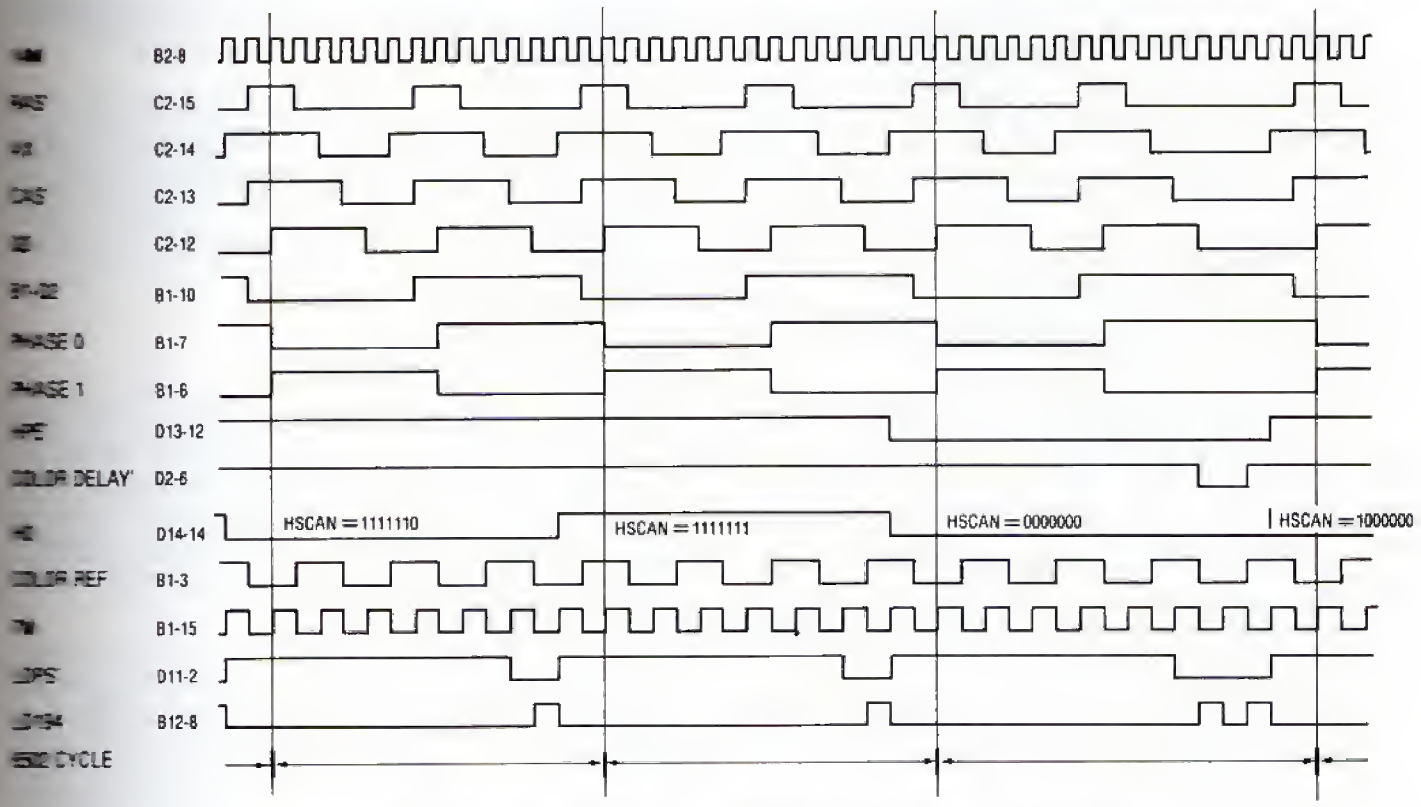


Figure 3.2 Idealized Timing Diagram for the Timing Generator.

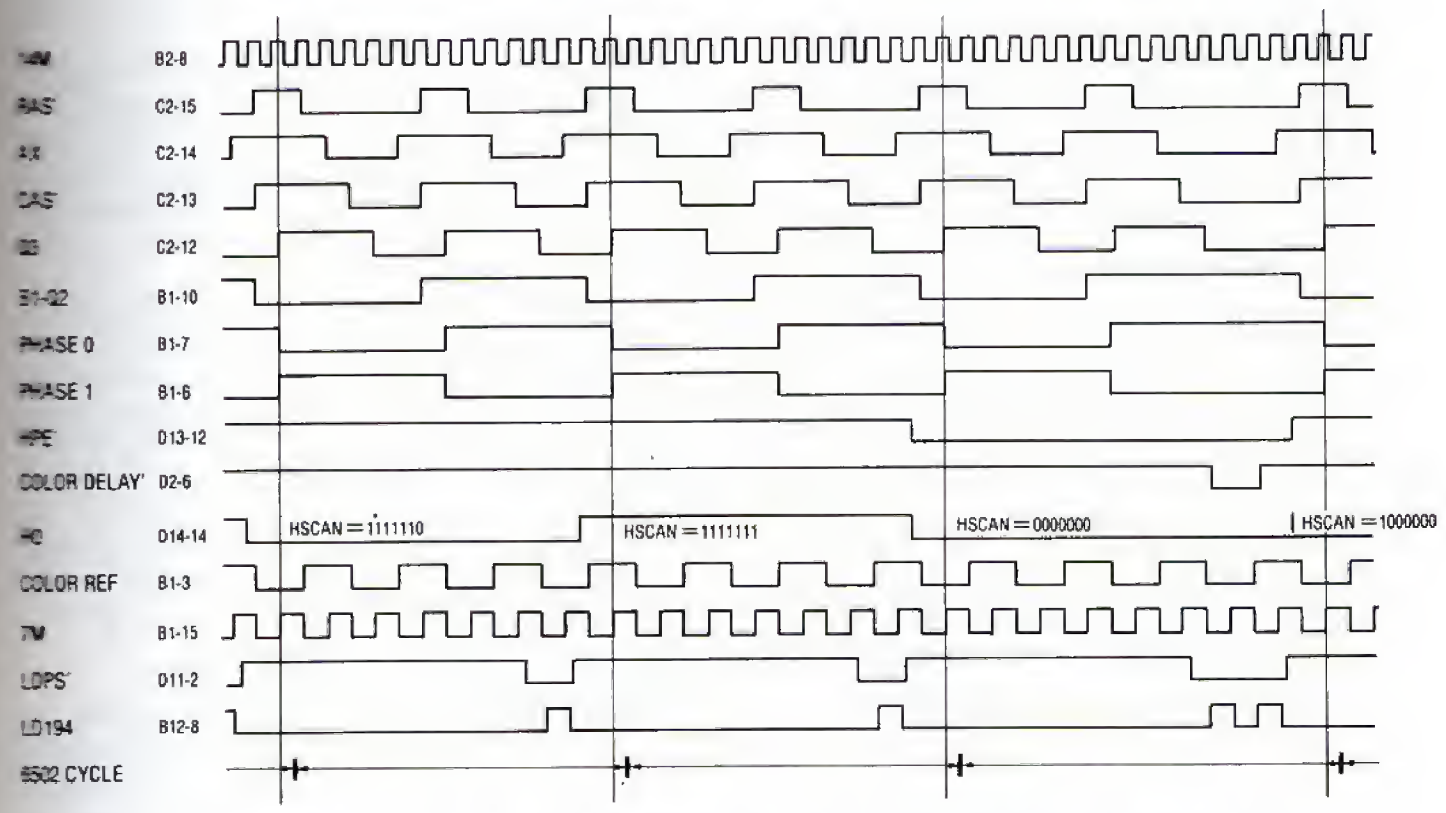


Figure 3.3 Timing Diagram for the Timing Generator, Showing Propagation Delay.

generating color video. The delay extends the high duration of PHASE 0 and extends the low duration of RAS', AX, CAS', Q3, and LDPS'. It also causes a double LD194 pulse and the extension of the current 6502 machine cycle. The logic signal which delays timing generation is labeled COLOR DELAY' in Figure 3.2.

The B1-Q2 signal in Figure 3.2 is a timing signal used in generating PHASE 0. B1-Q2 is not a timing generator output, but is used only internally. PHASE 0 lags B1-Q2 by one 14M period.

Figure 3.3 is an alternate timing diagram showing the same signals as Figure 3.2 but with typical propagation delay illustrated. The cause and effect relationships in Figure 3.3 are not as easily discerned as they are in Figure 3.2, but Figure 3.3 more accurately details the actual timing.*

It is very difficult to illustrate minute propagation delay in a diagram with the time scale of Figure 3.3. Figure 3.4 more accurately depicts the delay hierarchy that exists. The rising edge of 14M is the master reference of Apple timing, and the basic features of propagation delay are:

1. RAS', AX, CAS', Q3, PHASE 0, PHASE 1, 7M, 7M', and COLOR REFERENCE are all clocked by the rising edge of 14M, and their propagation delay from the rising edge of 14M is typically nine nanoseconds.
2. LD194 and COLOR DELAY' are logic functions of other timing signals propagated through a logic device. Their delay from the rising edge of 14M is typically 18 nanoseconds.
3. LDPS' is a logic function of AX and CAS' propagated through two logic devices. Its delay from the rising edge of 14M is typically 27 nanoseconds.
4. The video scanner is clocked by the rising edge of LDPS'. The delay between the rising edge of 14M and HPE' or H0 is typically 42 nanoseconds.
5. PHASE 0 is routed to the 6502 through one logic device. Internal 6502 actions cause a further delay before the 6502 data clock (the falling edge of the 6502 PHASE 2 clock). The typical 6502 internal delay is not specified in data sheets. The delay between PHASE 0 falling at the peripheral slots and PHASE 2 falling at the 6502** was measured by the author at 32 nanoseconds.

*Figures 3.2 and 3.3 show the timing signals switching instantly, and both figures are idealized in this sense. In reality, it takes the timing generator signals about six nanoseconds to rise or fall.

**Synertek SY6502 (marking 78360) in an Apple II computer.

DETAILED DESCRIPTION OF THE TIMING SIGNALS

The following sections describe in detail how the signals of the Apple II are used. The distribution of these signals among the various functional areas of the Apple is shown in Figure 3.5. Please refer to Figures 3.2 and 3.3 as needed while reading these discussions.

PHASE 0 and PHASE 1

PHASE 0 is the 1 MHz clockpulse input to the 6502. As such, its frequency determines the execution time of instructions in the Apple computer. The duration of a PHASE 0 cycle is equal to the duration of a 6502 cycle. This duration is .98 microseconds in a normal cycle and 1.12 microseconds in a long cycle.

PHASE 1 is simply PHASE 0 inverted. It is high when PHASE 0 is low and vice versa. The PHASE 0 cycle period is almost coincident with a 6502 machine cycle but slightly leads it. Speaking of PHASE 1 and PHASE 0 as positive gating signals, PHASE 1 occurs approximately during the first half of the 6502 machine cycle and PHASE 0 occurs approximately during the second half. The time relationships of PHASE 1, PHASE 0, and the 6502 machine cycle are shown in Figure 3.6.

Clockpulse action takes place when the PHASE 0 line switches from high to low or low to high. These transitions trigger actions inside the 6502 which will be discussed in greater detail in the next chapter. A high to low transition of PHASE 0 causes the 6502 to begin a new machine cycle after a short delay.

In addition to triggering 6502 events, PHASE 0 is used as a time reference on the motherboard. During PHASE 0 the 6502 address is valid, so address decoding from the address bus takes place during PHASE 0. RAM is addressed by the MPU during PHASE 0 and by the video scanner during PHASE 1. Also, since scanner access is during PHASE 1, the RAM read/write control is set to "read" during PHASE 1 even if the 6502 R/W' line is set to "write".

14M, 7M, and COLOR REFERENCE

14M, 7M, and COLOR REFERENCE (3.5M) are utility clocks which are used in the generation of video. The frequency of Apple video can be as high as 7 MHz, so generating the video signal requires fast clocks.

14M, 7M, and COLOR REFERENCE are unaffected by the long cycle and have fixed frequencies

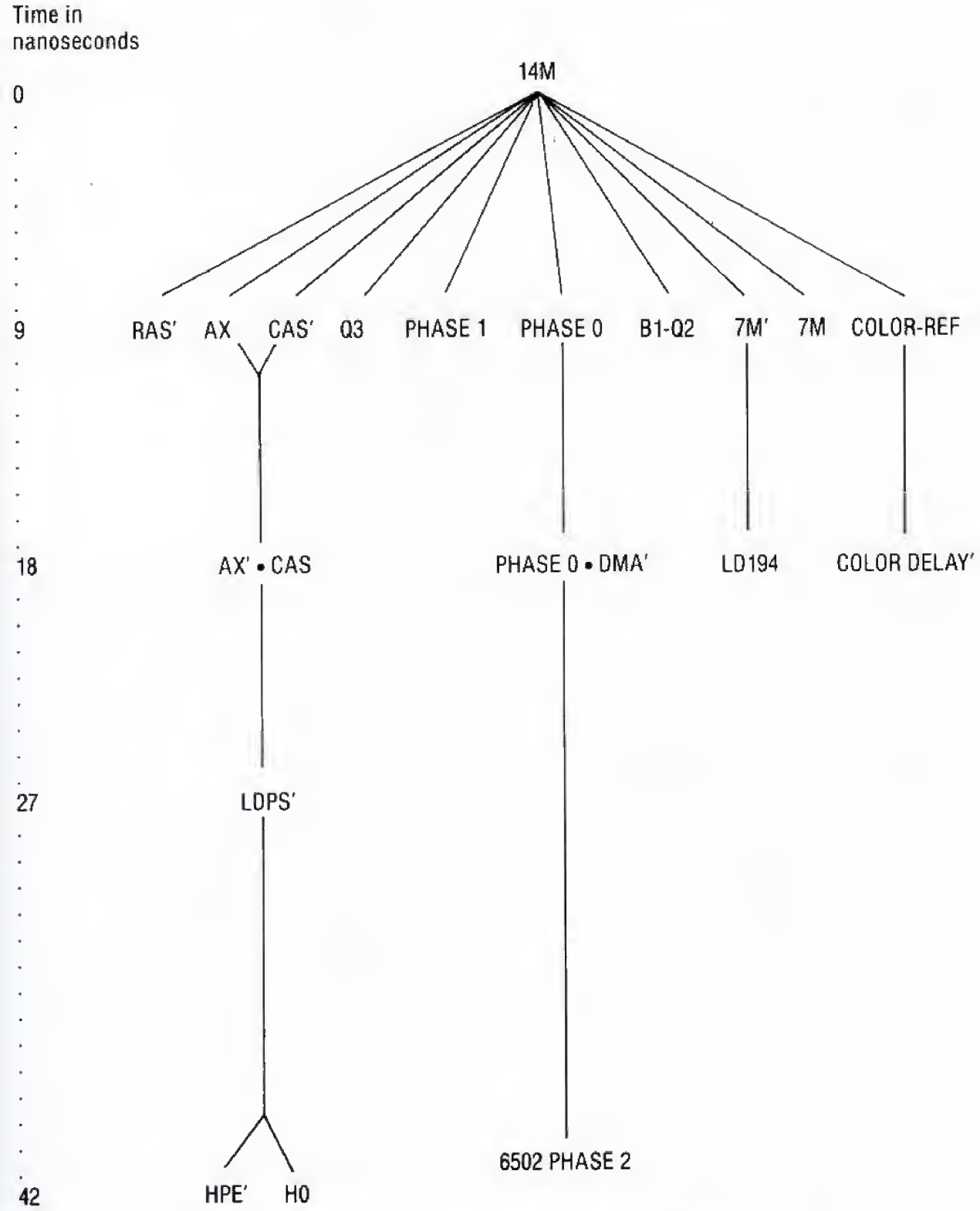


Figure 3.4 Propagation Delay Hierarchy.

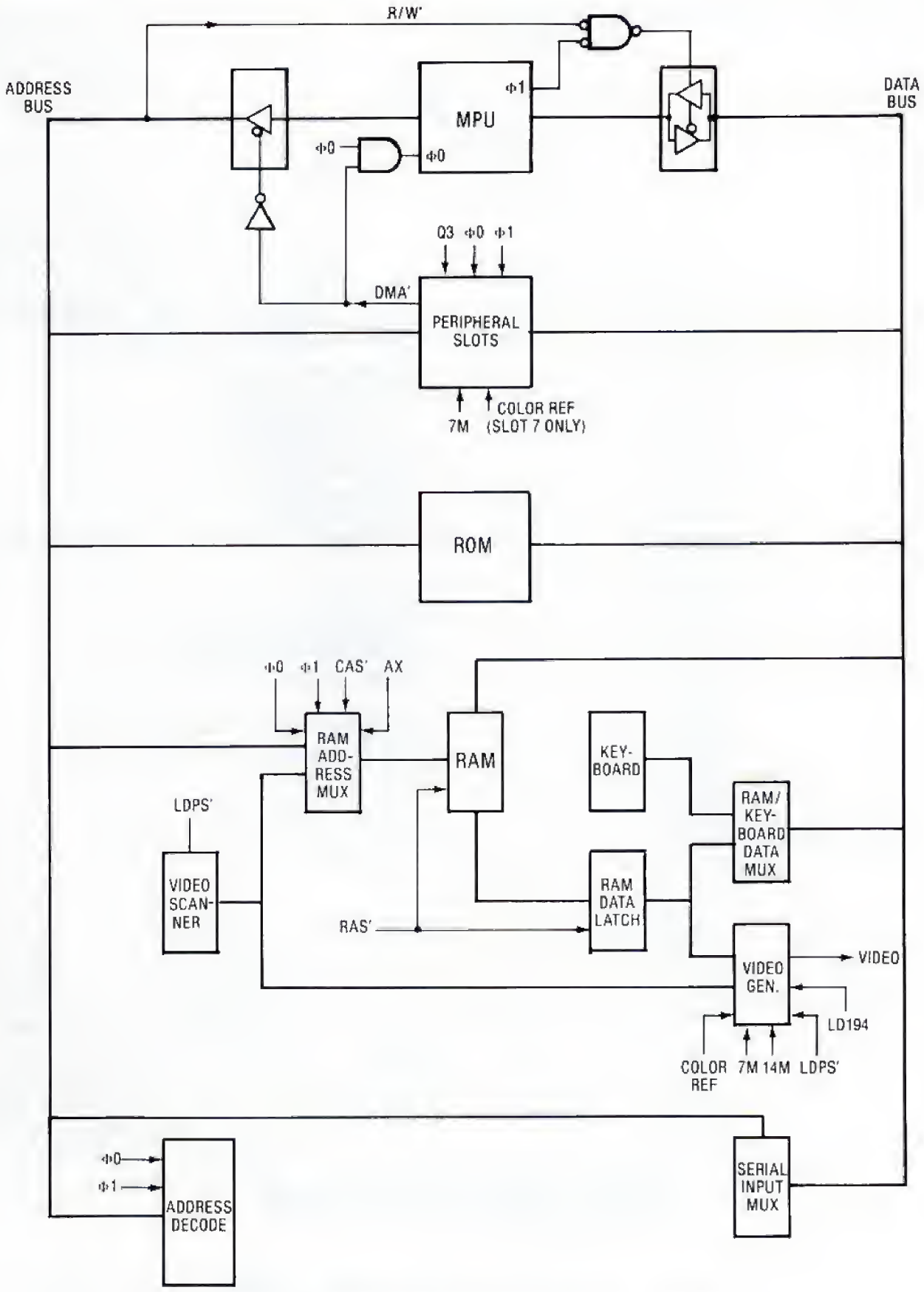


Figure 3.5 Distribution of Timing Generator Outputs.

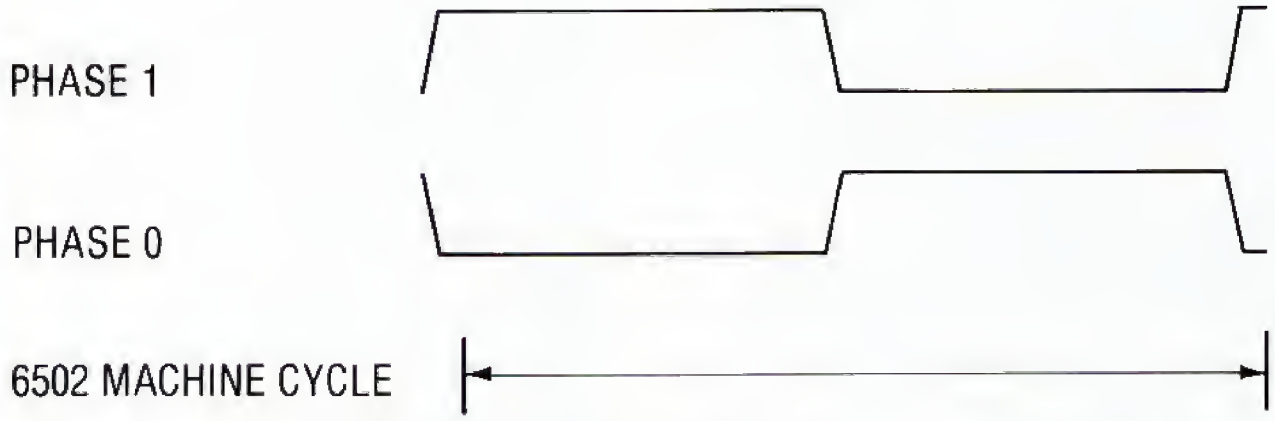


Figure 3.6 The 6502 Machine Cycle Slightly Lags the PHASE 0 Clockpulse.

of 14.318180 MHz, 7.15909 MHz and 3.579545 MHz respectively. 14M and 7M are used strictly as clockpulses in the **video generator**, but COLOR REFERENCE is used differently. Short bursts of the COLOR REFERENCE signal are placed on the video output line once every horizontal scan. A television set is capable of reproducing the continuous COLOR REFERENCE signal from these short bursts, allowing the COLOR REFERENCE input to the television to become the phase reference for color generation. The Apple produces color on a television by shifting the PICTURE signal in relation to the COLOR REFERENCE.*

There is some confusion over the labeling of COLOR REFERENCE because changes were made to the color burst killing circuitry in the RFI revision. The changes were such that COLOR REFERENCE', rather than COLOR REFERENCE, is required in the video generator. Because of this, COLOR REFERENCE' (B1, pin 2) is routed to the video generator on RFI Revision boards, while COLOR REFERENCE (B1, pin 3) is routed to the video generator on older boards. The confusion results from the fact that Apple changed the label of COLOR REFERENCE' to COLOR REFERENCE in the schematic published in an addendum to the *Apple II Reference Manual*. This schematic also shows B1, pin 3 open, but this pin is actually connected to Slot 7, pin 35. In the face of this inconsistency in Apple's documentation, this book sticks to the old labels for COLOR REFERENCE. When COLOR REFERENCE is referred to, it means the Q1' output at B1, pin 3. The Q1 output at B1, pin 2 is referred to as COLOR REFERENCE'.

7M is available at pin 36 of the peripheral slots. COLOR REFERENCE is available at pin 35 of Slot 7 only. 14M is not available at the peripheral slots.

*This book refers to the signal which controls the intensity of the Apple display as the PICTURE signal. When the PICTURE signal is at the white level, the electron beam in the television picture tube strikes the picture screen with enough intensity to cause light emission. The PICTURE signal, SYNC, and COLOR BURST are the three components of the Apple VIDEO signal. More information on this subject is contained in Chapter 8.

RAS', AX, CAS', and Q3

RAS', AX, CAS', and Q3 are 2 MHz signals. Q3 is used in the Apple strictly for the generation of other timing signals. It has status as a timing generator output only because it is distributed to pin 37 of the peripheral slots. RAS', AX, and CAS' are RAM timing signals.

It can be seen from Figure 3.2 that a RAS', AX, CAS' sequence occurs twice every 6502 cycle. The PHASE 1 sequence controls the video scanner access to RAM, and the PHASE 0 sequence controls the MPU access to RAM. The falling edges of RAS' and CAS' strobe the ROW address and COLUMN address to RAM, while AX selects ROW or COLUMN address lines at the RAM address multiplexor. There is a continuing cycle of RAM access:

1. Select ROW address via AX high.
2. Strobe ROW address via RAS' falling.
3. Select COLUMN address via AX low.
4. Strobe COLUMN address via CAS' falling.

RAS' is wired directly to the RAM chips, but CAS' is distributed to RAM through RAM select logic in the RAM address multiplexor. When RAM is not being accessed, such as during PHASE 0 when the MPU is addressing ROM, CAS' is not gated to RAM. When RAM is being accessed, CAS' is gated to RAM row C or D or E on the motherboard depending on the RAM address.

The rising edge of RAS' latches the data output of RAM on the Apple. This means that whatever is on the output lines of RAM when RAS' goes high will be saved at the RAM data output latch until the next time RAS' goes high. The latched output is used by the video generator to make up the video display, and it is gated to the data bus when the MPU is reading RAM. More detail on the timing of MPU access to RAM is given in the chapters on the MPU and RAM.

LDPS' and LD194

LDPS' and LD194 are timing signals used in the generation of video. The signals occur at 1 MHz and provide the time reference for video output.

The rising edge of LDPS' causes the video scanner to increment. Since the video scanner addresses RAM, a different memory location is processed for video output every LDPS'.

The generation of the PICTURE signal is a load/shift process. Data is loaded from RAM or from a dot matrix text ROM addressed by RAM data. Then it is shifted out as the PICTURE signal. LD194 and LDPS' are the load/shift reference for the PICTURE signal generation. While LDPS' is low, dot matrix text patterns are loaded in the video generator. While LDPS' is high, they are shifted out. While LD194 is high, graphics dot patterns are loaded in the video generator. While LD194 is low, they are shifted out.

LD194 is the load/shift control for graphics. LD194 occurs when COLOR REFERENCE is low and H0 is low, or it occurs when COLOR REFERENCE is high and H0 is high. It is to keep this relationship between video timing and the COLOR REFERENCE that the long cycle exists. The elongation of the 6502 machine cycle is an incidental side effect.

TELEVISION SCANNING

To understand the operation of the video scanner, it is necessary to understand a little bit about television operation.* The television display is achieved by scanning an electron beam across the screen. The PICTURE signal level controls the beam intensity and the resulting light intensity as the viewer sees it.

The electron beam scans much faster horizontally than it does vertically, so the scan or **raster** is made up of many nearly horizontal lines as shown in Figure 3.7. The scanning circuitry is internal to the television, but the signal input synchronizes the scanning with horizontal and vertical sync. The horizontal sync causes the beam to return very quickly to the left side of the screen, and the vertical sync causes the beam to return very quickly to the top of the screen. The horizontal and vertical sync must occur approximately at television horizontal and vertical frequencies for the television to become synced. In American television, the horizontal scanning frequency is 15,734 Hz and the vertical scanning frequency is 59.94 Hz.

Horizontal and vertical sync occur while the PICTURE signal is at a black, or **blanking**, level. After the horizontal sync causes the beam to go to the left side, the beam traces left to right while the PICTURE signal controls beam intensity.

The Apple must generate the television signal which is a combination of horizontal sync, vertical sync, picture level, and a color burst. It does this by scanning memory for video output with a counter which has recurring periods approximately equal to the horizontal and vertical periods of a television. This counter is the video scanner.

THE VIDEO SCANNER

The video scanner is a counter that counts like a television scans. The low order bits (H0-H5 plus HPE') form the horizontal section which sequences through its counts one time for every horizontal scan. The entire scanner (H0-H5, HPE', VA-VB, and V0-V5) sequences through its counts once every vertical scan. In the video generator, outputs of the video scanner are used to develop horizontal and vertical sync for the video signal.

Since states of the video scanner synchronize the television scan, the video scanner can be thought of as scanning the TV screen as it scans memory. The electron beam is always in the same spot on the screen when a given memory location is accessed by the scanner.

*Chapter 8 contains a more detailed description of television operation. The important concepts here are television scanning and synchronization.

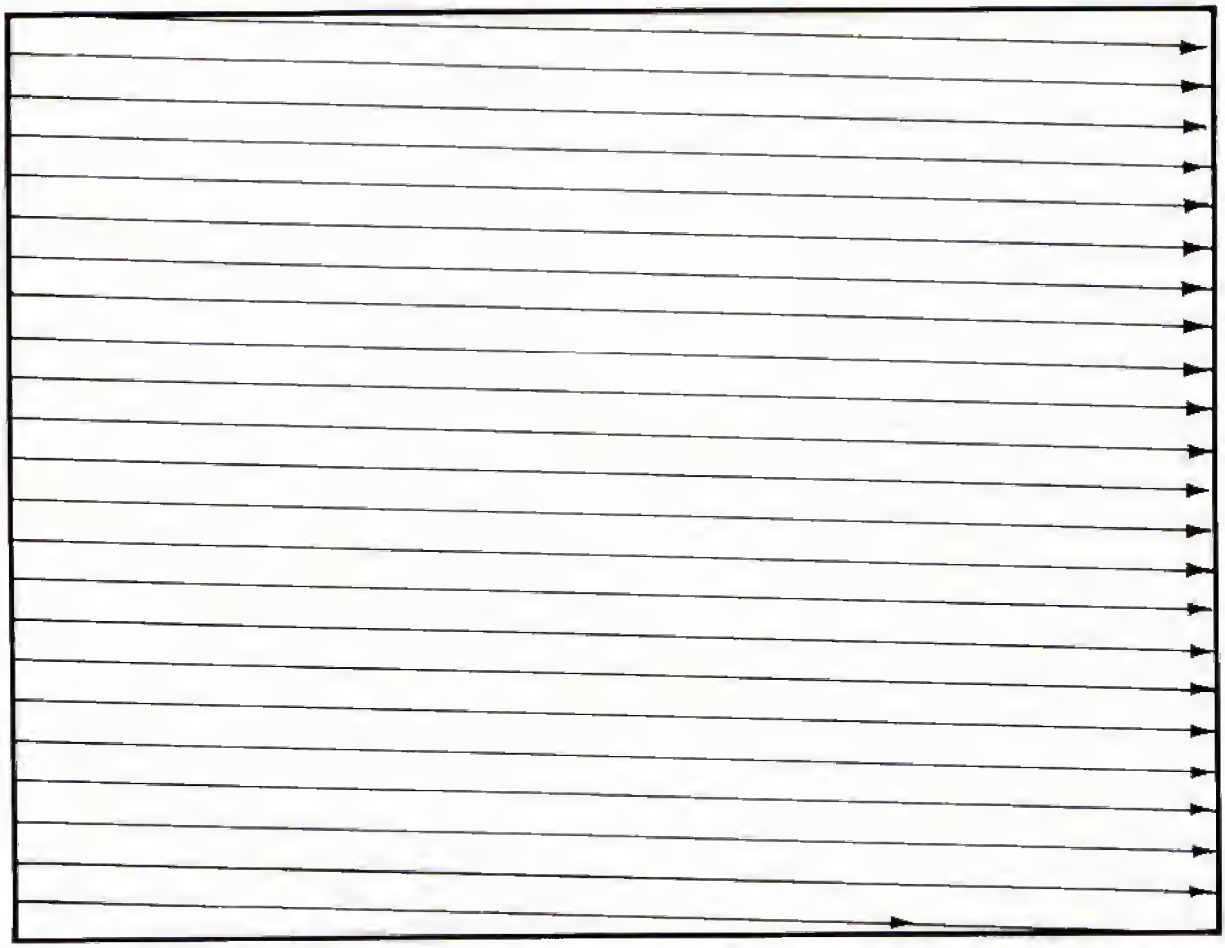


Figure 3.7 Exaggerated View of a Television Scan. The Apple Scans 262 Times Horizontally for Each Vertical Scan.

The video scanner increments on the positive going edge of LDPS', which occurs toward the end of PHASE 0. Just like the MPU, the scanner operates at 1 MHz. There is a one microsecond period for which every state of the scanner is held until the scanner increments to the next state. During one microsecond, the electron beam travels the width of one text character, one LORES block, or seven HIRES dots.

Horizontal Scanning

The video scanner is divided into the horizontal section and the vertical section. The horizontal section is made up of H0-H5 plus HPE' (Horizontal Preset Enable). These seven bits are mechanized as a 65 state counter which counts LDPS'. The 65 states of the horizontal counter are 0000000 and 1000000 through 1111111. HPE' is low only during one of the 65 states (0000000) and the fact that it is low does two things. First, it presets the horizontal section to 1000000. Second, it is fed back to the timing generator to cause the long cycle.

One horizontal SYNC pulse is output from the video generator for every time the horizontal section of the video scanner goes through its 65 state sequence, so the 65 state sequence represents one horizontal scan. During 40 of the states, picture information is output on the video line. During 25 of the states no information is sent to the screen. This blanking period includes the left margin, right margin, and retrace (quick movement of the beam from right to left).

The duration of the horizontal sequence is equal to 64 normal 6502 cycles and one long cycle. This takes 63.695 microseconds, which gives a horizontal frequency of 15,700 Hz. This is very close to the standard television horizontal frequency of 15,734 Hz.

Vertical Scanning

The vertical section of the video scanner is made up of VA-VC and V0-V5. The vertical section increments every time there is an overflow from the horizontal section, meaning it increments when the horizontal count is 1111111 just before HPE' goes low. The vertical section counts horizontal scans.

The nine bits of the vertical section are mechanized as a 262 state counter. The 262 states are 011111010-111111111. It is a straightforward binary counter which presets on overflow to 011111010. Once each vertical sequence, the video generator sends vertical sync, so the 262 state sequence represents a vertical scan. During 192 of the scanner states, picture information is output on the video line. The 70 blanked horizontal lines represent the top margin, the bottom margin, and the retrace to the top of the screen.

There are exactly 17030 (65 x 262) 6502 cycles in every television scan of an American Apple. The duration of the television scan is equal to 262 horizontal scans. This is 16,688 microseconds which gives a vertical frequency of 59.92 Hz. This is very close to the standard American television vertical frequency of 59.94 Hz.

In a standard television picture, alternating vertical scans are interlaced. This means that every other scan is displaced vertically half of the vertical distance between two horizontal scans. Interlacing gives an effective vertical resolution of 525 lines. There is no vertical interlace in the Apple display. This accounts for a disparity in vertical/horizontal frequency relationships between Apple video and broadcast television video. In the Apple, the horizontal frequency is 262 times the vertical frequency. In American broadcast television, the horizontal frequency is 262.5 times the vertical frequency.

Eurapple and the Video Scanner

It is possible to make the Apple II television scanning compatible with European televisions by reconfiguring the **Eurapple jumpers**. These jumpers do not affect the horizontal scanning rate, but they do affect the vertical scanning rate. With Eurapple jumpers, the vertical section presets on overflow to 011001000 instead of 011111010. There are 312 states represented by 011001000-111111111. This gives a vertical frequency of 50.32 Hz. Even though there are 50 extra horizontal scans in the Eurapple, there is no extra vertical resolution. In either Apple system there are 192 horizontal scans in which picture information is displayed.

THE LONG CYCLE

The discussions have alluded to the **long cycle** in a limited way, but we are now in a better position to understand the reasons for it.

Video output begins each horizontal scan when the horizontal count reaches 1011000. For color coherency the video output needs to begin at the same point in relation to COLOR REFERENCE on every scan. Since there are 3.5 COLOR REFERENCE cycles in a video scanner cycle, the phase of COLOR REFERENCE at the start of a video shift alternates 180 degrees each scanner cycle. Because of the 180 degree phase alternation each cycle, a seven dot HIRES pattern represents different colors when it is stored in an even RAM address than when it is stored in an odd RAM address.

There are 65 video scanner cycles per horizontal screen line. This is an odd number so there would be an odd number of 180 degree phase alternations per horizontal line. This would cause the starting phase relationship to alternate every horizontal line. By delaying video shift timing half a color reference period once every horizontal line, the same beginning phase relationship occurs every horizontal line. As a side effect all 1 MHz and 2 MHz signals are elongated once every horizontal line.

TIMING GENERATOR HARDWARE

Timing generation in the Apple consists of making a lot out of a little. The 14M clock is divided and processed to make the slower, more complex signals. Figure 3.8 is an annotated schematic of the timing generator and the primary analysis aid for studying timing generator hardware.

14M, 7M, and COLOR REFERENCE generation is straightforward frequency division. 14M comes from a crystal controlled 14.31818 MHz oscillator via one fourth of a 74S86 quad exclusive-OR gate. 14M is pretty symmetrical but symmetry is not important since only the rising edge of 14M is used. 7M is 14M divided by two. COLOR REFERENCE is 7M divided by two. The exclusive-OR gate at the input of B1-Q1 forces COLOR REFERENCE to toggle at the falling edge of 7M.

RAS', AX, CAS', and Q3 are generated in shift register C2 (74S195). C2 is configured to shift "0"s to RAS', AX, CAS', and Q3 when Q3 is set, then to shift "1"s to AX, RAS'/CAS', and Q3 when Q3 is reset. The continuous four clock/three clock shift makes up a seven clock cycle for each of the 2 MHz signals.

The shift "1" input to AX is the COLOR DELAY' signal. The long cycle is created when COLOR DELAY goes low at:

HPE • PHASE 0 • AX' • CAS • COLOR REF'.

This causes the shift "1" phase of C2 to be delayed by one half a COLOR REFERENCE period.

PHASE 0 is generated in sync with the 2 MHz signals. Generating PHASE 0 from the available 2 MHz signals would not seem complex, but hardware expedience and engineering resourcefulness

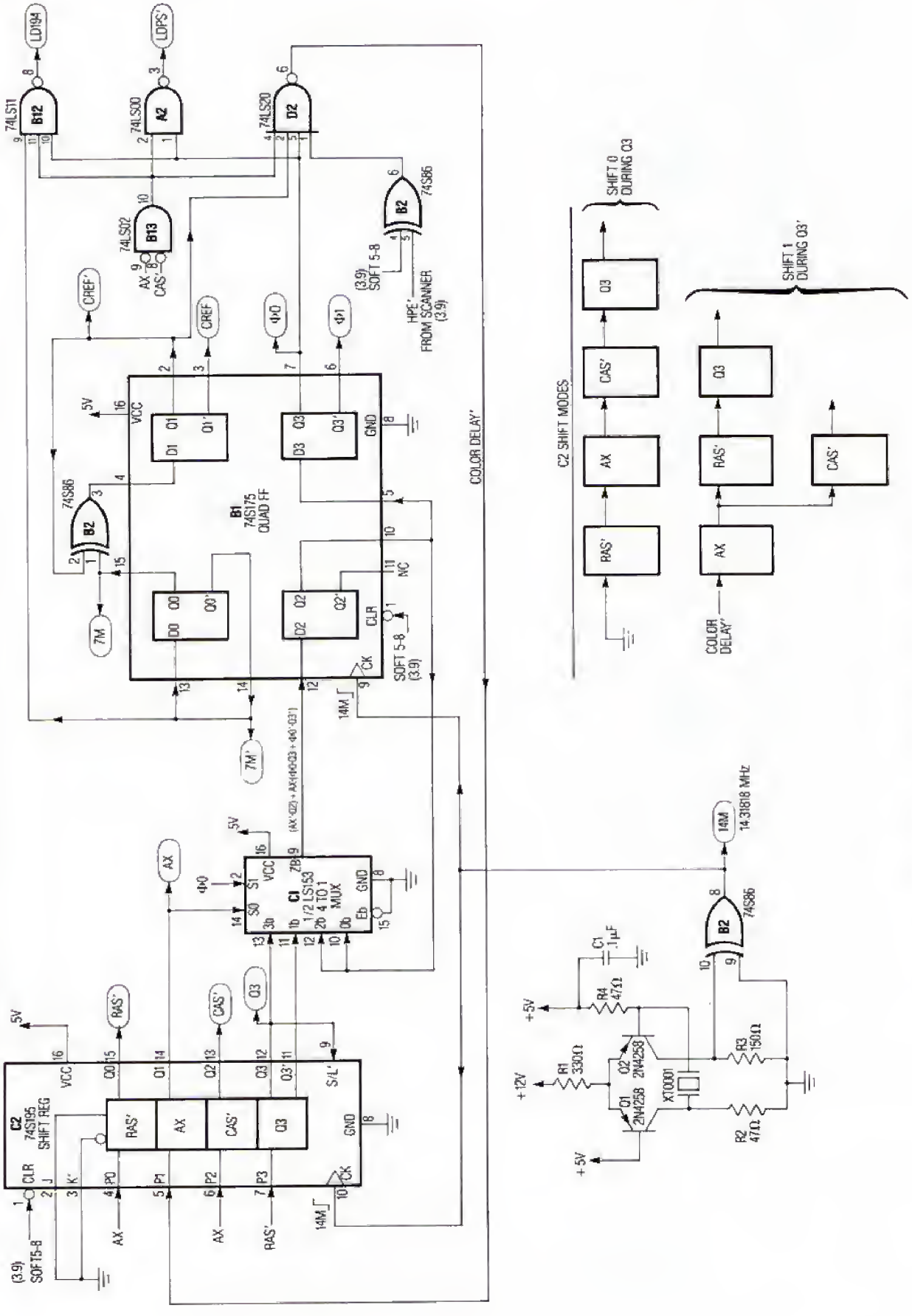


Figure 3.8 Schematic: The Timing Generator.

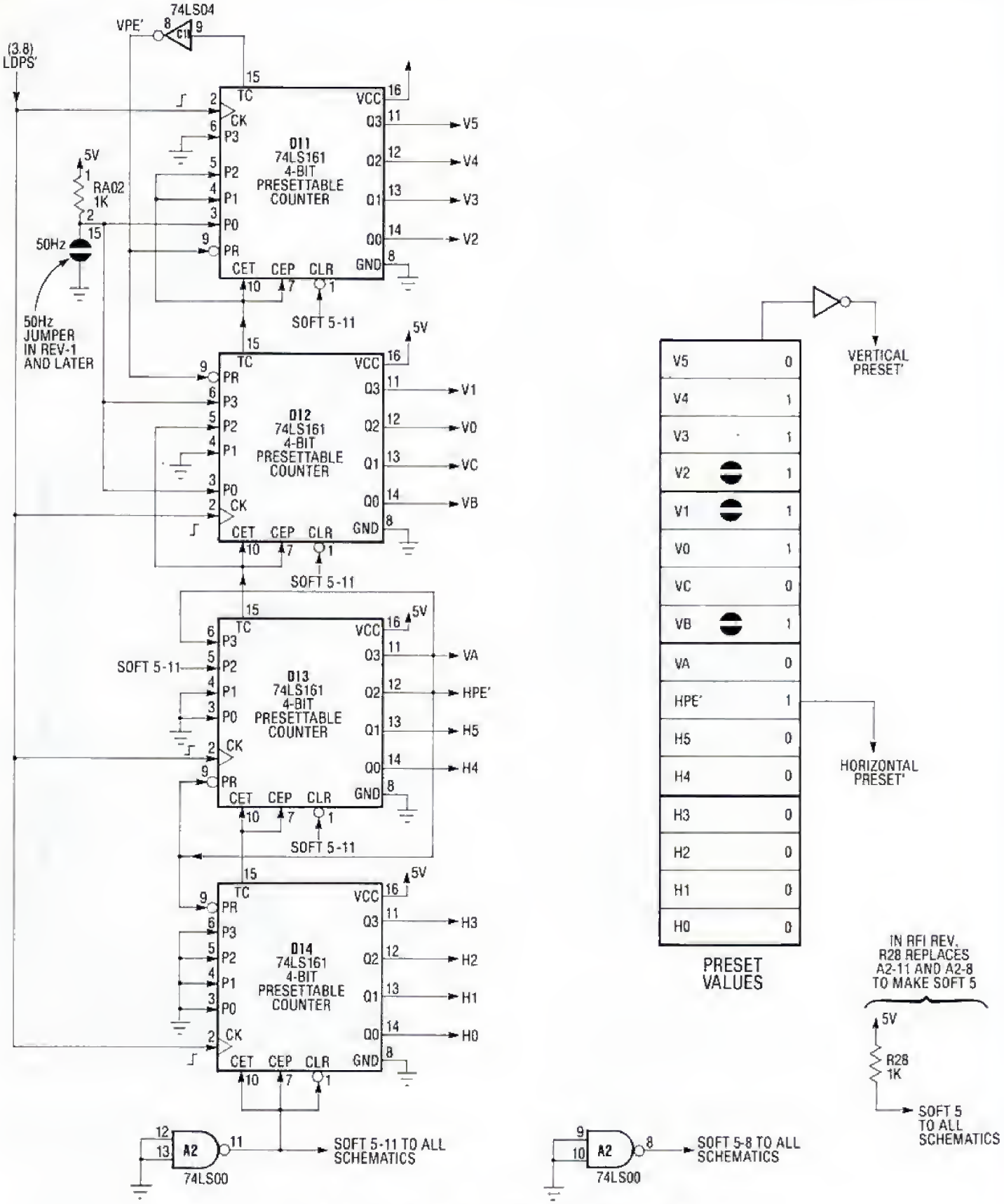


Figure 3.9 Schematic: The Video Scanner.

made the Apple inventor do it in a complex way. PHASE 0 is generated by two flip-flops in B1 and one half of a 4 to 1 multiplexor, C1. All of this generates a PHASE 0 signal which simply toggles every time CAS' is high and Q3 is low. The way it is done in the Apple, C1 develops the logic equation

AX' • Q2 + AX(PHASE 0 • Q3 + Q3' • PHASE 0'),

where Q2 is B1-Q2 and Q3 is the Q3 output of the timing generator. This signal is the D-input to B1-Q2. PHASE 0 follows B1-Q2 by one 14M clock period. Analysis shows that this generates PHASE 0 as pictured in Figures 3.2 and 3.3.

A point worth noticing in the timing generator is that 74S devices are used to generate signals which are widely distributed. Most Apple TTL is mechanized with 74LS devices. 74S devices have more drive capability than 74LS devices, and by the use of 74S devices in the timing generator, the number of 74LS devices which can be driven by timing generator signals is doubled.

VIDEO SCANNER HARDWARE

The video scanner is a television synchronous 16-bit counter implemented on four 74LS161 4-bit counters. The scanner increments on the rising edge of LDPS'. The four chips (see Figure 3.9) are cascaded into a single counter by connecting the carry output of each chip to the count enable inputs of the next chip in the string.

The counter is divided into two separately preset sections. The horizontal section (H0-H5 plus HPE') is a 7-bit modulo 65 counter. The 65 states are 0000000 and 1000000 through 1111111. The low state of bit 7 (HPE') presets the horizontal section to 1000000.

The vertical section (VA-VC and V0-V5) is a 9-bit modulo 262 counter which counts every time the horizontal section equals 1111111. The vertical section presets on scanner overflow to 011111010. Its 262 states are 011111010-111111111. Eurapple jumpers change the vertical preset to 011001000 and change the number of vertical count states to 312.

The vertical section increments on the LDPS' before HPE'. A typical count sequence is
111100000/1111111; 111100001/0000000; 111100001/1000000.
The vertical preset sequence is
111111111/1111111; 011111010/0000000; 011111010/1000000.

VA is located on the same LS161 as H4, H5, and HPE', and it is not controlled by the vertical preset, but by HPE'. However, it always resets at vertical preset time because of its count logic. During HPE', it holds its value. Thus, VA is effectively preset to "0" at vertical preset time.

The video scanner implementation is one of a number of examples of design resourcefulness in the Apple II. The design of a 65 state counter cascaded into a 262 state counter is a fairly simple task, but doing it with four and a quarter TTL chips involves less obvious methods. This makes the circuit harder to understand, but increases the educational value of studying such a circuit.

The SOFT-5 signals shown in Figure 3.9 are not just part of the video scanner. They are constantly high logic levels distributed throughout the motherboard to those logic devices which require them. In earlier Apples, SOFT-5 was developed by two 74LS00 NAND gates with their inputs grounded. In RFI Revision Apples, SOFT-5 is a connection to the +5 volt line through a 1000 ohm resistor.

SOFTWARE APPLICATION

SWITCHING SCREEN MODES IN TIMED LOOPS

A horizontal scan in the Apple takes exactly 65 machine cycles of the 6502. A vertical scan takes exactly 17030 machine cycles. This information can be used to switch screen modes in timed loops to give apparent combination screen modes.

For example, the screen can be split so that half of each horizontal line is LORES and the other half of each horizontal line is HIRES by switching between modes in alternating 33 and 32 cycle loops. Similarly, the screen can be split so that half of all the horizontal lines are LORES and the other half are HIRES by switching back and forth every 8515 cycles. The latter can be accomplished using the sample programs listed in Figures 3.10a and 3.10b.

The assembly language program of Figure 3.10a, when assembled, is a subroutine that performs the screen splitting. The BASIC program of Figure 3.10b sets up a color display and calls the machine language subroutine.

The example program causes the Apple to be in LORES for 131 TV lines and in HIRES for 131 TV lines. The display is aligned vertically by pressing any key on the keyboard while simultaneously holding REPT. The result of running this program is the split screen display pictured in Figure 8.9. A more sophisticated method of combining Apple II screen modes is discussed in an application note at the end of Chapter 5.

```
SOURCE FILE: FIGURE 3.10A
0000:              1 ************************************************
0000:              2 *                                              *
0000:              3 *                HIRES/LORES SPLIT             *
0000:              4 *                    8515/8515                 *
0000:              5 *                  BY JIM SATHER               *
0000:              6 *                    2/15/1983                 *
0000:              7 *                                              *
0000:              8 ************************************************
C000:              9 KBD      EQU  $C000
C010:             10 KBDSTRB  EQU  $C010
C054:             11 PAGE1    EQU  $C054
C056:             12 LORES    EQU  $C056
0000:             13 *
0000:             14 * THIS PROGRAM TOGGLES THE HIRES/LORES SWITCH
0000:             15 * EVERY 8515 CYCLES.
0000:             16 *
------ NEXT OBJECT FILE NAME IS FIGURE 3.10A.OBJ0
1F00:             17          ORG  $1F00
1F00:AC 54 C0     18 SPLIT    LDY  PAGE1
1F03:A0 27        19 SLEW     LDY  #39        ;SLEW SCREEN IF KEY PRESSED.
1F05:20 27 1F     20          JSR  WAITX10
1F08:AC 10 C0     21          LDY  KBDSTRB
1F0B:AC 00 C0     22 KEYCHK   LDY  KBD
1F0E:30 F3        23          BMI  SLEW
1F10:69 01        24          ADC  #1         ;TOGGLE HIRES/LORES SWITCH
1F12:29 01        25          AND  #$01
1F14:AA           26          TAX
1F15:BC 56 C0     27          LDY  LORES,X
1F18:A2 08        28          LDX  #8
1F1A:20 31 1F     29          JSR  WAITX1K    ;WAIT 8000 CYCLES
1F1D:A0 31        30          LDY  #49
1F1F:20 27 1F     31          JSR  WAITX10    ;WAIT 490 CYCLES
1F22:18           32          CLC
1F23:90 E6        33          BCC  KEYCHK
1F25:             34 *
1F25:             35 * TIMING ROUTINES:
1F25:             36 * WAITX10 WAITS Y-REG TIMES 10 CYCLES.
1F25:             37 * (MINIMUM WAIT 20 CYCLES)
1F25:             38 * WAITX1K WAITS X-REG TIMES 1000 CYCLES.
1F25:             39 *
1F25:D0 01        40 LOOP10   BNE  SVCYCLES
1F27:88           41 WAITX10  DEY             ;WAIT Y-REG TIMES 10
1F28:88           42 SVCYCLES DEY
1F29:EA           43          NOP
1F2A:D0 F9        44          BNE  LOOP10
1F2C:60           45          RTS
1F2D:48           46 LOOP1K   PHA
1F2E:68           47          PLA
1F2F:EA           48          NOP
1F30:EA           49          NOP
1F31:A0 62        50 WAITX1K  LDY  #98        ;WAIT X-REG TIMES 1000
1F33:20 27 1F     51          JSR  WAITX10
1F36:EA           52          NOP
1F37:CA           53          DEX
1F38:D0 F3        54          BNE  LOOP1K
1F3A:60           55          RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 3.10a Assembler Listing: A Screen Splitting Program.

```
10  REM
11  REM
12  REM  SET UP LORES AND HIRES AND CALL SPLIT SCREEN.
13  REM
14  REM
20  PRINT  CHR$ (4);"BLOAD SPLIT SCREEN.OBJ0"
30  HGR : HOME : VTAB 21: PRINT "1  7  D  2  8  E  B  4  5  A 3 6 C 9 F 8"
40  DIM COLR(39),X(21)
100  FOR A = 0 TO 39: READ COLR(A): COLOR= COLR(A): VLIN 0,39 AT A: NEXT A
200  FOR A = 0 TO 21: READ COLR(A): READ X(A): HCOLOR= COLR(A)
210  HPLOT X(A),0 TO X(A),159: NEXT A
220  FOR A = 8319 TO 16383 STEP 128: POKE A,64: NEXT A
300  CALL 7936
400  REM  LORES DATA
410  DATA  1,0,7,7,0,13,13,0,2,2,0,8,8,0,14,14,0,11,11,0
420  DATA  4,4,0,0,5,0,0,10,0,3,0,6,0,12,0,9,0,15,0,8
500  REM  HIRES DATA
510  DATA  4,0,3,20,4,21,3,41,4,42,7,62,7,83,7,104,3,105,7,125,3,126,7,159,3,161
520  DATA 7,180,3,182,3,206,7,220,3,233,7,247,3,262,3,263,7,279
```

Figure 3.10b BASIC Listing: A Split Screen Example.

SOFTWARE APPLICATION

APPLE TIMING LOOPS

It is not generally known that the 6502 clock of the Apple is not fixed frequency. The frequency and stability of the MPU clock are important factors in precision timed loop assembly language programs. The *Apple II Reference Manual* gives the 6502 frequency as 1.023 MHz, but it also gives a hint that there is more involved. In the WAIT routine of the old Monitor ROM listing, there is an interesting comment. It reads simply "1.0204 USEC." This is 1.02048432 rounded off incorrectly.

The composite frequency of the Apple is 1.0205 MHz. (The program comment in the reference manual confuses frequency with time periods.) 1.0205 is the result of 14.31818x(65/(65x14+2)). The average period of duration of an Apple 6502 machine cycle is .9799268644 microseconds. This is the value which should be used for computing exact time durations of Apple programs.

When very precise time measurement is necessary, the programmer has to consider the impact of clockpulse jitter, which is caused by the long cycle. Since the Apple II has no real time clock, timed output must be done with program loops which take a specific number of clock pulses to execute. When possible, these loops should be written in multiples of 65 cycles. This will eliminate loop output jitter. Otherwise the application must be able to tolerate a 140 nanosecond jitter. 140 nanoseconds is the difference between a normal cycle and a long cycle. The programmer should be aware of Apple clockpulse jitter and determine its affect on his particular application.

HARDWARE APPLICATION

DETECTING TELEVISION SYNC

There is no signal or interrupt in the Apple II to tell the 6502 where the television scan is at any moment. Reading television sync is of interest in several applications: screen mode splitting, video camera interfacing, flickerless animation, and light pen interfacing, to name four. This application note has some suggested methods for gaining the capability to detect television sync through peripheral card design. Since no signals from the video scanner are connected to the peripheral slots, some less obvious methods must be used to read the television scan.

A simple but powerful form of television scan interruption is shown in Figure 3.11. This circuit generates an interrupt when the video scanner overflows, 416 cycles before the start of the screen display. A jumper is connected between this peripheral card and D11, pin 15 on the motherboard to give the card its scan reference. The interrupt can be acknowledged, enabled, or disabled under program control via references to its DEVICE SELECT' addresses.* $C080.X (slot number times $10 in X) acknowledges the interrupt and inhibits further interrupting. $C081,X acknowledges the interrupt and enables further interrupting. System RESETs also disable interrupts.

*Acknowledging the interrupt allows the IRQ' line to return to the high state after the interrupt.

As an example of what this circuit will do when installed, the short 6502 assembly language program listed below will create a mixed mode with four lines of text at the top of the screen and a graphics display for the rest of the screen. The program assumes that the interrupter is installed in Slot 0. Load this program and call 24576 ($6000) in the middle of any BASIC program. The BASIC program will continue to execute but the screen will be split with text on the top four lines. Also, the BASIC program will run about 15% slower because time is wasted between the interrupt and the period just before the fifth line of text.

This circuit has several strengths and weaknesses. It is interrupt based, so it allows programs to execute normally, leaving the interrupt handler to process tasks related to the video scan. Also, it is very simple, requiring only one and a quarter integrated circuits. Its primary weakness is that it is capable of interrupting at only one spot on the scan. Another problem is that an interrupt does not stop program flow until the current instruction is executed. This means the point in relation to the video scan at which the interrupt handler is entered will vary plus or minus three cycles, depending on the length of the interrupted 6502 instruction. Another weakness is the required jumper to the motherboard. It is preferable to keep this sort of extraneous clutter to a minimum.

```
6000:  LDA #$00
6002:  STA $3FE
6005:  LDA #$61
6007:  STA $3FF  ;NMI Vector Fixed
600A:  CLI
600B:  LDA $C081 ;Enable Video Interrupter
600E:  RTS
6100:  LDA $C051 ;This is the IRQ handler
6103:  LDA #$1C
6105:  JSR FCA8  ;Wait 2351
6108:  LDA #$03
610A:  JSR FCA8  ;Wait 76
610D:  LDA $45   ;Apple stored accumulator here
6110:  BIT $C050
6112:  BIT $C081 ;Acknowledge the interrupt and
6115:  RTI       ;enable further interrupts
```

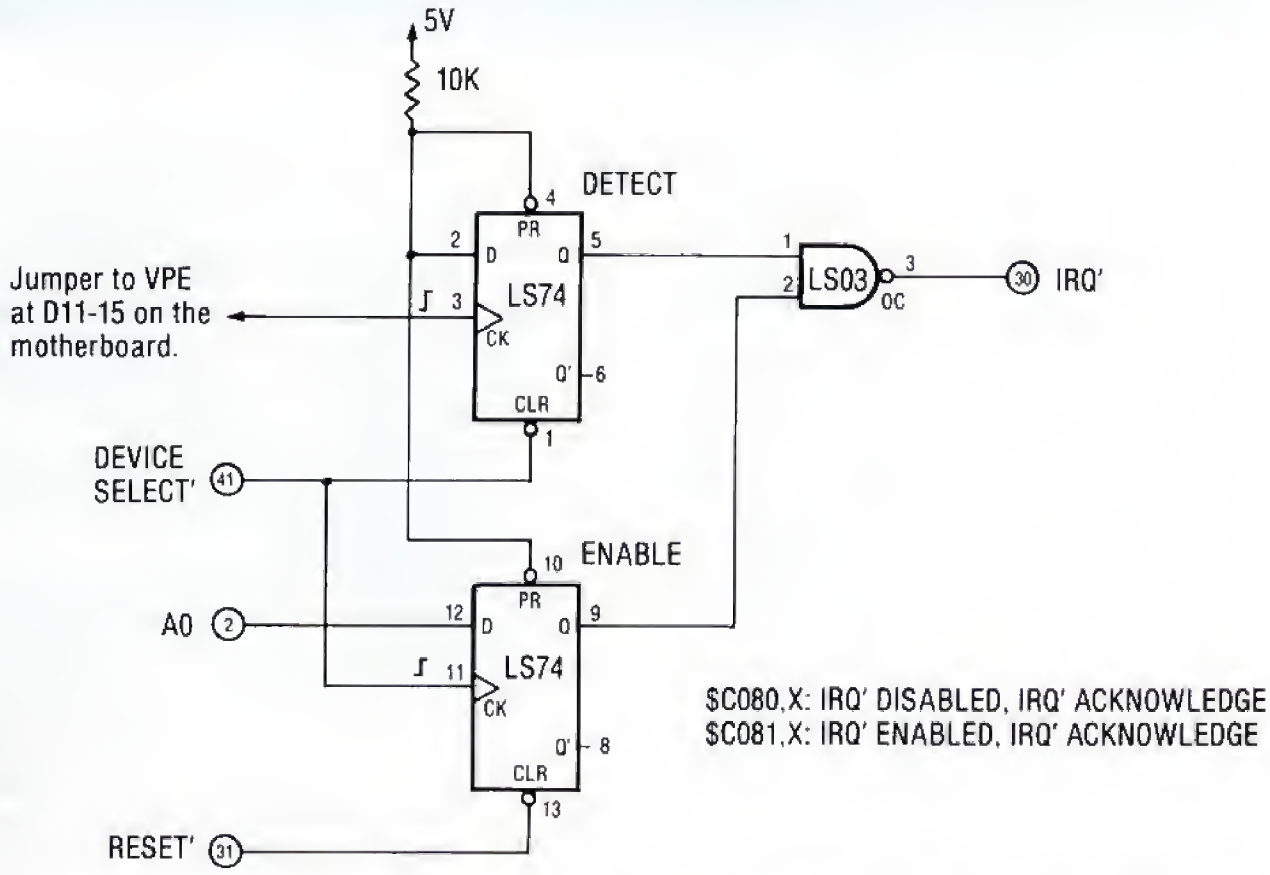


Figure 3.11 Circuit to Generate a 6502 Interrupt 416 Cycles Before the Start of the Screen Display.

One way to make a jumper unnecessary is to solder the wire between D11-15 and pin 35 or 19 on any slot between 0 and 6. Pins 35 on Slots 0-6 are connected together and the same is true of pin 19. No signal is connected to these pins, so they are generally available for some user defined purpose. You can solder the long wire on the underside of the motherboard, then tack it to the motherboard with several spots of glue. This entails removing the motherboard from the case, but it has the advantage of making the Figure 3.11 interrupter a cleaner installation.

It is not actually necessary to jumper video scanner signals to peripheral cards to allow them to sync to the television scan. Any slot can be synced with the video scanner, but it is a little easier in Slot 7, since the television SYNC signal is available there on pin 19. This signal is a composite of the horizontal and vertical sync signals which cause horizontal and vertical retrace on the television. The horizontal signal is easily detected by clocking a flip-flop or other synchronous device on the falling edge of the television SYNC signal. The vertical signal is slightly more difficult to detect. As described in Chapter 8, vertical sync is made up of four long duration negative pulses. Figure 3.12 is a circuit to separate a single vertical pulse from the television SYNC signal. It works by counting up while the SYNC signal is low and counting down when it is high. If it counts up to 15, then a vertical sync pulse is present and the VERTICAL SYNC flip-flop is set. When it counts back down to zero, the vertical sync period is past, and the VERTICAL SYNC flip-flop is reset. The vertical and horizontal sync may be used in any number of Slot 7 designs, possibly to drive an interrupter like the one in Figure 3.11.

This Slot 7 sync separator will not work on all Apples. The SYNC signal was not connected to pin 19 of Slot 7 in Revision 0. Also in Revision 0, the vertical sync was three times as long as later revisions with no horizontal serrations. You can make the sync separator work in Revision 0 by jumpering SYNC (C13-8) to pin 19 on any slot, but there would be no horizontal sync to detect during the long vertical sync period. A second problem occurs in Revision 7 motherboards. In Revision 7, video generation

was changed so that the three horizontal serrations in the vertical sync became double pulses. This double pulse was changed back to a single pulse in the RFI Revision. The result is that digital processing of the Apple SYNC signal must tolerate or mask out the second pulse if it is to work with Revision 7 Apples.

Some readers might be surprised to find it is also possible to sync a non-Slot 7 peripheral card to the video scanner without connecting jumpers to the motherboard. In fact, it is possible to duplicate the entire video scanner on a peripheral card with no jumpers to the motherboard. The reason for this is that data from the video scanner is present on the data bus when PHASE 0 rises on all 6502 read cycles. Suppose you build a counter just like the video scanner and put it on a peripheral card. Call it the scanner simulator. Then devise hardware to read flag bits from the data bus. Finally, cause the scanner simulator to preset when the flag bits are detected. The flag bits are set up by a program, and the scanner simulator preset is enabled by program control. This only has to be done once, after which the two video scanners will remain in sync until power is removed from the Apple.

Figure 3.13 is a five chip circuit which duplicates the vertical portion of the video scanner on a peripheral card. It is made up of a mod-262 counter, a long pulse detector, and a synchronizer to sync the counter to the vertical section of the video scanner. The long pulse detector simply detects the Apple's long cycle, which occurs once every horizontal scan at the beginning of the horizontal blanking period. This long pulse is easily detected from any peripheral

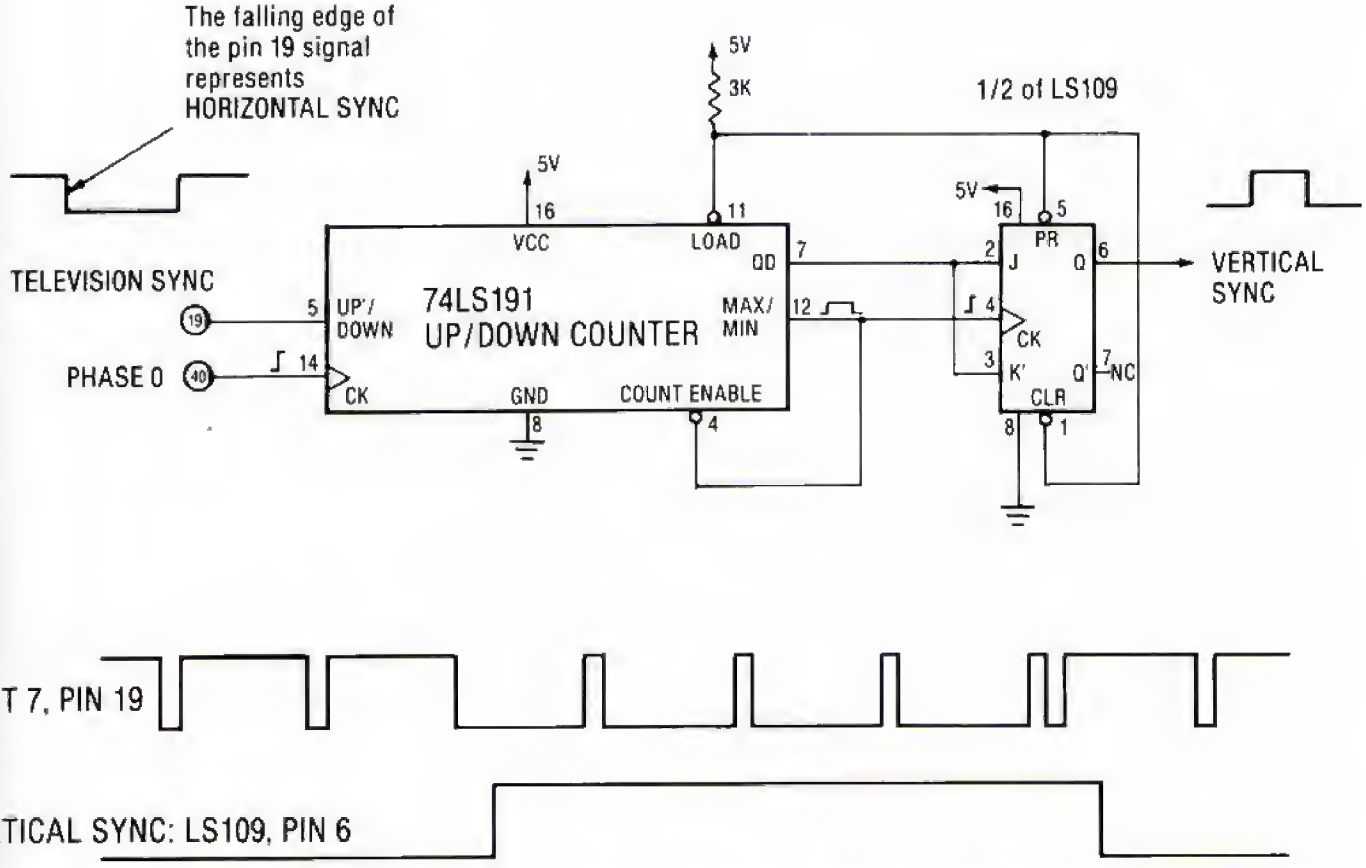


Figure 3.12 Circuit to Separate a Vertical Sync Pulse from the Television Sync Signal at Pin 19 of Slot 7.

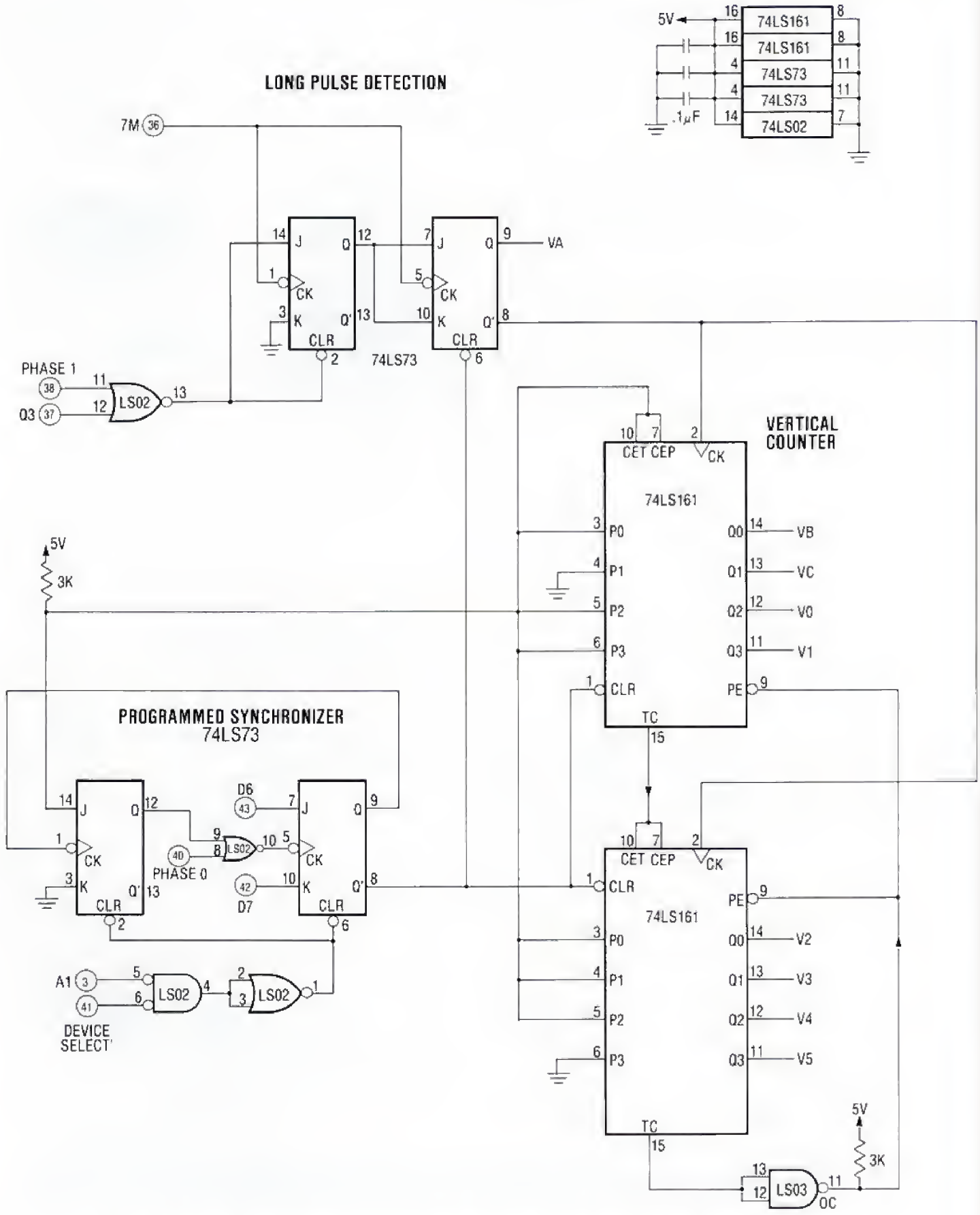


Figure 3.13 Circuit that Duplicates the Vertical Section of the Video Scanner.

card, because the 7MHz clockpulse falls twice during PHASE 0 • Q3' of the long cycle, but only once during PHASE 0 • Q3' of a normal cycle. In Figure 3.13, every time 7M falls twice during PHASE 0 • Q3', the vertical counter counts. This means the vertical counter counts horizontal scans.

The vertical counter is nearly identical to the vertical portion of the video scanner. It uses 74LS161 counters which preset on overflow to -262. The LSB (VA) is mechanized on one half of a 74LS73 flip-flop. The main difference between this vertical counter and the video scanner is that this counter has its CLEAR input connected to the programmed synchronizer. This circuit will detect the top displayed line of the television scan and clear the vertical counter. The synchronizer is designed to detect flags stored in bits D6 and D7 of scanned memory. Memory is flagged as follows:

1. Clear all HIRES, Page 1 ($2000-$3FFF). HIRES, Page 1 is selected because clearing it will not clobber BASIC or DOS.
2. Place a $40 at memory location $3FD0. (See Figures 5.9 and 5.17.) This is the first byte scanned during line 191 displayed, and it is only scanned once per television scan.
3. Place a $80 at memory location $3827. This is the last byte scanned during line 6 displayed.

Once memory is flagged, the video counter is synced by performing a reference to $C080,X then waiting 21686 or more cycles in a non-writing wait routine. What happens is this:

1. The reference to $C080,X enables monitoring of D7 and D6 while PHASE 0 rises.
2. Within 17030 cycles, $3FD0 is scanned, driving out D6 true. This causes the CLEAR line to drop low, clearing the vertical counter and holding it clear.
3. The CLEAR line is held low for 5045 cycles until $3827 is scanned, driving out D7 true. This brings the CLEAR line high and disables further monitoring of D7 and D6. This allows the vertical counter to start counting with the first long cycle at the beginning of line 1.

Figure 3.14 is an assembly language subroutine to sync the video counter. After running this subroutine, the counter will remain in sync in all video modes until the computer is turned off or an accidental reference is made to $C080,X. Control commands can be made at addresses with A1 set ($C082,X for example) and they will not affect the video counter.

Figure 3.15 is an example of what one could do with the video scan information on a peripheral card. This circuit will generate an interrupt before any one of the 256 horizontal scans beginning with line 0. The eight bit interrupted scan number is saved in an octal latch, and a pair of 74LS85 magnitude comparators tell when the video scan is equal to the line number to be interrupted.

The circuits of Figures 3.13 and 3.15 are combined on the prototype card pictured in Figure 3.16. This programmable scan interrupter represents a considerable enhancement to the Apple II, implemented on just eight TTL chips. Program control of this card in Slot 0 would be as follows:

$C080	Synchronize scanner simulator as in Figure 3.14 program.
$C082	IRQ' disabled; IRQ' acknowledged.
$C083	IRQ' enabled; IRQ' acknowledged.
STA $C08X	Set number of scan to be interrupted.

The circuits shown in this application note are useful by themselves but could also be used in combination with other circuits. The resourceful designer can use video scan synchronization, long pulse detection, scan interruption, and Slot 7 sync separation to create exciting new capabilities for the Apple.

```
SOURCE FILE: SYNC INTERRUPTER SLOT7
0000:              1 ************************************************************
0000:              2 *                                                          *
0000:              3 *                                                          *
0000:              4 *            SYNCHRONIZE VIDEO SCAN SIMULATOR              *
0000:              5 *                                                          *
0000:              6 *                        BY JIM SATHER                     *
0000:              7 *                                                          *
0000:              8 *                           1/20/83                        *
0000:              9 *                                                          *
0000:             10 *                                                          *
0000:             11 ************************************************************
0000:             12 *
0000:             13 *
0006:             14 BASL     EQU  $06
0007:             15 BASH     EQU  $07
0070:             16 SLOTNUM  EQU  $70
3827:             17 LINE6    EQU  $3827
3FD0:             18 LINE191  EQU  $3FD0
C050:             19 GRAFIX   EQU  $C050
C052:             20 NOMIX    EQU  $C052
C054:             21 PAGE1    EQU  $C054
C057:             22 HIRES    EQU  $C057
0000:             23 *
0000:             24 *
0000:             25 *
----- NEXT OBJECT FILE NAME IS SYNC INTERRUPTER SLOT7.OBJ0
1000:             26          ORG  $1000
1000:AD 50 C0     27 SYNCIT   LDA  GRAFIX
1003:AD 52 C0     28          LDA  NOMIX
1006:AD 54 C0     29          LDA  PAGE1
1009:AD 57 C0     30          LDA  HIRES
100C:A9 20        31          LDA  #$20       CLEAR SCREEN MEMORY
100E:AA           32          TAX
100F:85 07        33          STA  BASH
1011:A9 00        34          LDA  #0
1013:85 06        35          STA  BASL
1015:A8           36          TAY
1016:91 06        37 BLANKLP  STA  (BASL),Y
1018:88           38          DEY
1019:D0 FB        39          BNE  BLANKLP
101B:E6 07        40          INC  BASH
101D:CA           41          DEX
101E:D0 F6        42          BNE  BLANKLP
1020:             43 *
1020:             44 *
1020:A9 40        45          LDA  #$40       SET FLAGS
1022:8D D0 3F     46          STA  LINE191
1025:A9 80        47          LDA  #$80
1027:8D 27 38     48          STA  LINE6
102A:A2 70        49          LDX  #SLOTNUM
102C:BD 80 C0     50          LDA  $C080,X    ENABLE D7 & D6 MONITORING
102F:A2 11        51          LDX  #17        WAIT 21829 CYCLES
1031:88           52 WAITLP   DEY
1032:D0 FD        53          BNE  WAITLP
1034:CA           54          DEX
1035:D0 FA        55          BNE  WAITLP
1037:60           56          RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 3.14 Assembler Listing: Syncronizing the Video Scan Simulator.

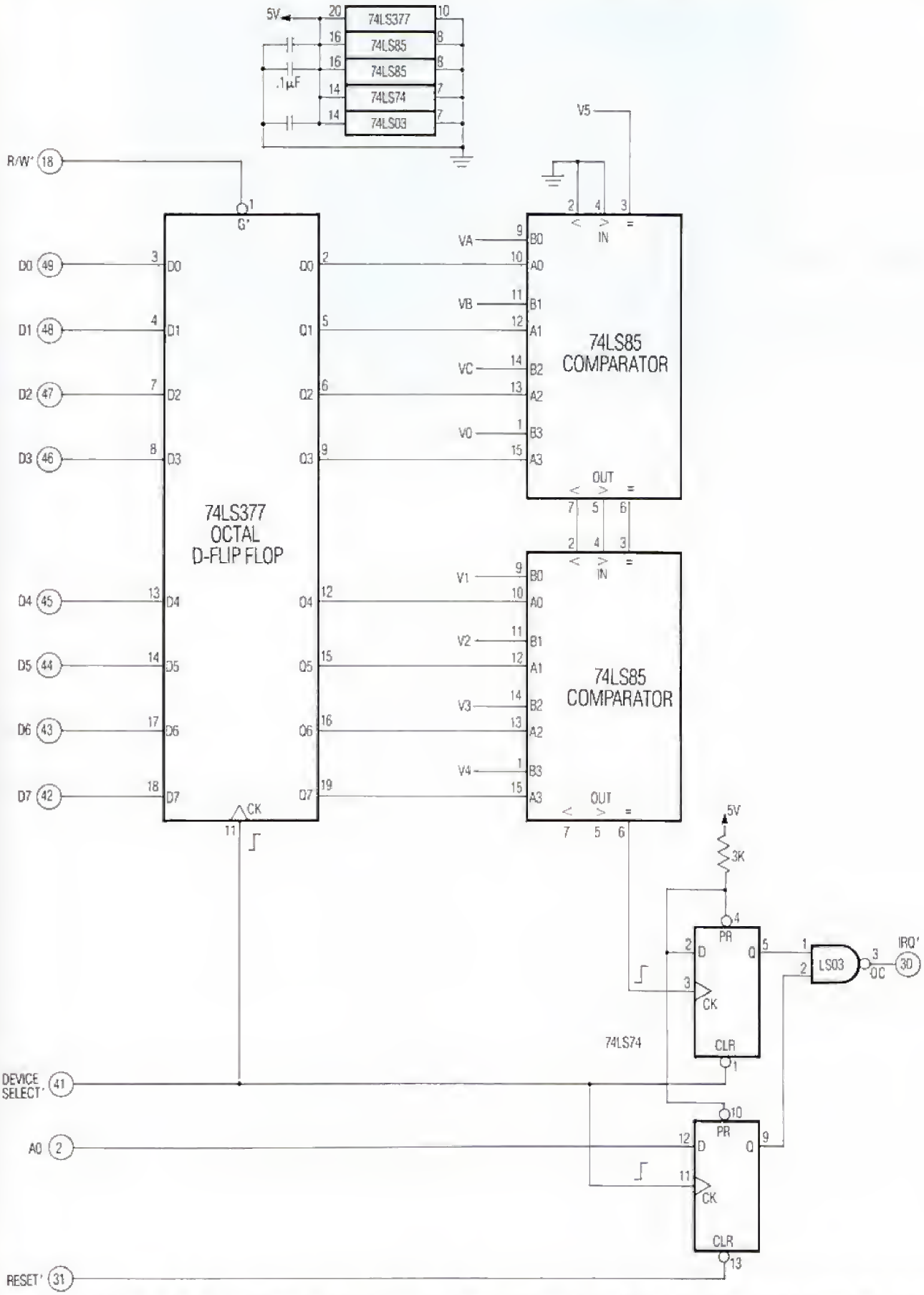


Figure 3.15 A Programmable Interrupter. It Generates an Interrupt Before Any One of 256 Selected Horizontal Lines.

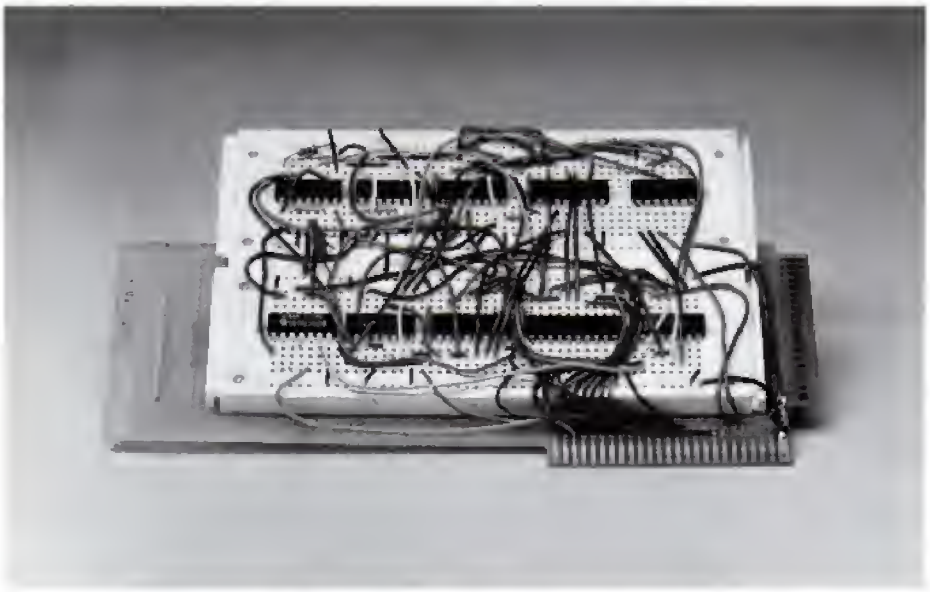
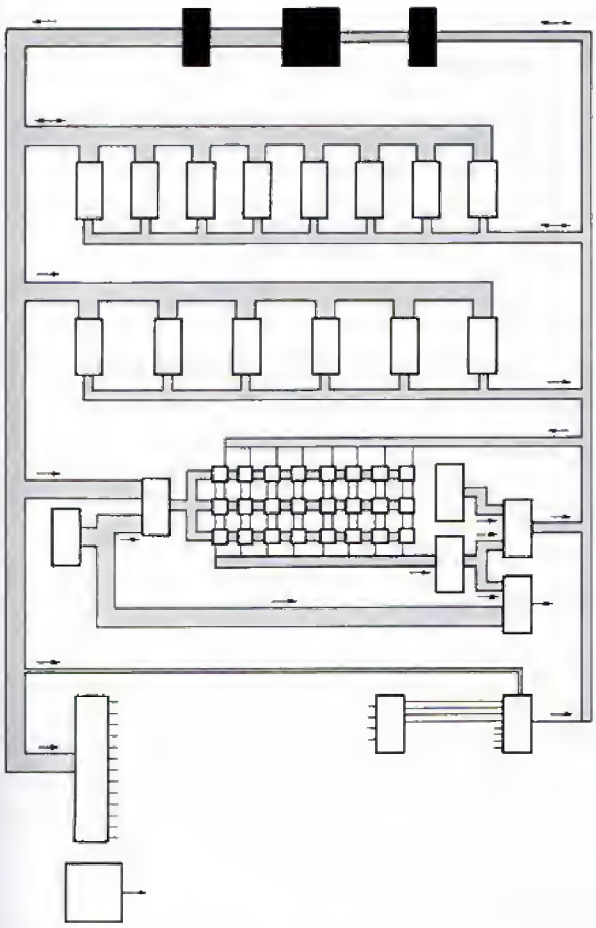


Figure 3.16 A Programmable Video Interrupter Card.



chapter 4

The 6502 Microprocessor

The 6502 was designed by MOS Technology in the mid 1970s as part of their MCS6500 series microprocessor family. It has been a popular choice as a microprocessor for personal computers, being used in computers produced by Apple, Atari, Commodore, Ohio Scientific, Rockwell International, and other manufacturers. Only the Z80 could compete as the MPU most often found in big selling microcomputers. The 6502 gives adequate computing speed and versatility at a very low cost. Its programming language is very simple, making it an ideal MPU for the occasional computer programmer.

The most important 6502 related knowledge for an Apple owner to attain is programming knowledge. The ability to read and write 6502 assembly language programs greatly expands the horizons of an Apple computerist. 6502 assembly language is not, however, a major topic of this book. These pages are concerned primarily with the hardware implementation of the 6502 in the Apple II computer. Volumes have been written about various aspects of the 6502, especially programming. The choice of 6502 topics in this chapter was governed by the unique features of 6502 use in the Apple, and by the goal of this book to fill information gaps in Apple literature available to the public.

Manufacturers of the 6502 are:

MOS Technology, Inc.
950 Rittenhouse Rd.
Norristown, Pa. 19403

Synertek, Inc.
P.O. Box 552
Santa Clara, Ca. 95052

Rockwell International
Microelectronic Devices
P.O. Box 3669, RC55
Anaheim, Ca. 92803

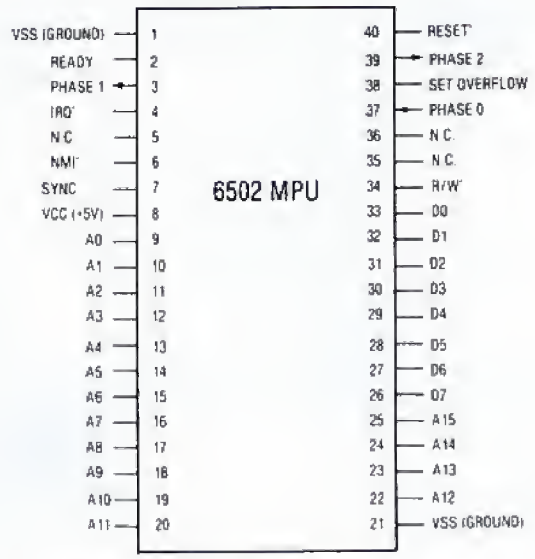


Figure 4.1 6502 Pin Assignments.

6502 SIGNALS

There are 40 pins on a 6502, three of which serve no function. In addition to the address output and data input/output, there are four outputs (R/W', PHASE 1, PHASE 2, and SYNC) and six inputs (READY, IRQ', NMI', PHASE 0, SET OVERFLOW', and RESET'). There are three power supply connections. One pin requires +5 volts and two pins require ground. Figure 4.1 shows the 6502 pin assignments. Figure 4.2 shows the 6502 hardware implementation in the Apple. A brief discussion of the 6502 signals with Apple implementation notes follows.

Clockpulses—PHASE 0, PHASE 1, PHASE 2

The 6502 has most of its required clockpulse generation circuitry built-in. It requires only an externally generated time base which can be implemented in several ways. In the Apple, the PHASE 0 time base is developed independent of 6502 internal circuits and fed as the clockpulse input to the 6502.

The 6502 generates its required PHASE 1 and PHASE 2 clocks from the PHASE 0 input. PHASE 1 is high during the first half of a machine cycle, and PHASE 2 is high during the second half of a machine cycle. PHASE 1 is not the simple inversion of PHASE 2. There is a slight delay between the PHASE 1 transitions and the PHASE 2 transitions. The rising edge of one always follows the falling edge of the other. The PHASE 1 and PHASE 2 clocks are available at pins 3 and 39 of the 6502. PHASE 2 is not connected in the Apple. PHASE 1 is used to control the direction of data flow in the external MPU data bus transceiver during MPU write cycles.

Address and R/W'

During every machine cycle, the 6502 places an address on its address output. In association with the address it outputs, it brings its R/W' line high or low, thereby telling the world whether it wants to read or write data. With 16 address lines, the 6502 is capable of producing 65536 different values at its address output.

The 6502 address and R/W' outputs are not tri-state, but in the Apple these signals are connected to the address bus through external **tri-state bus drivers**. This enables peripheral cards to gain access to the address bus via the DMA' line.

Data Bus

The data input/output of the 6502 is eight lines. This gives the 6502 its overall classification as an 8-bit microprocessor. Data direction is inward except during PHASE 2 of write cycles.

In the Apple, the 6502 data lines are connected to the data bus through external **bidirectional bus drivers**. These drivers enable the 6502 to write to the data bus with all the devices that are connected to it. Data direction in the external data drivers is from the data bus to the 6502 except during write cycles when the 6502 PHASE 1 clock is low.

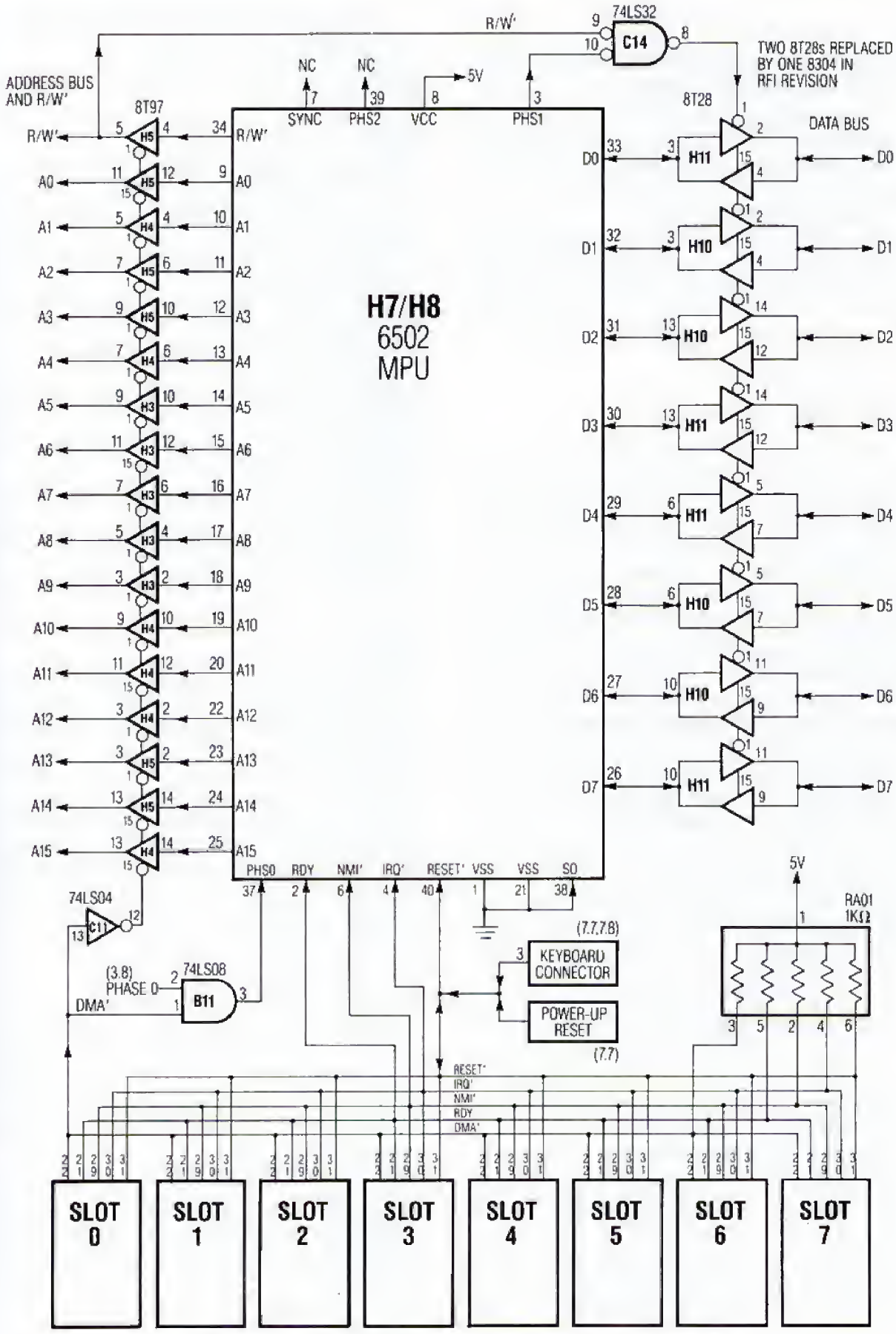


Figure 4.2 Schematic: 6502 Connections in the Apple II.

RESET'

The RESET' input to the 6502 causes the 6502 to start or restart. A RESET' causes the 6502 to disable interrupts and begin program execution at an address stored in locations $FFFC and $FFFD of ROM. 6502 operation is inhibited while RESET' is held low. The RESET' sequence begins when RESET' transits from low to high.

In the Apple the RESET' line is connected to pin 31 of the peripheral slots, to the keyboard RESET key, and to a circuit which generates a RESET' when the computer is first turned on. A peripheral card can either cause a RESET', respond to a RESET', or ignore a RESET'.

Interrupts—IRQ' and NMI'

The interrupts cause the 6502 to stop its sequential program execution and execute interrupt handling routines. Interrupts are normally associated with input/output functions, but they are a way for any type of device to get the microprocessor's attention. The IRQ' (Interrupt ReQuest) is enabled or disabled by program control, so the 6502 doesn't have to respond to an IRQ'. The NMI' (Non-Maskable Interrupt) cannot be disabled by program control.

An NMI' causes the 6502 to perform an interrupt sequence after the current 6502 instruction has been executed. The 6502 saves its program location counter and its Status Register (with BREAK flag reset) in an area of RAM called the stack. It disables interrupt requests, then begins program execution at the address stored in $FFFA and $FFFB of memory. The NMI' input to the 6502 is edge sensitive, meaning the 6502 responds only to a high to low transition of NMI'. To generate a second interrupt, the NMI' line must be brought high, then low again.

An IRQ' causes the 6502 to perform an interrupt sequence after the current instruction has been executed if the program has interrupt requests enabled. The IRQ' sequence is identical to the NMI' sequence except the address of the IRQ' handling routine is stored at $FFFE and $FFFF. The IRQ' signal is not edge sensitive, so the IRQ' must go high before interrupts are enabled again, or the same interrupt will be answered more than once.

The interrupt inputs to the 6502 are connected to the peripheral slots in the Apple. There are no motherboard devices which generate 6502 interrupts, and I/O in the Apple II is normally accomplished without interrupts. Most real time clock cards are capable of generating interrupt requests, and cards which dump Apple memory to disk are based on non-maskable interrupts. The IRQ' line is tied to pin 30 of the peripheral slots and the NMI' line is tied to pin 29.

READY

Bringing the READY input to the 6502 low during the PHASE 1 clock in a read cycle causes the 6502 to go into its wait state. In the wait state, the 6502 holds the current address and does nothing. The wait state lasts until READY is sensed high during PHASE 2. If the high to low transition occurs during a write cycle, the wait state will not begin until the next read cycle.

The wait state of the 6502 can be used for interfacing to slow memories, single step operation, slow step operation, or just plain stopping the MPU indefinitely. It has no impact on the Apple's video circuitry or on RAM refresh, so the video display appears frozen on the screen when the 6502 is halted via the READY line. The READY line in the Apple is connected only to pin 21 of the peripheral slots. This 6502 capability has gone largely unexploited in the Apple.

SYNC

The 6502 SYNC output goes high when the 6502 is performing an op code fetch. This is the first cycle in the execution of any instruction in which the 6502 fetches the one byte operational code of the instruction. The SYNC signal can be used for single instruction execution steps (in conjunction with the READY line) and otherwise identifying the op code of a 6502 instruction. The SYNC output of the 6502 is not connected to anything in the Apple. It cannot be used in the Apple without modifying the motherboard or jumpering it to a circuit.

SET OVERFLOW'

A high to low transition on the SET OVERFLOW' line sets the overflow flag of the 6502. The overflow flag is normally set or reset as a logical result of some 6502 instructions, but the SET OVERFLOW' input forces the flag regardless of instruction execution.

The SET OVERFLOW' input has limited value as a control input, because it must be used only in conjunction with instructions that affect the overflow flag or in avoidance of such instructions so as not to interfere with them. It is not used in the Apple, but is tied directly to ground.

6502 CONNECTIONS IN THE APPLE

Figure 4.2 shows the 6502 hardware implementation in the Apple. Address, R/W', and data connections are routed to the address bus and data bus through external drivers. The PHASE 0 clock comes from the timing generator, gated by DMA' from the peripheral slots. All other signals are connected directly to the peripheral slots.

The method of tying the 6502 control inputs to multiple sources is called **wire-ORing** or **collector-ORing**. A logical OR function is achieved by tying lines directly together. As an example, if Slot 0 OR Slot 1 OR any other slot pulls pin 29 low, the 6502 will sense a non-maskable interrupt. In a wire-OR circuit, the line is pulled high by a voltage through a resistor if no card is pulling the line low. Peripheral cards should not try to pull the wire-OR lines high. They either pull the line low or present a high impedance to the line, usually by driving the line with open collector TTL circuits. The 6502 literature specifies that 3000 ohm pull-up resistors be used for wire-OR inputs to the 6502. Apparently this is not a particularly important specification because the Apple works fine with 1000 ohm pull-up resistors.

The tri-state address bus driver is necessary for DMA operations because the 6502 address and R/W' connections are not tri-state. The address drivers used in the Apple are 8T97 6-bit tri-state bus driver. The data bus drivers are 8T28 4-bit transceivers in older Apples. The two 8T28s were replaced by a single 8304 in the RFI Revision. These transceivers increase the data bus driving capability of the 6502 on write cycles. On read cycles they have a different effect. The Apple data bus is a heavy capacitive load and it takes a while for a device to bring it low or especially to bring it high. The transceiver acts like a threshold detector on read cycles. As the data bus gradually changes states, the voltage at the transceiver input reaches a threshold at which the transceiver output rapidly changes states. This presents a cleanly switched data input to the 6502 as opposed to the dirty data bus.

6502 MEMORY USAGE

Use of the 6502 in the Apple dictates various aspects of the memory layout. For example, addresses $0-$1FF are always RAM in a 6502 system. Apart from design dictates, the 6502 also uses parts of memory so that they are normally not available for Apple programs.

Page 0 and Page 1 ($0-$FF and $100-$1FF) of a 6502 system must be RAM simply because the 6502 has special read/write uses for Page 0 and Page 1. **Page 0** locations are used as **indirect address locations** in 6502 machine language. Additionally, the 6502 has a **zero page addressing mode** which speeds and compacts programs making heavy use of zero page locations for various storage functions. As a result, big machine language programs like Applesoft BASIC make heavy use of zero page locations. If BASIC is operating and you indiscriminately POKE values into zero page locations, you will deep six BASIC. This is because the critical pointers of BASIC will be lost. The following program must crash:

```
10  FOR A = 0 TO 255 : POKE A,0 :
    NEXT A : END
```

Page 1 is the 6502 stack. The stack of a microprocessor is an area of RAM which it uses as a **last in—first out memory**. To the computer program, the stack is like a stack of playing cards which it can discard to or draw from. Conceptually, data is stored to the top of the stack or withdrawn from the top of the stack. The stack is actually part of RAM. While the program pushes data to or pulls data from the stack, the MPU must increment or decrement a read address and keep track of where in memory the "top" of the stack is. In the 6502, the location of the "top" of the stack is stored internally in an 8-bit register called the **Stack Pointer**. When the stack is accessed, the 6502 addresses a location in Page 1 of RAM determined by the Stack Pointer. Virtually all machine language programs access the stack via Jump SubRoutine and Return from SubRoutine instructions, so at no time can a program indiscriminately modify Page 1.* The following BASIC program will crash as surely as the earlier one:

```
10  FOR A = 256 TO 511 :
    POKE A,0 : NEXT A : END
```

The 6502 also dictates that the highest memory location is $FFFF and that it will be assigned to ROM. That $FFFF is the highest address is an obvious consequence of the fact that the 6502 has 16 address lines. In a similar vein, the eight data lines

*Exceptions are copy protect schemes which call for programming without JSR, RTS, PHP, PLP, PHA, or PLA instructions. In these schemes, critical data is stored in Page 1 of memory, and most attempts to examine memory result in the loss of the critical Page 1 data.

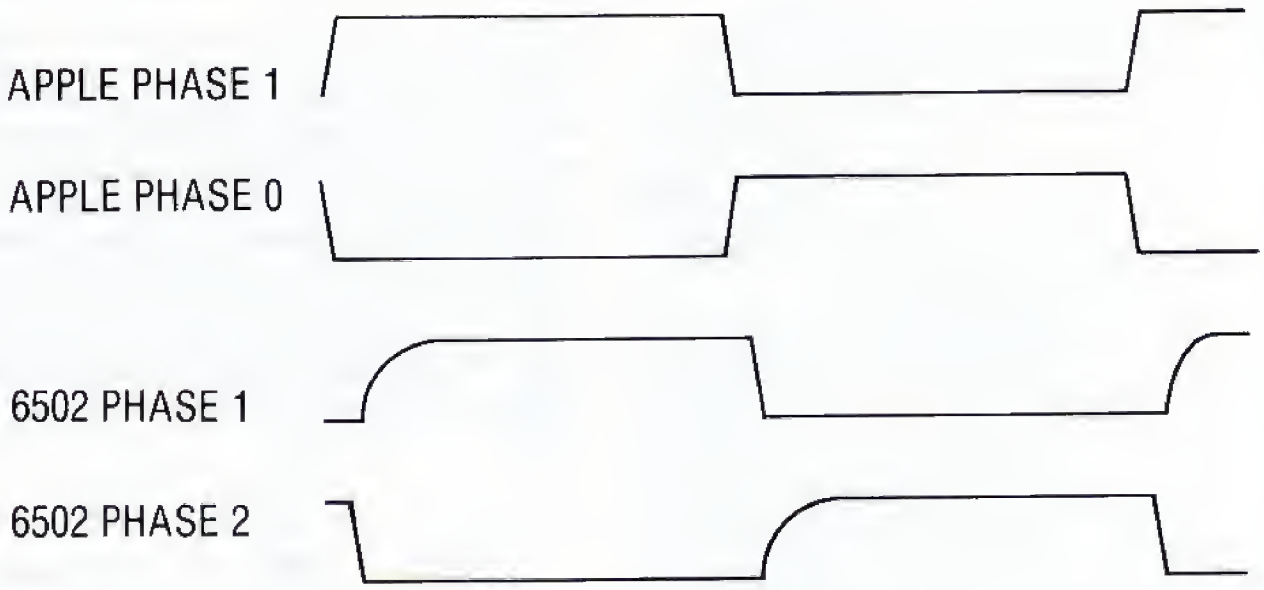


Figure 4.3 6502 Clockpulse Relationships.

of the 6502 dictate that memory is organized into 8-bit locations. The reason for assigning the highest address to ROM is that the 6502 RESET, NMI, and IRQ vectors must be stored in locations $FFFA through $FFFF. In particular, the RESET vector in ROM enables the Apple to immediately begin executing a non-erasable program at power up.

Since the 6502 has no special input/output control features, it must control input/output functions with commands decoded from the address bus. In the Apple, addresses are assigned to the peripheral slots and built-in I/O functions which could be otherwise assigned to memory. This is referred to as **memory mapped I/O.** It was logical in the Apple design to assign the address space between RAM and ROM to I/O. That way there are three contiguous addressing groups RAM ($0-$BFFF), I/O ($C000-$CFFF), and ROM ($D000-$FFFF).

6502 TIMING IN THE APPLE II

The 6502 was designed to be similar to the Motorola MC6800 microprocessor, but improved. The clock requirements of the 6502 are the same as the MC6800, two alternating positive pulses. In the MC6800, the two clocks must be generated externally and input. In the 6502, the two clocks are generated internally from the PHASE 0 clock input. This is one of the 6502 improvements.

The relationship between the PHASE 0 clock input and the PHASE 1 and PHASE 2 6502 clocks is shown in Figure 4.3. The PHASE 1 and PHASE 2 clocks are not symmetrical but are low slightly longer then they are high. The high period of one clock always fits neatly inside the low period of the other. The PHASE 1 and PHASE 2 transitions are clocked by the transitions of the PHASE 0 input in a repetitive cycle. The falling edge of PHASE 0 is followed by the falling edge of PHASE 2 and then the rising edge of PHASE 1. The rising edge of PHASE 0 is followed by the falling edge of PHASE 1 and the rising edge of PHASE 2. To put it differently, PHASE 0 falling clocks the end of PHASE 2 then the beginning of PHASE 1, and PHASE 0 rising clocks the end of PHASE 1 then the beginning of PHASE 2.

The effect of the **long cycle** on 6502 clocks is to elongate PHASE 2 by 140 nanoseconds. This has no particular ill effects outside of program timing considerations mentioned in the previous chapter. By lengthening PHASE 2, all response criteria for communicating with the 6502 become less critical. The following timing discussions are valid for either a normal cycle or a long cycle, but the diagrams picture normal cycles. The timing specifications of the 6502 are not affected by the long cycle.

The 6502 PHASE 1 clock is not the same as the PHASE 1 signal developed in the timing generator. PHASE 1 from the timing generator is simply PHASE 0 inverted. It was named PHASE 1 because of its kinship with the 6502 PHASE 1 clock. Semantic ambiguity is a great way to confuse those who would understand. The term which is distributed to the peripheral slots, address decode, and RAM is PHASE 1 from the timing generator. The 6502 PHASE 1 clock is used only inside the 6502 and for control of the external MPU bidirectional data bus driver. In this chapter only, "PHASE 1" refers to the 6502 PHASE 1 clock. Outside of this chapter "PHASE 1" refers to the inversion of PHASE 0, distributed from the timing generator.

Timing specifications in the 6502 are referenced to the rising and falling edge of the PHASE 2 clock (at the .4V point). Important 1 MHz timing specifications for MOS Technology 6502s are shown here with Synertek and Rockwell International ratings shown in parenthesis when they differ:

1. The 6502 address and R/W' line will be valid within 300 nanoseconds (225 nsec Synertek; 225 nsec Rockwell) after the falling edge of PHASE 2. They will stay valid until at least 30 nanoseconds after the next falling edge of PHASE 2. The address becomes valid during PHASE 1.
2. 6502 write data will be valid within 200 nanoseconds (175 nsec Synertek; 175 nsec Rockwell) after the rising edge of PHASE 2. The write data will remain valid until at least 30 nanoseconds (60 nsec Synertek) after the falling edge of PHASE 2.
3. 6502 read data must be valid at least 100 nanoseconds (50 nsec Rockwell) before the falling edge of PHASE 2 and must be held valid at least 10 nanoseconds after the falling edge of PHASE 2. PHASE 2 falling is the 6502 data transfer clock.
4. The maximum delay between PHASE 0 falling and PHASE 2 falling is 65 nanoseconds. The maximum delay between PHASE 0 rising and PHASE 2 rising is 75 nanoseconds. These values are specified only by Synertek and only with a 100 picofarad load on PHASE 2.

The time periods represent worst case conditions over an operating range from 0 to 70 degrees centigrade. Worst case timing specifications are shown in Figure 4.4. MOS Technology time values are used because they are more conservative than Synertek and Rockwell and, therefore, represent the actual worst case. Synertek values for PHASE 0 to PHASE 2 delay are used because MOS Technology and Rockwell don't publish this important specification. Ten nanoseconds could probably be subtracted from the clockpulse delay specifications to reflect the fact that there is no load on PHASE 2 in the Apple.

The Synertek, MOS Technology, and Rockwell International 6502s are probably all made the same. When one company gives tighter specifications than another, it obligates itself to test its microprocessors using more difficult criteria. We shall soon see that the worst case timing specifications are pretty far beyond typical operation in any case. They had better be, because the Apple design doesn't meet the MOS Technology/Synertek 100 nanosecond minimum requirement for 6502 read data setup when reading from RAM.

Figure 4.5 is a diagram showing the timing relationships actually found in one Apple. The measurements were made in an Apple using a Synertek SY6502 marked 78360 (December 26, 1978?). Figure 4.5 may be considered fairly typical of 6502 timing in the Apple. The important features of Figure 4.5 are:

1. PHASE 1 and PHASE 2 transitions occur roughly 30-35 nanoseconds after PHASE 0 transitions at the peripheral slots.
2. The 6502 address becomes valid 128 nanoseconds after PHASE 0 falls at the peripheral slots. This indicates a setup time of under 100 nanoseconds from PHASE 2 falling, far under the 300 nanosecond maximum. In a typical Apple, the address is valid before Q3 falls during PHASE 1. This is probably true of most Apples and perhaps all Apples.
3. Write data becomes valid at the data bus 120 nanoseconds after PHASE 0 rises. This is well under the MOS Technology maximum of 200 nanoseconds setup time from PHASE 2 rising. 6502 write data must be valid before CAS' falls for it to be read by RAM. CAS' in the Apple falls 209.5 nanoseconds after PHASE 0 rises, but CAS' falling is delayed to RAM by 20 nanoseconds typical and 32 nanoseconds maximum. Therefore, 6502 write data in the Apple that becomes valid on the data bus 200 nanoseconds after PHASE 2 begins to rise will be read correctly if the PHASE 0 to PHASE 2 delay is no more than about 40 nanoseconds.
4. Read data from RAM becomes valid at the 6502 just 35 nanoseconds before PHASE 0 falls or about 67 nanoseconds before PHASE 2 falls. Thus the Apple design does not meet the 100 nanosecond setup time required by MOS Technology and Synertek. A 100 nanosecond setup time is unusually long. One wonders which magician pulled that number out of his hat.

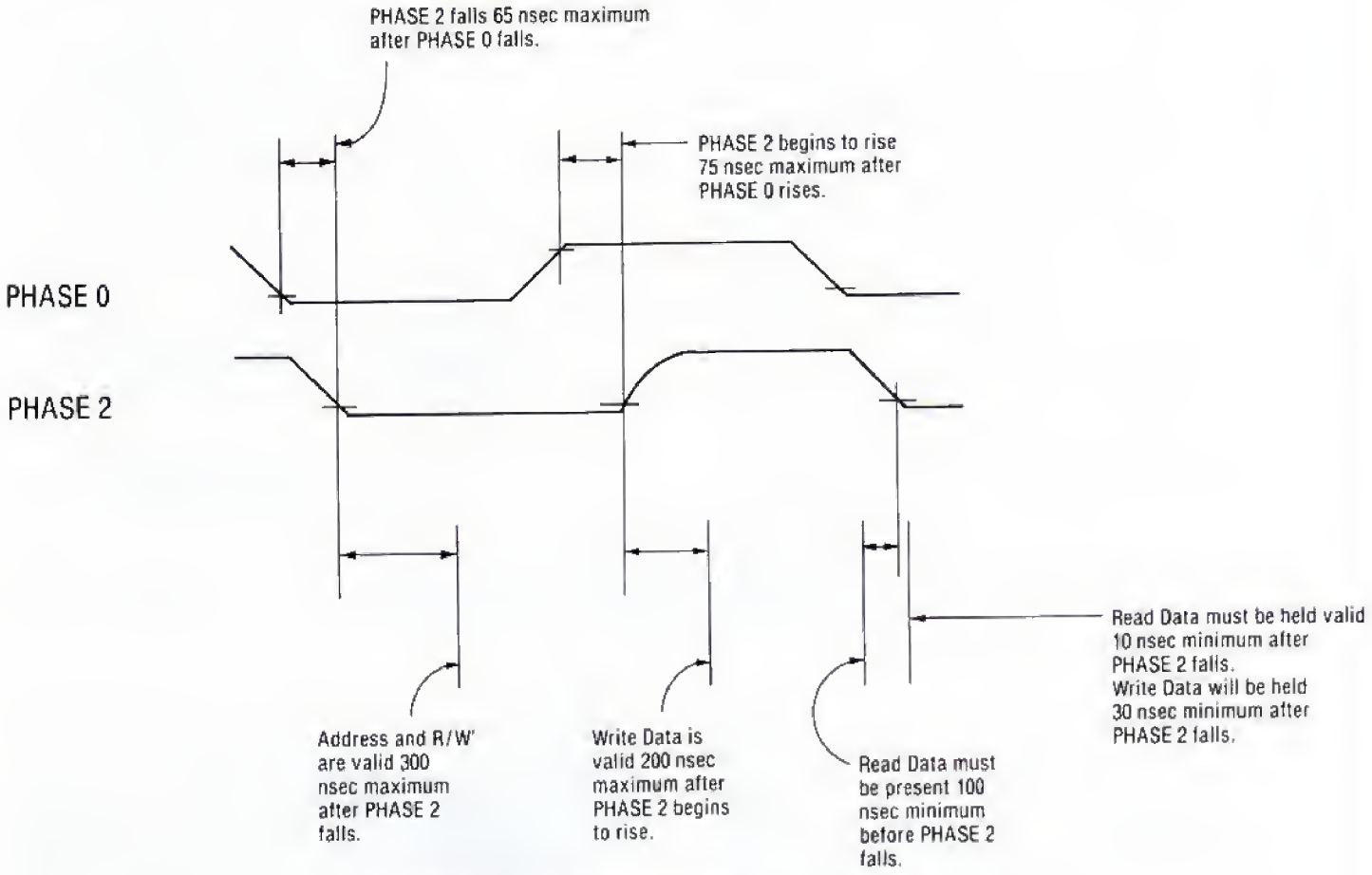


Figure 4.4 Some Worst Case 6502 Specifications.

Specific timing of 6502 communication with various Apple devices will be discussed in chapters covering those devices. Listed below are some general guidelines for Apple peripheral design. They serve to illustrate the basics of Apple bus management. The primary data bus management signal is the RAM SELECT' term generated in the RAM address multiplexor. The RAM chapter should be studied to clarify the details of 6502 communication in the Apple.

1. The 6502 address can be read before PHASE 0 in time to trigger a DMA action that same cycle.
2. Write data from the 6502 can be clocked to a peripheral by the falling edge of PHASE 0.
3. Read data from peripherals should be valid on the data bus by 45 nanoseconds before the end of PHASE 0, and should stay valid at least 40 nanoseconds after PHASE 0 has fallen. 45 nanoseconds does not meet the MOS Technology/Synertek 100 nanosecond data setup specification, but it is not necessary to do so. If it were, the 6502 could not read RAM in the Apple.
4. At approximately 60 nanoseconds after PHASE 0 falls, the latched RAM data output is gated to the data bus. Peripherals should present a high impedance to the data bus by this time.

The requirements for read data being on the data bus before and after PHASE 2 can be met in a peripheral by gating read data with **DEVICE SELECT'. This signal does not overlap PHASE 2, but the data stays valid on the data bus until after PHASE 2 anyway.** When the data bus of the Apple is floated (when all devices on the data bus present a high impedance to the data bus), **the last**

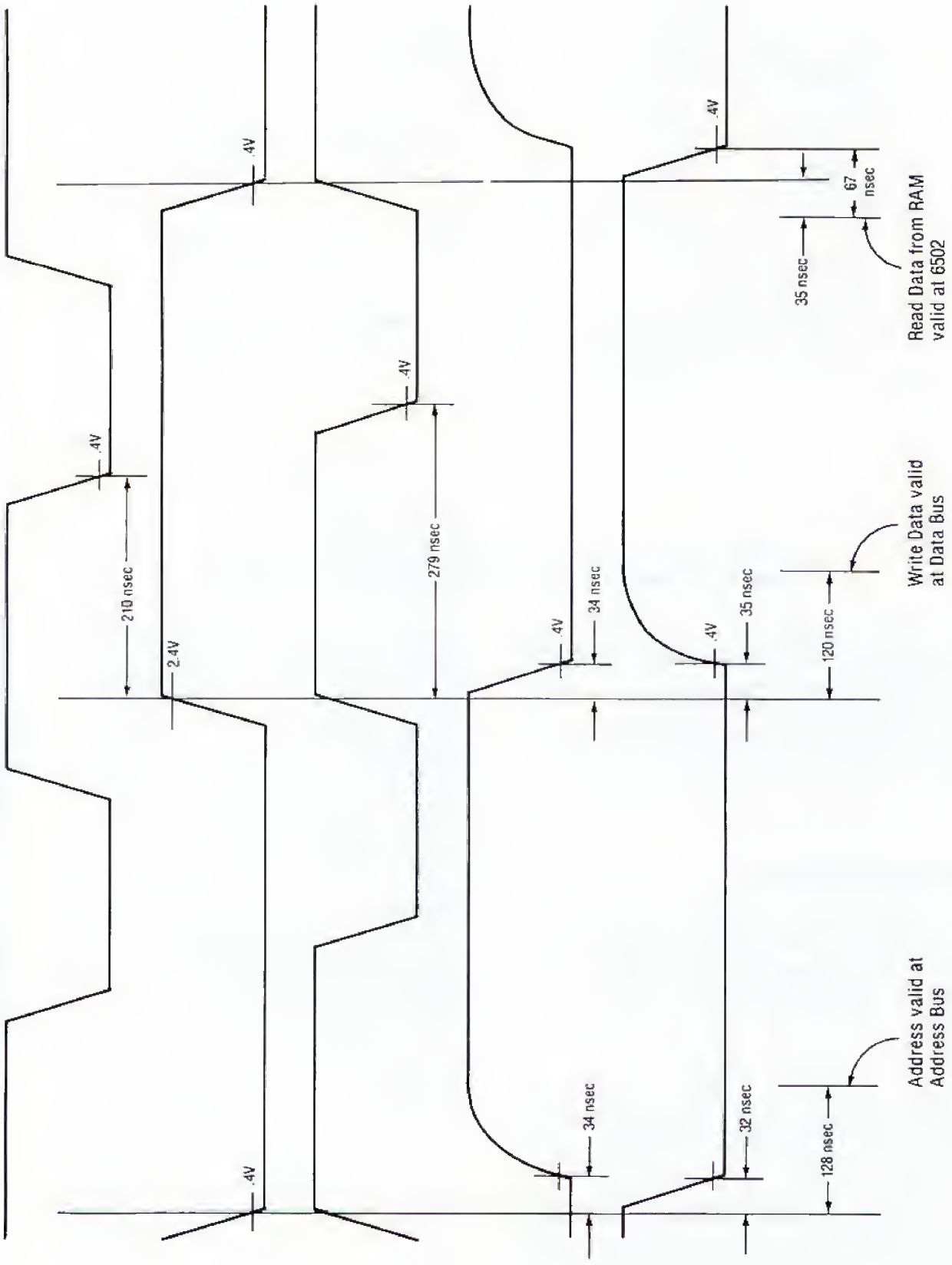


Figure 4.5 Experimental 6502 Timing Relationships.

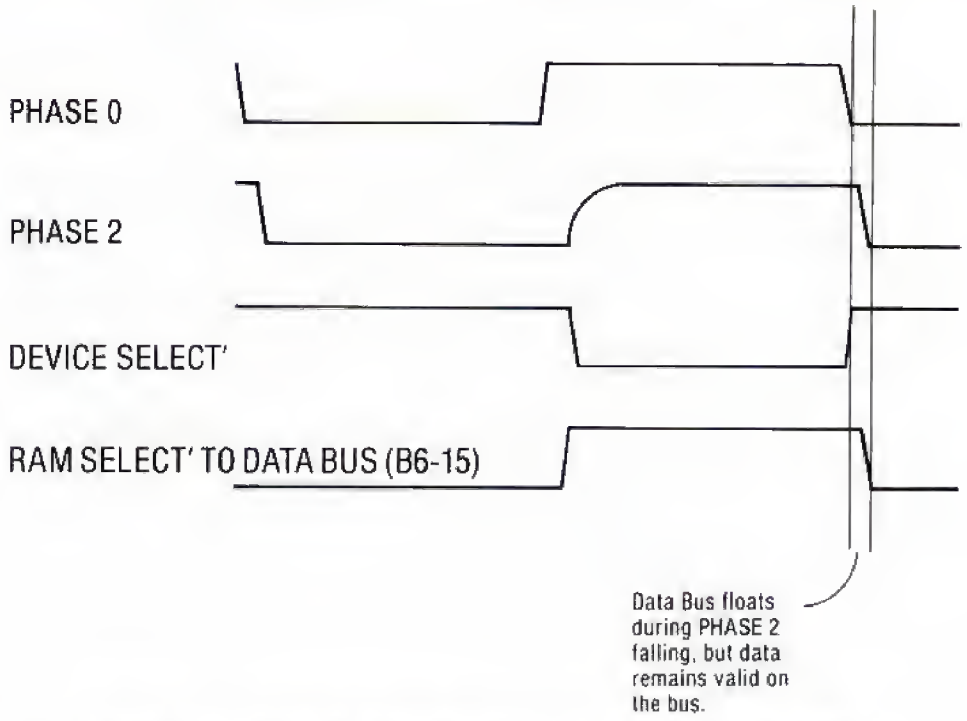


Figure 4.6 Reading the Disk Input Port Using Device Select'.

valid data on the data bus at the time it was floated remains valid until the bus is brought back under positive control. Therefore, if the data bus is floated just before PHASE 2 falls, the 6502 will still read the last valid data before the data bus was floated.* As an example, Figure 4.6 shows a read access to a peripheral slot data input address.

APPLE PROGRAMMING

There are four levels at which programs can be written in the 6502 based Apple: 6502 machine language, 6502 assembly language, high level compiler language, and high level interpreter. The order of listing is from most difficult to least difficult.

*I assume that the data is stored on the data bus via capacitive charges. I am not sure how long it would take to discharge the bus, because the bus is normally floated for a maximum of about 700 nanoseconds (during a LDA $C050 in a long 6502 cycle, for example). It takes much longer to discharge the bus than to charge it. A 74LS257 can bring the data bus to 63% of its high state in about 35 nanoseconds. Assuming a collector impedance of 100 ohms in the LS257, this indicates a charging capacitance of 350 picofarads. The disk controller (see Chapter 9) is an example of a peripheral which uses DEVICE SELECT' to gate data to the data bus for reading by the MPU. The controller could not successfully transfer data to the MPU if it weren't for the slow degradation of data when the data bus is floated. Since the disk controller works reliably, it is my conclusion that the DEVICE SELECT' signal can be used to gate data to the data bus for transfer to the MPU in Apple peripheral designs.

A **6502 machine language program** is a series of numeric bytes. The bytes are stored in sequence in memory where the 6502 accesses them by incrementing the address bus and reading the program while executing. 6502 machine language instructions consist of one, two, or three bytes in succession. Each instruction consists of an **op code** and possibly a one or a two byte **operand**. Execution of a three byte instruction requires three cycles to fetch the instruction plus additional cycles to execute the instruction.

The 6502 has a set of **internal registers** which are manipulated by the program. A 6502 program performs its functions by overseeing the interplay among the internal registers and memory. The 6502 internal register complement is made up of five 8-bit registers and the 16-bit Program Counter. The following is a list of the registers and their functions:

REGISTER	FUNCTION
Program Counter	Contains current address of instruction being executed.
Accumulator	Principle arithmetic and logical register.
X-register	Index register.
Y-register	Index register.
Stack Pointer	Contains current stack address.
Status Register	Contains flags indicating 6502 operating modes and logical results of instructions.

Generally, programs center around the Accumulator and memory with the X- and Y-registers being used for address indexing. Values of the Program Counter, Stack Pointer, and Status Register are automatically kept by the 6502 and don't usually have to be accessed directly by the program. Provisions exist for direct control of the Stack Pointer and processor status. The Program Counter is controlled by the flow of the program.

The following is a three instruction 6502 machine language program listed in hexadecimal:

OP CODE	ADDRESS LOW	ADDRESS HIGH
AD	89	1D
85	16	
00		

The first instruction loads the 6502 Accumulator from address $1D89. The second instruction stores the 6502 Accumulator contents at address $16. The final instruction is a BREAK instruction which terminates the program. The purpose of the program is to transfer the contents of $1D89 to $16. Machine language programs may be entered and executed from the Apple monitor using methods described in the *Apple II Reference Manual*.

Assembly language is a way of writing machine language programs with computer assistance. Many aspects of machine language programming are performed better by computer than by humans. Some such aspects are remembering op codes, addition and subtraction of addresses, remembering addresses of subroutines, and checking for syntax errors. Assembly language assists the programmer with these and other details and allows the use of English language symbology for addresses, operands, and opcodes. A prime goal in computer language development is English language compatibility.

The same program that was listed above in machine language is listed here in assembly language.

```
LABEL   OP CODE ADDRESS   COMMENT
RESTORE   LDA     $1D89   RESTORE SAVED POINTER
          STA     $16
          BRK
```

This program contains English language which cannot be executed by the 6502. A computer can, however, take this program and convert it to 6502 machine language program. A program that does this is an assembler. An assembler takes an assembly language **source program** and assembles from it a machine language **object program**.

6502 programs can be assembled in disk-based Apples using any of several commercially available assemblers. This is the best way for most Apple owners to write extensive 6502 programs. Compared to almost any computer, minicomputer, or microprocessor machine language, 6502 machine language is very simple to use. This extends to 6502 assembly language. There are only 56 mnemonic codes to learn, and the logical selection of mnemonics makes this a simple learning task.

The simple instruction set has advantages and disadvantages. The chief disadvantage is that in some instances a program will require more instructions to accomplish a purpose than it would if powerful special purpose instructions were available. This can result in loss of speed and waste of memory space in some programs. One should not get the idea that 6502 is without powerful features. It has a very versatile set of addressing modes and a decimal mode which speed execution of certain types of programs considerably. It's just that there are more powerful and complex microprocessors around.

The chief advantage of the simple instruction set is ease of programming. A second advantage is that it made it easier for the Apple designer to include a 6502 **disassembler** and **Mini-Assembler** in the firmware of the original Apples. The Mini-Assembler in ROM is an invaluable aid to students and practitioners of 6502 programming. It comes as an associated utility with Integer BASIC and is available to Apple users who have Integer BASIC.

The Mini-Assembler has some characteristics of a full assembler. It automatically translates mnemonic op codes and 6502 assembly language convention operands to 6502 machine language code. It also automaticaly computes relative branch references from absolute address entries and checks for syntax errors. In other words, it allows machine language code to be immediately translated and entered from assembly language keyboard entries. Some very important assembly language capabilities are lacking in the Mini-Assembler. There is no English language symbolism except for the op code mnemonics. Also, there is no relocatability feature which is inherent with full assemblers. 6502 machine language is usually not relocatable, but machine code can be assembled to operate at any valid memory location from an assembly language source program. Still, there is no easier way to write and enter a short 6502 program than on the Apple's Mini-Assembler.

As an example for Apple users who may never have used the Mini-Assembler before, here is a short sequence of keystrokes to enter some 6502 code. Start by entering Integer BASIC. If you do not have Integer BASIC, you probably don't have the Mini-Assembler.

```
SCREEN PROMPT /
KEYBOARD ENTRIES      REMARKS
>CALL-2458            Enter Mini-Assembler
! 1000:  LDX #10
! JSR FF3A            Use Apple's BELL
                      routine
! DEX
! BNE 1002
! RTS
! [CTRL-RESET]        Re-enter BASIC
>CALL 4096            Call Bellringer
```

This program calls the BELL routine in Apple firmware 16 times. This is the end of assembly language instruction in this book. Persons who wish to learn more can do so by reading the *Apple II Manual* and studying the listings of the Autostart ROM and Monitor ROM. There are several 6502 assembly language books which one can purchase. The best way of all to learn is to buy an assembler and start programming.

Another way to produce machine code involves the use of **compilers**. Programs may be written in high level languages such as BASIC, Pascal, and FORTRAN. High level language programs consist of powerful symbolic commands such as "PRINT" and "=". A 6502 cannot execute such commands, but computer programs (compilers) can examine such commands and produce 6502 machine language code which will cause the 6502 to perform the indicated functions.

A compiler is like an assembler in that it takes a symbolic language source program and translates it to a machine language object program. It is different from an assembler in that whole machine language routines are generated by a single compiled instruction. Only one machine language instruction is generated by an assembly language instruction. High level languages are much more powerful than assembly language in easing the task of the programmer. However, machine language code compiled from high level languages is generally less efficient than code assembled from assembly language programs. The programmer has direct control over the machine code generated in assembly language, and human minds generate more efficient code than compiler programs. With compilers, as with assemblers, symbolic source code must be entered with the assistance of a text editor. The compiler source code must be compiled into machine language object code before a program can be run.

In some ways, this process of converting a high level language program to machine code is a nuisance. The object code must be compiled before it can be run and debugged. In an alternate process, a high level language program can be interpreted as it is run. The interpreting program examines the high level language commands during program execution, and it directs program flow to resident machine language routines which perform the indicated functions. This is the process used with the Applesoft and Integer BASIC languages supplied with the Apple II computer.

Both the compiling and interpreting processes are available for high level languages in the Apple II. In addition to the Applesoft and Integer BASIC interpreters in common usage, compilers are available that will compile stored Applesoft and Integer programs into machine language routines. These routines will execute much faster than an interpreter performing the same function, because the time consuming interpretation process is separated from execution. Compilers and interpreters for other high level languages are also available.

Which language should you program in—assembly language or a high level language? The answer depends not only on the programmer's background, experience, and personal preference, but also on the requirements of the particular application. Assembly language is fastest and provides the most efficient use of memory space. Some programs requiring speed or large amounts of memory can be written only in assembly language. Machine code compiled from high level language source code offers a great combination of programming ease and speed of execution. BASIC programs interpreted and executed by the firmware interpreter supplied with the Apple are the easiest of all to write and debug, but very slow in execution.

Whatever language you program in, the 6502 will be executing machine language code. All of the important Apple operating systems—BASIC, Pascal, DOS, the monitor, and the Mini-Assembler—are machine language code which was originally written in assembly language.

An important footnote while discussing Apple programming languages and operating systems is the secondary MPU which may replace the 6502 via the DMA' line. These secondary MPUs greatly expand the possibilities of what one might find

operating in the Apple. Of particular importance is the Z80 card and the associated CP/M operating system. CP/M (Control Program for Microprocessors) is a disk operating system developed by Digital Research company for which many programs are available. The Apple with Z80 card is potentially the most important CP/M computer.

DMA IN THE APPLE

DMA (Direct Memory Access) refers to a form of fast I/O in which the I/O device directly accesses memory. In DMA, the MPU is removed from the data transfer path between the device and RAM. There is no program sequence loading data from the source and storing it at the destination.

The video scanner access to RAM while PHASE 0 is low is a form of DMA referred to as simultaneous DMA. It is possible because RAM can be accessed twice as fast as the MPU access in the Apple and because actual MPU data transfers occur only during a short period at the end of the 6502 machine cycle. This simultaneous DMA is completely transparent to the MPU. It has no effect on program execution since it does not affect the 6502 machine cycle.

A second form of DMA is cycle stealing. In cycle stealing DMA, the clock input to the MPU is stopped for a machine cycle, and the DMA device accesses RAM while the MPU is stopped. Thus, a cycle is stolen from the MPU. This type of DMA slows program execution.

Cycle stealing DMA is implemented in the Apple. The DMA' line is wired to pin 22 of the peripheral slots and any peripheral card can directly access RAM by pulling DMA' low while PHASE 0 is low and holding DMA' low until PHASE 0 goes high then low again. Pulling DMA' low forces the MPU address bus driver to a high impedance state and gates off the PHASE 0 clock input to the MPU. With DMA' low, even though PHASE 0 goes high at pin 40 of the peripheral slots and everywhere else on the motherboard, PHASE 0 does not go high at pin 37 of the MPU. The 6502 waits with PHASE 1 high and PHASE 2 low. PHASE 1 high forces the direction of the MPU external data bus driver to be from the data bus to the MPU. The MPU is thus isolated from the address bus and data bus, and the peripheral card can take control of both buses and the R/W' line. DMA devices should communicate with the data bus at the end of PHASE 0 as the 6502 does. In the Apple, PHASE 1 belongs to the video scanner and devices do not respond to addresses except during PHASE 0.

Figure 4.7 shows the timing for stealing a cycle from the 6502 on the Apple. It would be wise not to steal too many cycles at a time, because if the clock is stopped too long, the 6502 will lose its internal data and the program will be bombed. It is not clear how long the clock can be stopped before the 6502 operation becomes unreliable. The MOS Technology data sheet lists the maximum PHASE 0 pulse width at 520 nanoseconds. This is clearly not accurate because every Apple in the world operates very well with a 629 nanosecond PHASE 0 pulse on one out of 65 cycles. The *Osborne 4 & 8-Bit Microprocessor Handbook* (copyright 1981, McGraw-Hill, Inc. by Adam Osborne and Gerry Kane) states that you cannot stretch the PHASE 1 or PHASE 2 clocks on MCS6500 microprocessors. Osborne and Kane must

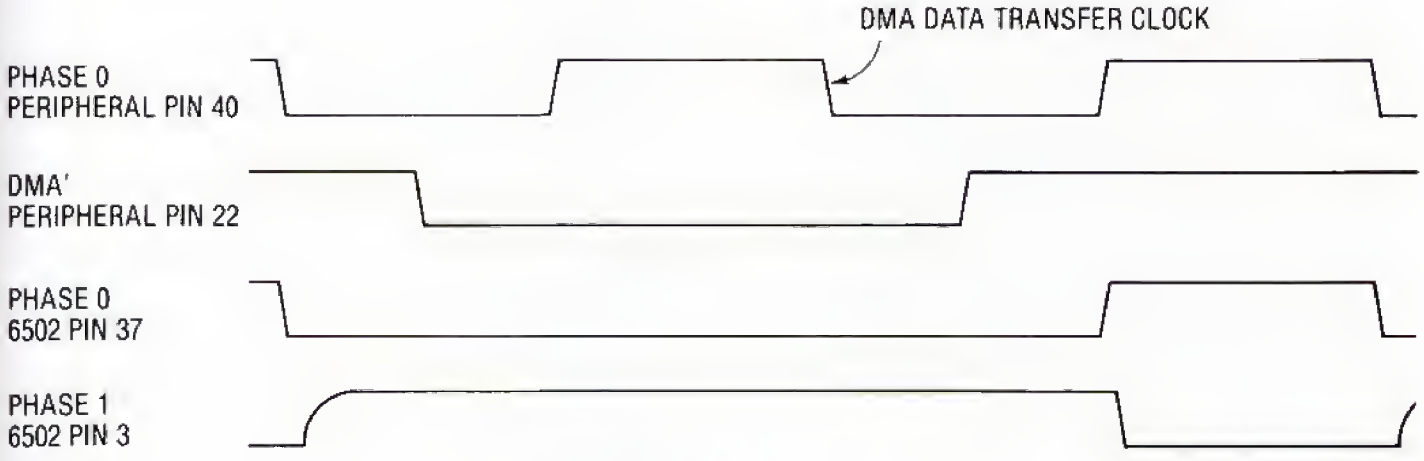


Figure 4.7 Cycle Stealing DMA.

have read the same data sheet. The Synertek data sheet for SY650X microprocessors shows a maximum cycle time of 40 microseconds. This seems to indicate that you can perform DMA in the Apple for 40 consecutive PHASE 0 cycles without adversely affecting the Apple. The Rockwell International data sheet shows a maximum cycle time of 10 microseconds which is probably a good number.

It happens that Steve Wozniak, the designer of the Apple II, knows a great deal about this subject. In a conversation with the author, Mr. Wozniak revealed that his original design for the Apple II used a different method of scanning memory for video output than the simultaneous DMA used in his final design. When he was designing the Apple II, RAM chips which could be accessed at 2 MHz were just becoming available. As a consequence, the early design had a 1 MHz 6502 from which 40 out of 65 cycles were stolen for memory scanning. The 6502, therefore, effectively ran at about 385 KHz (25/65 x 1 MHz). What Mr. Wozniak found out was that you could hold off the clock on a new 6502 for 40 microseconds, but that as the chip cooked in, this hold off capability deteriorated. He found it necessary to keep new 6502s handy so he could replace the MPU when the Apple started to malfunction. The 6502s were not failing. They were just becoming unable to retain data for 40 microseconds with the clock stopped. Mr. Wozniak speculates that the reason for this is a deterioriation in capacitance of internal elements after the 6502 is run for a while.

Mr. Wozniak never determined the maximum reliable hold off time of the 6502 experimentally. The availability of faster RAM chips enabled him to design the superior version of the Apple II which was eventually released. His feeling is that it is safe to hold off the clock to a 6502 for five microseconds, which is the value used in Microsoft's Z-80 card. He also cautions that any experimental determination of this capability would have to be performed on new 6502s, used 6502s, and very used 6502s.

It's pretty obvious that the DMA' line can be used for more than just direct access to RAM. Since 6502 control of the Apple is via address decode, any device controlling the address bus can control the Apple. For example, a very simple peripheral card could change Apple screen modes via pushbutton. It would just have to steal a single cycle from the 6502 and gate $C05X to the address bus during PHASE 0 to select a screen mode depending on which button had been pressed. The most common use of the DMA' line in the Apple is to operate an MPU other than the 6502 from a peripheral slot. A Zilog Z80 card, Motorola MC6809 card, Intel 8088, or what have you can be plugged in to allow control of the Apple by the owner's favorite MPU. These cards gain access to the Apple via the DMA' line.

The DMA' line has no effect on video scanner access to RAM, since the video scanner is isolated from the address bus. The DMA device must make its access during PHASE 0, however, for Apple devices to respond properly. Apple circuits listen to the address bus only during PHASE 0.

The 6502 designers intended that the **READY line** be used for DMA. Their idea was to stop the 6502 in a read cycle and bring an external tri-state address bus driver to high impedance with the READY line while DMA took place. The READY line in the Apple has no effect on the tri-state address bus driver, so DMA can only be accomplished by pulling the DMA' line low. The DMA' line can be pulled low in conjunction with the READY line, but after a number of cycles the 6502 will lose its internal data, because it has no input clock. Long DMA operations with the MPU in a wait state must be accomplished by bringing the READY line low and performing cycle stealing if it is important that the 6502 restarts coherently from where it stopped.

There is a **priority system** of DMA operation in the Apple in which the lowest peripheral slot has priority if more than one peripheral tries to perform DMA at the same time. The priority system is implemented by a DMA in/DMA out priority chain which goes from slot to slot. Pin 27 is the DMA input on each peripheral slot which tells a card that no higher priority card is performing DMA. Pin 24 is the DMA output by which DMA from lower priority cards is disabled. Slot 0 has the highest priority and Slot 7 has the lowest priority. Pin 24 on each slot is tied to pin 27 on the next slot with Slot 0, pin 27 and Slot 7, pin 24 not connected as shown in Figure 7.9. Please note that in Revision 7 and later Apples, pin 24 of Slot 7 is connected to jumper pad 7 as shown in Figures 7.9 and 7.10.

In the priority system, when pin 27 is low a card should not attempt DMA, because a higher priority card is performing DMA. The card should also bring pin 24 low so lower priority DMA cards are disabled. If pin 27 is high, a card may perform DMA. It should bring pin 24 low while performing DMA and bring it high while not performing DMA. Non-DMA cards are always designed with pin 24 jumpered to pin 27 so they can be inserted between DMA cards in the peripheral slots. This keeps the priority chain intact. There can be no empty slots between DMA cards in a priority chain.

The DMA priority chain can be used to prioritize other functions besides DMA. Apple does this with its firmware cards which substitute peripheral card ROM for motherboard ROM. Several firmware cards can be placed in a priority chain which prevents ROM on two separate cards from being simultaneously enabled. If two groups of cards use the DMA chain for different purposes, they may have to be separated by an empty slot or by a card with pin 27 or pin 24 open. For example, a firmware card in Slot 0 would interfere with the operation of a DMA card in Slot 1. Even when a firmware card is enabled, cycles are available when RAM or I/O is accessed in which the DMA priority line stays high. A DMA device down line from the firmware card will operate if it needs only to steal an occasional cycle, and can wait for the firmware card to access RAM.

6502 INTERRUPTS IN THE APPLE

There are actually four types of 6502 interrupt: RESET', NMI', IRQ', and the BREAK instruction. Each has its own unique characteristics and purposes as determined by the 6502 design. The hardware interrupts are connected to the peripheral slots, and RESET' is also connected to the RESET key and **power-up reset** circuit. The BREAK instruction is a **software interrupt**. The response of the 6502 to interrupts in the Apple is determined by programs contained in the F8 ROM.

RESET'

Except for RESET', the general idea of the interrupts is to interrupt the MPU, perform an interrupt handling routine, and then return to the interrupted program. The general idea of RESET' is to interrupt the MPU and go to a coherent program start. There are no provisions in the 6502 response to a RESET' for saving internal registers and returning to the place where the program was interrupted. The 6502 response to RESET' is as follows:

1. Pull three meaningless values from the stack.
2. Fetch the RESET routine address from $FFFC and $FFFD, low byte first.
3. Set IRQ' disable bit of Status Register; leave other Status bits as they are.
4. Begin execution of RESET routine.

The reason for the three meaningless stack accesses is that RESET' is a modified form of the other interrupts with R/W' forced high. Accordingly, the Stack Pointer is decremented while the three values are being read from memory, as if data were being pushed to the stack. Normally, the Stack Pointer is incremented during pull operations and decremented during push operations.

In the original Apples, the F8 ROM was called the **Monitor ROM** because it contained the system monitor, which is a program that enables the user to communicate with the Apple on a very basic level. A RESET' in the old monitor ROM caused the 6502 to vector to $FF59 which is an entry point to the monitor. Thus, when the computer was turned on or RESET was pressed, the user found himself communicating with the Apple through the keyboard and television screen as controlled by the system monitor.

Today's Apples have a different F8 ROM called the **Autostart ROM**. It still has the system monitor in it, but some old capabilities were lost and some new ones were gained. The response to RESET' is completely different. The RESET vector in the Autostart ROM is $FA62. The operation of RESET' with the Autostart ROM is fully explained in the *Apple II Reference Manual*, but the basic points are given here. Most notably, the RESET' transfers program control to a RAM vector. This means that the ultimate response of the Apple's RESET key is controllable by software. At power up, the RAM RESET vector ($3F2 and $3F3) is set, and from that point, it may be set to any value by whatever program is controlling the Apple at a given moment. If the Apple has no disk drive, the RAM RESET vector is set to enter BASIC at power up. If there is a disk drive, the Apple enters the bootstrap routine contained in ROM on the disk controller. The RAM RESET vector is usually set by software loaded from the disk. The automatic startup of the disk is what gives the Autostart ROM its name.

Autostart ROM firmware only boots the disk on a RESET that occurs at power up. Other RESETs cause program flow to go to the address contained in the RAM RESET vector. The firmware uses a code at $3F4 to determine whether a given RESET' was initiated at power up or not. The code is never properly set at power up, but the power-up RESET sets the code so the following RESETs will not be "cold starts." The power-up byte ($3F4) must be the exclusive-OR between $A5 and the contents of $3F3, or a power-up RESET will be performed when RESET is pressed. Any program can scramble the power-up code and force a "cold start" when RESET is pressed.

There are advantages and disadvantages in the soft RESET vector. The chief advantage is that the user has a versatile, programmable reset function. The chief disadvantage is that the prime purpose of a microprocessor reset can be defeated by a program. There is no way for the user to reset his computer if a program doctors the RAM RESET vector for some other purpose. The Apple is set up so the user can be denied access to his own computer by accident or on purpose. Sometimes the Apple user is required to turn off the computer and turn it back on when it gets hung up. At turn on, electronic circuits are vulnerable to failure, so turning a computer off, then on, to achieve a reset is a chink in the armor.

NMI' and IRQ'

The NMI' and IRQ' lines are both connected only to the peripheral slots in the Apple. The IRQ' is the normal I/O interrupt signal because it can be enabled or disabled under program control. The idea of a non-maskable interrupt is to take action which has higher priority than any programming purpose. For example, an Apple may be required to take emergency action in the event of a failure in a manufacturing robot it is controlling. The non-maskable interrupt can also be used in monitoring the Apple operation from a remote panel or single instruction step execution of 6502 programs. These applications would, of course, require peripheral card designs.

The interrupt sequence is similar for either NMI' or IRQ'. The 6502 first completes execution of the current instruction. Then the following sequence occurs in the case of NMI' or IRQ' with interrupt requests enabled:

1. Program Counter is pushed on stack, high byte first.
2. Processor Status is pushed on stack with BREAK bit reset.
3. Contents of interrupt vector (NMI' = $FFFA-$FFFB; IRQ' = $FFFE-$FFFF) are fetched, low byte first.
4. Interrupt routine is begun with interrupt requests disabled.

There is a basic hardware difference between NMI' and IRQ' in the 6502. NMI' is **edge sensitive** like a clockpulse input, and IRQ' is **level sensitive.** A typical order of events with NMI' is:

1. The NMI' line drops low.
2. The NMI handling routine is executed with interrupt requests disabled.
3. The NMI' line is brought high.
4. Normal program flow is resumed with interrupt requests enabled or disabled as they were before the non-maskable interrupt occurred.

A second non-maskable interrupt will not interrupt the routine of the first as long as the NMI' line is held low. Thus while NMI' is not maskable by program control, it is hardware maskable in the sense that any interrupting device can prevent further interrupting by holding NMI' low. Recall that part of the NMI sequence is the disabling of interrupt requests, so the IRQ' cannot interrupt an NMI' handler unless the handler enables it.

A typical order of events with IRQ' is:

1. The IRQ' drops low.
2. The interrupt routine execution is begun with interrupt requests disabled.
3. The interrupt is acknowledged and IRQ' goes high.
4. The interrupt routine execution is completed and normal program flow is resumed with interrupt requests enabled.

Interrupt requests are disabled by the IRQ' sequence just as they are in the NMI' sequence. This prevents the still low IRQ' from immediately generating a second interrupt. The program maskable IRQ' can be used in any variety of implementation methods. The program must acknowledge and enable interrupts in a manner consistent with the protocol of the interrupting hardware. The point with IRQ' is to acknowledge the interrupt before enabling further interrupts so that multiple interrupts are not generated inadvertently. Interrupt acknowledges in the Apple usually consist of an access to one of the peripheral slot assigned addresses.

The enabling and disabling of IRQ' can be done fairly effortlessly in many applications. Either NMI' or IRQ' saves the Program Counter and processor Status Register on the stack before vectoring to the **interrupt handler.** The Status is saved before the interrupt disable bit of the Status Register is set. If, at the end of the interrupt handler, an RTI (ReTurn from Interrupt) instruction is executed, the Program Counter and Status Register are restored. Along with the rest of the Status Register, the pre-interrupt state of the interrupt disable bit is restored. Further interrupts are automatically disabled by the interrupt sequence and the disable/enable status is automatically restored by the RTI instruction. The other 6502 registers (Accumulator, X-register, Y-register, and Stack Pointer) are not automatically saved by interrupts. These must be saved and restored by the interrupt handler if the application demands it.

In some applications it would be desirable to enable interrupt handlers to be interrupted. This sort of processing is handled well by the stack architecture. Return link information for each interrupt is simply stacked over each other, possibly several interrupts deep. All of the interrupts are eventually fully serviced when the congestion is reduced.

Any peripheral card may interrupt the 6502 in the Apple. If there is a possibility of multiple interrupt sources, the 6502 needs to be able to distinguish among the interrupting devices. This can be done by polling. In polling, the interrupt handler checks each peripheral slot to see if it caused the interrupt. Each card in a polling system must be capable of responding to an address prompt by placing its interrupt status on the data bus (normally D7 of the data bus).

The peripheral slots have an **interrupt priority chain** which works exactly like the DMA priority chain. Card designs supporting the priority chain follow the same protocol as described in the section on DMA. As in other priority operations, Slot 0 has the highest priority and Slot 7 has the lowest priority. Cards in a priority chain control interrupts at lower priority cards and are controlled by higher priority cards. The priority chain does not eliminate the need for polling in a multiple interrupt source environment. Nor is the priority chain necessary to determine priority since this is determined implicitly by the order in which the interrupt handler polls the devices. Still, there are many conceivable uses for the priority chain. For example, a card may perform operations which will not tolerate interrupts from lower priority devices, but will tolerate interrupts from higher priority devices. Through the priority chain, system designs can be implemented to selectively enable high priority interrupts only.

There is a way in the Apple to determine priority of interrupts without any loss of time. This way would be to have the interrupting card contain its own IRQ vector. In the Apple, any peripheral card can disable motherboard ROM and steal ROM addresses. The interrupting card would only have to steal $FFFE and $FFFF to vector the Apple to its handler. This system could use the interrupt priority chain to prevent two cards from simultaneously responding to $FFFE or $FFFF.

The firmware implementation of NMI' and IRQ' handlers is very simple and nearly identical in the Monitor ROM and Autostart ROM. IRQ' and NMI' are handled the same in either ROM, but the addresses of the IRQ' firmware are different. An NMI' simply vectors straight to $3FB, where a JUMP instruction to the software NMI' handler must be stored.

The IRQ' is handled differently. A short firmware routine is executed which determines whether a BREAK instruction or an interrupt request is being processed. Both IRQ' and the BREAK instruction use $FFFE and $FFFF as their vector, and the IRQ' handler must distinguish between BREAK and an external interrupt request by checking the status that was pushed to the stack when the BREAK or IRQ' occurred. When it is determined that an external interrupt occurred, the program vectors to the contents of $3FE and $3FF. $3FE and $3FF should contain the address of the IRQ' handler.

In distinguishing between BREAK and IRQ', the Apple firmware saves the contents of the 6502 Accumulator at $45 and then modifies the Accumulator. The interrupted accumulator value must be retrieved from $45 if it is required for processing or restoration. Stacked interrupt applications requiring the saving of 6502 registers should save them on the stack. The accumulator value must be retrieved from $45 before pushing to the stack in the Apple.

The BREAK Instruction

The BREAK instruction is a software generated interrupt which is not disabled by the IRQ' disable bit of the Status Register. Its uses are not obvious, even to an experienced computer programmer who has not been exposed to it. Why would a program want to interrupt itself?

One use of BREAK is to make it the terminating instruction of 6502 programs rather than a RTS (ReTurn from Subroutine). The idea here is to have a program terminating routine which directs program flow to some sort of system utility. In this sense the BREAK is a programmable HALT instruction.

A second way of using BREAK is as a debugging breakpoint. When debugging or investigating software, it is often useful to stop a program at a specific address and to examine program progress. The BREAK instruction is a very convenient way of doing this. Instead of overwriting three bytes of code with a JUMP instruction, only one byte is overwritten by the BREAK instruction. The program counter and processor status are saved on the stack as with IRQ' and NMI' so a BREAK handler can be written to insert break points and resume flow after investigation.

A third use of BREAK is to allow out of control 6502 programs to bomb gracefully. A misdirected program tends to lead program flow to an address where no program has been stored. But the MPU doesn't know there is no program there. The 6502 is like a dog in heat; it will try to execute anything it finds on the data bus. This can be chaotic in any system but especially in a memory mapped I/O system like the Apple. Printers or disk drives can start operating when random addresses are accessed by the MPU. It happens that, at power up, much of RAM goes to a state of all zeroes. $00 is the op code of the BREAK instruction and, as a consequence, many bombed programs wind up executing a BREAK instruction. This is good, because the BREAK handler is usually designed to neatly terminate a program and enter a human communication utility. In this way the BREAK instruction redirects the indiscriminant 6502. It is an interrupt upon crash instruction. The response of the 6502 to a BREAK instruction consists of the following sequence:

1. Program Count +2 is pushed to the stack, high byte first.
2. Status Register is pushed to stack (BREAK bit set).
3. BREAK/IRQ' address is fetched from $FFFE and $FFFF, low byte first.
4. Program execution is begun at address contained in $FFFE and $FFFF with interrupt requests disabled.

The difference between the external interrupt request and the BREAK command is the BREAK flag, which is shown in 6502 literature as bit 4 of the Status Register. The BREAK flag is conceptually different from the other status flags, however. It is not tested by any 6502 instructions, and there is no set or clear instruction for the BREAK flag. It can only be checked after the Status Register has been placed on the stack. It is checked by pulling the Status value from the stack and checking bit 4. **Rather than a bit of the Status Register, the BREAK flag seems to be a characteristic of the way processor Status is pushed to the stack.*** The BREAK flag exists only in RAM after a push to the stack operation in accordance with the following rules:

1. PHP command sets bit 4 in RAM (no significance).
2. Push Status resulting from NMI' resets bit 4 in RAM (no significance).
3. Push Status resulting from IRQ' resets bit 4 in RAM (identifies IRQ').
4. Push Status resulting from BRK command sets bit 4 in RAM (identifies BREAK interrupt).

BREAK status is meaningful only in an IRQ' handler. It can be checked in an IRQ' handler with the following sequence:

```
PLA
PHA
AND #%00010000
BNE BREAK.HANDLER
BEQ CONTINUE.IRQ
```

The handling of BREAK commands is different between the Monitor ROM and the Autostart ROM. In both ROMs, the BREAK is first detected in the IRQ'/BREAK routine. Then the interrupted Status is restored, possibly enabling interrupt requests. Then all interrupted 6502 register states are stored in $3A, $3B, and $45 through $49. At this point in the old Monitor ROM, the interrupted register states are displayed, the instruction at interrupted Program Count +2 is disassembled and displayed, and the system monitor is entered. At the same point in the Autostart ROM, the program flow vectors to the contents of $3F0 and $3F1. The soft BREAK vector ($3F0 and $3F1) is loaded at power up with the address of the same BREAK routine that was contained in the old Monitor ROM. Therefore, the BREAK routines in the two ROMs perform identically after power up, but, from that point, the Autostart BREAK handler can be changed. The old monitor BREAK routine is fixed in ROM with no user programmable capability.

The Apple firmware routines are adequate for terminating programs and inserting debugging breakpoints. Program status saved by the BREAK handler is available for restart of flow via the G (GO) command of the monitor.

*The above concept of the BREAK flag is based strictly on my own experiments. In no literature was I able to find a satisfactory description of specifically when the BREAK flag is set and reset. The concept of BREAK status being stored in bit 4 of the Status Register simply does not fit the way I found BREAK status to be stored and checked. Inside the 6502, there may well be a bit of the Status Register which keeps track of BREAK status. In any case, the BREAK status can only be checked by retrieving it from RAM after a push Status to the stack operation.

Priority Among Interrupts

There are priority considerations among the interrupts which determine what happens when more than one interrupt occurs at the same time. The general priority of interrupts is as follows:

Highest	RESET'
	NMI'
	BREAK
Lowest	IRQ'

In the event of simultaneous interrupts, RESET' overrides all other processor actions. If NMI' drops low while RESET' is low, the processor will not respond to it. Once the RESET' routine has been entered, however, the processor can be interrupted by NMI' or BREAK. For this reason, it may be best for a card to disable its NMI' generating circuitry when RESET' occurs and leave it disabled until signaled by the 6502 that the RESET' routine is accomplished. The idea of RESET' is to reset the whole sytem, not just the 6502. All interrupts set the disable interrupt flag of the Status Register as part of their initial sequence. This disables external interrupt requests only (IRQ').

In the event of simultaneous NMI', BREAK, and IRQ' with IRQ' enabled, the processor would complete the BREAK instruction, fetching the contents of the IRQ'/BREAK vector and disabling interrupt requests. Then the NMI' sequence would occur. If the NMI' handler enabled interrupts and the IRQ' was still low, the IRQ' sequence would take place. The more likely case would be for the NMI' to be serviced with interrupts disabled. Following the RTI at the end of the NMI' handler the IRQ'/BREAKhandler would be executed with bit 4 of the top byte of the stack identifying the interrupt as a BREAK. In the Apple, the pre-BREAK Status is pulled from the stack as soon as a BREAK is identified. This would enable interrupt requests in our example and allow the IRQ' sequence to begin, assuming IRQ' was still low. Following the RTI instruction at the end of the IRQ' handler, the BREAK routine would be reentered and its course would be run.

SOFTWARE APPLICATION

6502 INSTRUCTION DETAILS

The state of the address bus and data bus on every cycle of operation are normally of no interest to the Apple programmer. However, there are non-obvious features of 6502 command execution which affect programming of I/O. This is a natural consequence of decoding I/O commands from the address bus. These address details are of particular interest to the assembly language programmer, but they affect some BASIC programs too.

Table 4.1 contains an example of every type of instruction sequence found in the 6502. It shows the state of the address bus and data bus for each cycle of execution. LDA, DEX, ASL, PHA, and PLA were chosen to represent classes of instructions whose execution sequences were identical. Table 4.2 is keyed to Table 4.1. To find an example of any instruction and address mode, look up the instruction in Table 4.2, then see the referenced example in Table 4.1.

The OP CODE of all instructions shown in Table 4.1 is assumed to reside at $1000. The X- and Y-registers both contain $20 in all examples. Y-indexed instructions are represented by X-indexed examples when Y-indexed execution is identical to X-indexed execution. When possible, LDA examples are used to represent storing instructions (STA, STX, STY), and in these examples the write cycles of storing instructions have a "w" following their address. Cycles that are always write cycles have a "W" following their address. The letters "PX" stand for Page Crossing. A few examples show the first cycle of the next instruction. This is indicated by "NEXT OP" on the data bus.

At times, the 6502 addresses parts of memory which have nothing to do with a given instruction. This occurs when the 6502 is performing an internal operation in a cycle and really doesn't need to address anything. Indexing or branching across page boundaries always results in a superfluous access to an address in the wrong page. It takes an extra cycle for the 6502 to increment or decrement the high portion of an address computed across a page boundary. A "LDA $5F72,X", for example, takes four cycles with no page crossing, and five cycles with a page crossing. STA instructions in which the possibility of a page crossing exists allow an extra cycle whether the page crossing occurs or not. The *Synertek Programming Manual* (May 1978) states that this is necessary to prevent a superfluous write to the wrong address.

There are other interesting points about 6502 addressing. The read-modify-write instructions (ASL, LSR, ROL, ROR, INC, DEC) always perform a double write to the valid address. The first write cycle writes the same data that was read, and the second write stores the modified data. Pulling data from the stack results in a superfluous access to a wrong Page 1 address. All superfluous accesses to wrong addresses are on read cycles, and the resulting data is ignored by the 6502.

Three software applications of 6502 addressing details are in the controlling of the serial outputs, the 16K RAM peripheral card, and the disk controller. The speaker and cassette are toggle outputs which are usually made to toggle up and down at an audio rate. The speaker, for example, should not normally be accessed by instructions which make a double or quadruple access to $C030, because that would result in the speaker line toggling back and forth at one megahertz. The idea is to toggle the speaker, wait a thousand microseconds or so, then toggle it again. Similar considerations exist for the C040 STROBE'. The programmer may select a single, double, triple, or quadruple strobe by utilizing one of the following instructions:

STA $C040	one Strobe
STA $C040,X (X=0)	two Strobes
ASL $C040	three Strobes
ASL $C040,X (X=0)	four Strobes

In BASIC, it helps to be aware of what machine language instruction actually performs the memory access when a PEEK or POKE instruction is executed. The following instructions perform the actual memory access in the Apple:

```
Applesoft PEEK - E76F:  LDA($50),Y   Y=0
Applesoft POKE - E781:  STA($50),Y   Y=0
Integer PEEK   - EEF9:  LDA($CE),Y   Y=0
Integer POKE   - EF0D:  STA($CE),Y   Y=0
```

Correlating the PEEK and POKE instructions with examples 26 and 28 of Table 4.1 indicates that POKE instructions generate a double access to the POKE'd address, and PEEK instructions generate a single access to the PEEK'd address. For this reason, speaker or cassette control from BASIC should be performed by PEEK instructions: "A = PEEK(-16336)" or "A = PEEK(-16352)." As for the

C040 STROBE'; "A = PEEK(-16320)" generates a single strobe, and "POKE-16320,0" generates a double strobe.

The way the 16K RAM card is controlled makes it a prime candidate for sneaky address bus manipulation. The operation of the 16K RAM card is covered fully in Chapter 5, but a small note about its operation belongs here. As described in Chapter 5, the RAM card is configured for writing by two successive reads to $C081, $C083, $C089, or $C08B (see Table 5.2). For this purpose, one instruction can accomplish the same as two. "ASL $C081,X" with X=0 performs the same task as "LDA $C081; LDA $C081." Read-modify-write, absolute indexed, no page crossing instructions generate two read accesses and two write accesses to the computed address. This is more cute than valuable, but it does illustrate the potential of controlling peripherals by single instruction address sequences in the Apple.

A more important application of knowledge of addressing detail can be seen at addresses $B82A through $B842 of the DOS 3.3 RWTS subroutine. $B82A is the beginning of the WRITE DATA routine which writes coded data to a sector of the disk. Direction of disk operations is accomplished on the disk controller by a logic state sequencer, which is a programmed hardware controller. Simply put, writing data to the disk consists of syncing the writing loop of the logic state sequencer to the writing loop of the controlling software. The following program steps check for write protect and resets the logic state sequencer to its idle location:

```
LDA $C08D,X   ;X = $60 if Slot 6.
LDA $C08E,X
BMI WPROTECT  ;Branch if disk
              ;write protected.
```

The program will fall through the branch if the disk is not write protected. From this induced idle state, the software can sync itself to the logic sequencer with the statement, "STA $C08F,X". This instruction performs a double access to $C0EF (assuming Slot 6). The first access is decoded in the disk controller to cause the logic state sequencer to leave its idle state and begin its write loop. The second access stores actual disk write data in the controller's input/output register. The controller will only accept data on the clockpulse after the one which started the logic state sequencer and on every fourth clockpulse afterward. The writing technique involves writing data in software loops that take exact multiples of four cycles to execute.

Persons wishing to imitate the writing technique of the RWTS subroutine should not substitute a "STA $C0EF" instruction for the "STA $C08F,X" at address $B83F of DOS 3.3. "STA $C0EF" will start up the software loop one clockpulse out of sync with the logic state sequencer and the controller won't accept the write data. "STA $C0EF,X" will work with 0 in the X-register. The instruction must make a double access to $C0EF. Another address mode of instruction which will work is a STA (ZP),Y with no page crossing.

No doubt, the Apple controller's logic state sequencer was designed around the "STA $C080,X" instruction, since this makes it possible to have the disk in other slots besides Slot 6. Given the hardware, Apple disk programmers must understand addressing details to program the disk on this level.

Table 4.1 6502 Instructions.

	1	2	3	4	5	6	7
1. DEX	$1000 $CA	$1001 IGNORE					
2. ASL A	$1000 $0A	$1001 IGNORE					
3. PHA	$1000 $48	$1001 IGNORE	SPNT W DATA				
4. PLA	$1000 $68	$1001 IGNORE	SPNT IGNORE	SPNT+1 DATA			
5. RTS	$1000 $60	$1001 IGNORE	SPNT IGNORE	SPNT+1 PCL	SPNT+2 PCH	PCH,PCL IGNORE	PCH,PCL+1 NEXT OP
6. RTI	$1000 $40	$1001 IGNORE	SPNT IGNORE	SPNT+1 STATUS	SPNT+2 PCL	SPNT+3 PCH	PCH,PCL NEXT OP
7. BRK	$1000 $00	$1001 INGORE	SPNT W $10	SPNT-1 W $02	SPNT-2 W STATUS	$FFFE IRQLO	$FFFF IRQHI
8. BEQ $10 (Z=0)	$1000 $F0	$1001 $10	$1002 NEXT OP				
9. BEQ $10 (Z=1)	$1000 $F0	$1001 $10	$1002 IGNORE	$1012 NEXT OP			
10. BEQ $F3 (Z=1) (PX)	$1000 $F0	$1001 $F3	$1002 IGNORE	$10F5 IGNORE	$FF5 NEXT OP		
11. LDA #$AA	$1000 $A9	$1001 $AA					
12. LDA $70 STA $70	$1000 $A5	$1001 $70	$0070 w DATA				
13. ASL $70	$1000 $06	$1001 $70	$0070 OLD DATA	$0070 W OLD DATA	$0070 W NEW DATA		
14. LDA $70,X STA $70,X	$1000 $B5	$1001 $70	$0070 IGNORE	$0090 w DATA			
15. ASL $70,X	$1000 $16	$1001 $70	$0070 IGNORE	$0090 OLD DATA	$0090 W OLD DATA	$0090 W NEW DATA	
16. LDA $5F72 STA $5F72	$1000 $AD	$1001 $72	$1002 $5F	$5F72 w DATA			
17. ASL $5F72	$1000 $0E	$1001 $72	$1002 $5F	$5F72 OLD DATA	$5F72 W OLD DATA	$5F72 W NEW DATA	
18. JMP $5F72	$1000 $4C	$1001 $72	$1002 $5F	$5F72 NEXT OP			
19. JSR $5F72	$1000 $20	$1001 $72	SPNT W IGNORE	SPNT W $10	SPNT-1 W $02	$1002 $5F	$5F72 NEXT OP
20. LDA $5F72,X (NO PX)	$1000 $BD	$1001 $72	$1002 $5F	$5F92 DATA			
21. LDA $5FF2,X STA $5FF2,X (PX)	$1000 $BD	$1001 $F2	$1002 $5F	$5F12 IGNORE	$6012 w DATA		
22. STA $5F72,X (NO PX)	$1000 $9D	$1001 $72	$1002 $5F	$5F92 IGNORE	$5F92 W DATA		
23. ASL $5F72,X (NO PX)	$1000 $1E	$1001 $72	$1002 $5F	$5792 IGNORE	$5792 OLD DATA	$5792 W OLD DATA	$5792 W NEW DATA
24. ASL $5FF2,X (PX)	$1000 $1E	$1001 $F2	$1002 $5F	$5F12 IGNORE	$6012 OLD DATA	$6012 W OLD DATA	$6012 W NEW DATA
25. LDA ($70,X) STA ($70,X)	$1000 $A1	$1001 $70	$0070 IGNORE	$0090 ADL	$0091 ADH	ADH,ADL w DATA	
26. LDA ($70),Y (NO PX)	$1000 $B1	$1001 $70	$0070 $72	$0071 $57	$5792 DATA		
27. LDA ($70),Y STA ($70),Y (PX)	$1000 $B1	$1001 $70	$0070 $F2	$0071 $57	$5712 IGNORE	$6012 w DATA	
28. STA ($70),Y (NO PX)	$1000 $91	$1001 $70	$0070 $72	$0071 $57	$5792 IGNORE	$5792 W DATA	
29. JMP ($5F72)	$1000 $6C	$1001 $72	$1002 $5F	$5F72 PCL	$5F73 PCH	PCH,PCL NEXT OP	

ADDR. BUS
DATA BUS

W - WRITE CYCLE
w - WRITE CYCLE IF STORING INSTRUCTION
PX - PAGE CROSSING
NEXT OP - OP CODE NEXT INSTRUCTION
X-REG = $20; Y-REG = $20.
Location $0070 contains $5772 or $57F2 as needed for illustration.

Table 4.2 6502 Instruction Cross Reference.

	IMP	REL	IMM	ACC	ØPG	ØPG X	ØPG Y	ABS	ABS X	ABS Y	IND	IND X	IND Y
ADC AND CMP EOR LDA ORA SBC			11		12	14		16	20 21	20 21		25	26 27
ASL LSR ROL ROR				2	13	15		17	23 24				
BCC BCS BEQ BMI BNE BPL BVC BVS		8,9 10											
CLC CLD CLI CLV DEX DEY INX INY NOP SEC SED SEI TAX TAY TSX TXA TXS TYA	1												
BIT					12			16					
BRK	7												
CPX CPY			11		12			16					
DEC INC					13	15		17	23 24				
JMP								18			29		
JSR								19					
LDX			11		12		14	16		20 21			
LDY			11		12	14		16	20 21				
PHA PHP	3												
PLA PLP	4												
RTI	6												
RTS	5												
STA					12	14		16	21 22	21 22		25	27 28
STX					12		14	16					
STY					12	14		16					

HARDWARE APPLICATION

D MANUAL CONTROLLER

How many times have you been working with your Apple and had to look up control addresses to select HIRES NO MIX or LORES MIX or any other screen mode? It's too bad, but screen mode selection isn't supported in the Apple firmware by escape codes or similarly easy interface. The Apple scheme of controlling operational features via soft switches is extremely effective for control by programs, but the operator at the keyboard is left without means of direct control, unless the operating program supports it. This Application Note describes a simple DMA controller which allows the operator to override program control and manually select among the Apple features, including screen modes. I call this circuit D MAnual Controller. Not everybody likes dis name, but dat's not my problem!

Figure 4.8 is a schematic of D MAnual Controller. It works by stealing a single cycle from the 6502 and placing an address in the $C0XX range on the address bus. This action is initiated when the operator presses one of eight pushbuttons (or four momentary on-off-on switches). Six slide switches (or six DIP switches) configure the controller so the pushbuttons will affect different Apple features—screen modes, annunciators, disk drives, memory configuration, etc. The concept is to place the operator switches on a small remote panel, connected by a 16 wire ribbon cable to the cycle stealing peripheral card. Figure 4.9 is a photo of an earlier prototype which controlled screen modes only.

D MAnual Controller can control some peripheral card functions as well as motherboard features. Those peripheral card functions which can be controlled are the ones normally programmed using DEVICE SELECT' addresses such as RAM card, firmware card, and disk controller management. Tables 4.3 and 4.4 are an operational summary of D MAnual Controller showing how some features are controlled. Some of these are only educational or cute, while others, like screen mode control and memory bank switching, can be very useful. It is recommended that the configuration switches be left in the position in which you will most often need them, so you will have convenient manual control of the features which are important to you.

Even though Table 4.3 shows how to control Slot 6 disk drives using D MAnual Controller, it doesn't follow that disk I/O can be performed manually. D MAnual Controller is not capable of transferring data via the data bus. It can only turn the drives on and off, select between drives, configure the disk controller for different functions, and position the head. Please take note that turning a drive on and setting READ/WRITE to WRITE will clobber the data on a disk which is not write protected. It is suggested that you experiment with no disk or an unimportant disk in the drive. Manual control of the disk drive is educational, but its only practical function would be to assist in the development of advanced disk programs and formats, or to aid maintenance technicians and disk hardware developers. Incidentally, to step the head, turn the phases on and off sequentially while a drive is rotating. Stepping through the phases in ascending order moves the head toward track 34. Stepping in descending order moves the head toward track 0.

D MAnual Controller is based in hardware and overrides program control. You can select features at any time, no matter what software or firmware is running. This can be very convenient for programmers while they are developing programs. The Controller does not lock out program control, though, so programs which repeatedly select a given mode will not appear to be affected when the Controller deselects that mode. It would be possible for a more ambitious design to lock out program control of selected Apple features via the USER1 line.

Circuit Operation

The heart of D MAnual Controller is a 74LS148 priority encoder which detects a button push and converts it to a 3-bit address. This address is latched in a 74LS374 when a button is pressed and placed on A2, A1, and A0 of the address bus at the first opportunity. The state of A7-A3 of the address bus and R/W' during the DMA cycle are determined directly by the six configuration switches. A15-A8 are always set to 11000000 during the DMA cycle, yielding an address in the $C0XX range, the critical control range of the Apple.

Pressing any of the pushbuttons causes the signal at pin 14 of the LS148 to go low. This signal is debounced and inverted and sent to a 74LS195 shift register for single cycle generation. If the DMA priority input is low, the shift register will shift the button press signal through, and a one cycle negative signal will be felt at pin 2 of a 74LS74 flip-flop. The LS195 is clocked by PHASE 1 rising, so this one cycle signal falls and rises just after PHASE 1 rises.

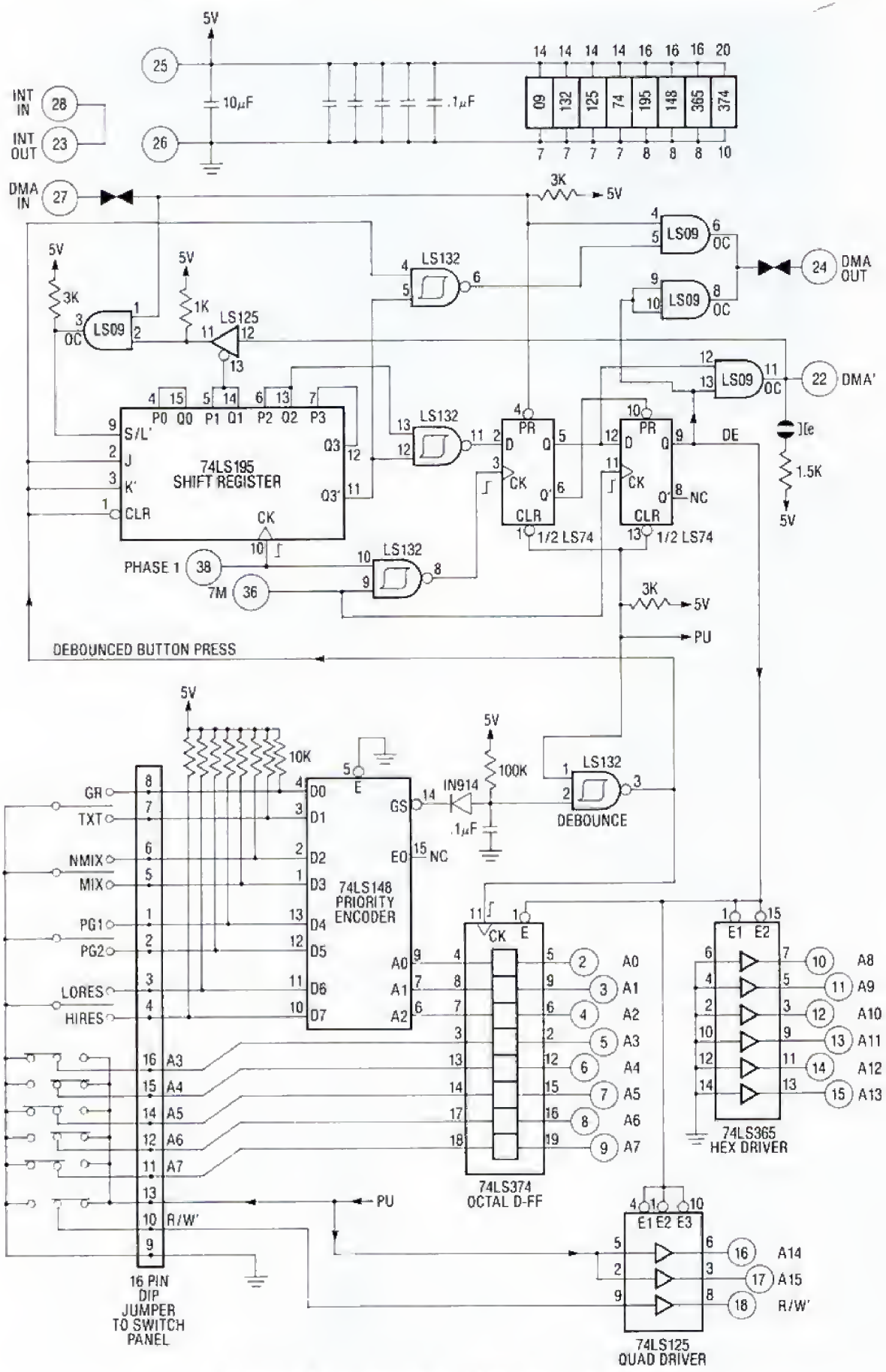


Figure 4.8 Schematic: D MAnual Controller.

The cycle is further delayed for one half a 7M period in the first half of the 74LS74. The resulting signal at pin 5 of the LS74 represents the DMA cycle.

When the DMA cycle signal at pin 5 of the 74LS74 goes low, the DMA' line is brought low. Half a 7M period later, the data enable signal (LS74-9) of D MAnual Controller's address bus drivers drops low. This delay allows the MPU to be isolated from the address bus before any attempt is made to control the address bus. At the end of the DMA cycle, the opposite order is observed. Because of propagation delays in the LS09 feeding the DMA' line and motherboard IC C11, control of the address bus is released before the MPU address bus driver is enabled.

D MAnual Controller supports the DMA priority chain, so it can operate with some other peripherals which perform DMA. It respects the DMA priority input and will delay its access until a higher priority device has finished its DMA. It also will steal cycles from a lower priority DMA card if that card respects its priority input.

Supporting the DMA priority chain is a little difficult, because it is the most abused protocol since "Do unto others..." Apple abused it by not publishing a protocol and by using the DMA priority chain in the firmware card. Microsoft abused it in their Z80 *Softcard* by requiring higher priority devices to wait several cycles after bringing the priority line low before the *Softcard* will get off the bus. Yet when the *Softcard* takes over the bus itself, it gives lower priority DMA cards no similar consideration. Some DMA based MPU cards don't support the DMA priority chain at all. Other DMA cards support or ignore the DMA priority chain in ways which make sharing of the DMA capability unpredictable.

D MAnual Controller gets around the unpredictability of other DMA card designs by monitoring the DMA' line and delaying its own DMA cycle if DMA' is being held low by another card. This should work with other DMA designs which make any attempt to support the priority chain. Here are some ways to install D MAnual Controller with other DMA related cards:

1. **Z80 Softcard.** Install D MAnual Controller in a higher priority slot. It will steal a cycle from the *Softcard* without affecting its operation. Switch S2 on the *Softcard* must be on for this configuration to work. Solder the IIe jumper on D MAnual controller if operating in an Apple IIe.* Other MPU cards which support the DMA priority chain should work in this configuration.
2. **Firmware Card.** The firmware card uses the DMA priority line even though it does not perform DMA. Since D MAnual Controller only needs a single cycle, it will work with a firmware card enabled in a higher priority slot. It just waits until the MPU accesses a non-ROM address, then steals a cycle. If a secondary MPU card like the *Softcard* happens to be in a lower priority slot, the firmware card can interfere

*In the Apple IIe, 3000 ohm pull-up resistors are used on the wire-OR lines instead of 1000 ohm resistors. This results in a switching time which is too slow for stealing cycles from the *Softcard*. Soldering the IIe jumper will speed switching time by paralleling the 3000 ohm motherboard resistor with a 1500 ohm resistor. If other DMA card designers begin to add this 1500 ohm resistor, the IIe jumper should only be connected on one of the cards. Incidentally, D MAnual Controller is a good candidate for installation in Slot 3 of the Apple IIe, because it will work in Slot 3, even though an 80 column card is plugged into the auxiliary slot.

Table 4.3 Operation of Soft Switches from D MAnual Controller.

RW	AAAA 7654	A 3	FUNCTION	BUTTON 0/1	BUTTON 2/3	BUTTON 4/5	BUTTON 6/7
W	0000	0	IIe MEMORY MANAGE	80STORE	RAMRD	RAMWRT	SLOTCXROM
W	0000	1	IIe MEMORY MANAGE	ALTZP	SLOTC3ROM	80COL	ALTCHARSET
X	0010	X	CASSETE OUT TOGGLE	----------PUSH ANY BUTTON----------			
X	0011	X	SPEAKER TOGGLE	----------PUSH ANY BUTTON----------			
X	0100	X	C040 STROBE'	----------PUSH ANY BUTTON----------			
X	0101	0	SCREEN MODE CTRL	GR/TXT	NMIX/MX	PG2/PG1	LORES/HIRES
X	0101	1	ANNUNCIATOR CTRL	AN0	AN1	AN2	AN3
X	0101	1	IIe 560 POINT MODE	--------	---------	----------	ENA/DSBL
X	1000	X	FIRMWARE CARD CTRL	ENA/DSBL	ENA/DSBL	ENA/DSBL	ENA/DSBL
X	1110	0	DISK HEAD CONTROL	PHASE-0	PHASE-1	PHASE-2	PHASE-3
X	1110	1	DISK CONTROL	OFF/ON	DRIVE 1/2	SHIFT/LOAD	READ/WRITE

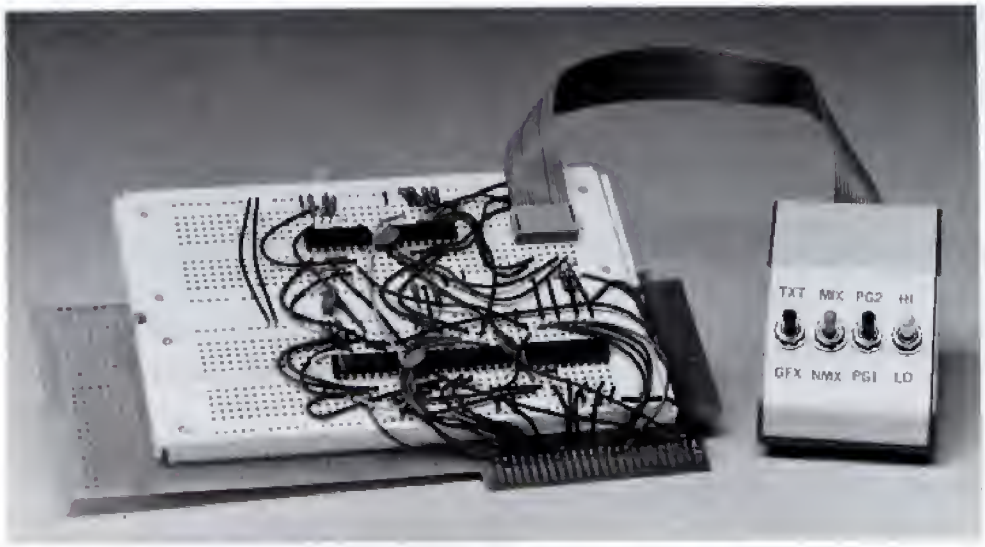


Figure 4.9 A Screen Mode Controller.

with that card's operation. This can be prevented by opening the DMA IN jumper on D MAnual Controller. Lower priority firmware cards are not interfered with by D MAnual Controller, because the Controller does not generate addresses in the firmware card range.

3. **Disk or Cassette I/O.** Disk and cassette I/O in the Apple are normally performed in precise timing loops. Any DMA device which is activated in the midst of such loops will interfere with them and the associated data transfer. Therefore, you should never operate a pushbutton of D MAnual Controller while cassette or disk I/O is being performed, especially during write operations. Some hard disk or eight inch floppy disk Apple interfaces are DMA based. Any such devices should probably be mounted in a higher priority slot than D MAnual Controller, so that if they support the DMA priority chain, the integrity of disk data transfer will be insured.
4. **DMA cards which do not support the priority chain.** Cards like these are like citizens who do not meet their responsibilities to society. Do not operate the pushbuttons of D MAnual Controller when such cards are active. Only one card at a time can perform DMA in the Apple.

Readers who wish to are encouraged to build D MAnual Controller for their own use. They may also purchase the Controller, assembled and tested. The Controller is being manufactured by the Southern California Research Group of Goleta, California. Readers of this book may order D MAnual Controller by contacting:

D MAnual Controller
Southern California Research Group
Post Office Box 2231-U
Goleta CA 93118
(805) 685-1931 for information
(800) 821-0774 In California, for orders only
(800) 635-8310 Outside California, for orders only

Table 4.4 Selection of 16K RAM Card or IIe Bank Switched RAM from D MAnual Controller.

RW	AAAA 7654	A 3	FUNCTION	BUTTON 0 OR 4	BUTTON 1 OR 5	BUTTON 2 OR 6	BUTTON 3 OR 7
R	1000	0	BANK 2 CTRL	READON - WRTOFF WRTCOUNT=0	READOFF WRTCOUNT+1	READOFF - WRTOFF WRTCOUNT=0	READON WRTCOUNT+1
W	1000	0	BANK 2 CTRL	READON - WRTOFF WRTCOUNT=0	READOFF WRTCOUNT=0	READOFF - WRTOFF WRTCOUNT=0	READON WRTCOUNT=0
R	1000	1	BANK 1 CTRL	READON - WRTOFF WRTCOUNT=0	READOFF WRTCOUNT+1	READOFF - WRTOFF WRTCOUNT=0	READON WRTCOUNT+1
W	1000	1	BANK 1 CTRL	READON - WRTOFF WRTCOUNT=0	READOFF WRTCOUNT=0	READOFF - WRTOFF WRTCOUNT=0	READON WRTCOUNT=0

HARDWARE APPLICATION

AN NMI' BASED SINGLE STEPPER

There is an inherent ability in the 6502 to interrupt execution of every instruction. The non-maskable interrupt is not used for anything in most Apples and is available for any sort of high priority control of the computer. An NMI' handled by firmware in Slot 0 can be made to override any operating software and give control of the Apple to its owner.

Figure 4.10 is the schematic of a very simple circuit which uses the high priority features of the non-maskable interrupt to achieve a hardware based HALT/STEP/TRACE capability. The operational philosophy of this NMI STEPPER is to interrupt every instruction when the RUN/HALT switch is in the HALT position. The interrupt is serviced in a

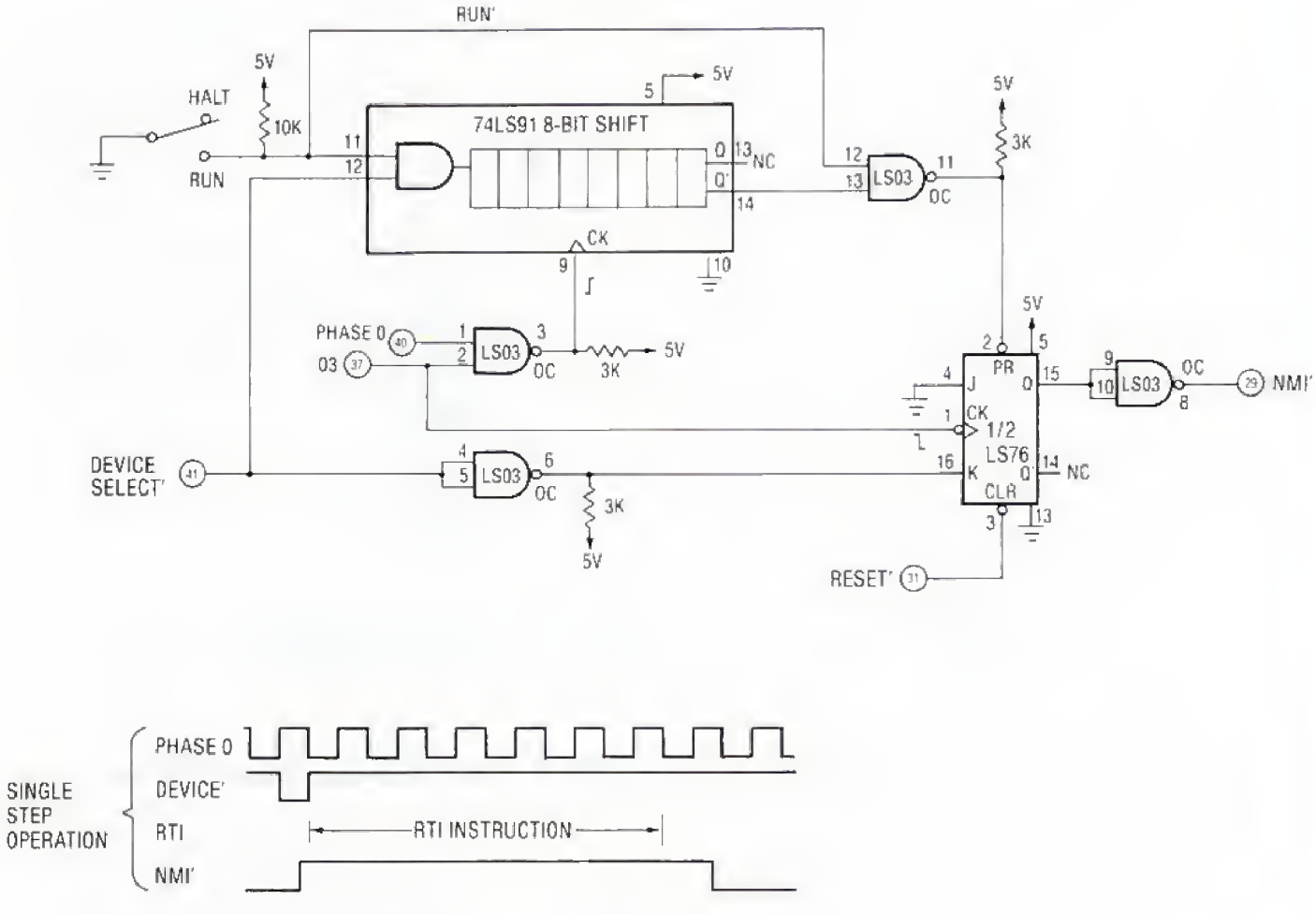


Figure 4.10 Schematic: An NMI Based Single Stepper.



Figure 4.11 The Author's Front Panel.

utility designed for the STEPPER, then, as the control is passed back to the interrupted program, circuitry is triggered to interrupt the very next instruction. This controlled generation of NMI' is based on the six cycle RTI instruction. The NMI' line is made to go high by a DEVICE SELECT' gate at the pertinent peripheral slot. If the HALT/RUN switch is still in the HALT position, the NMI' will drop low again after seven cycles. This gives just enough time to execute an RTI so that an interrupt is generated during the next instruction of normal flow.

Circuit operation is very simple. An 8-bit shift register generates the seven cycle delay before re-interrupt. The interrupt flip-flop is reset by RESET' or by a DEVICE SELECT'. The interrupt flip-flop is set by switching from RUN to HALT or on the eighth cycle after a DEVICE SELECT'. The J-input to the interrupt flip-flop is available for interrupt sources other than the HALT/RUN switch.

The NMI STEPPER may be controlled by software using Apple's $3FB - $3FD jump to NMI vector. The user could get the NMI STEPPER software up and available as part of his HELLO program on his disk. The STEPPER is more valuable however if the handling program resides in firmware. This way, the Apple user always has a programmed halt switch at his fingertips. This is achieved by using a custom F8 EPROM instead of the Autostart ROM or Monitor ROM. Here are some possible schemes for substituting EPROM for the F8 ROM:

1. Modify the firmware card in Slot 0 so the F8 socket accepts EPROM and is independently enabled by NMI'.
2. Replace cassette READ and WRITE routines of the F8 ROM with user routines to handle NMI'. EPROM must be installed on the motherboard using an adapter to make it pin-compatible with the Apple's ROM sockets. Such an adapter is described in an Application Note in Chapter 6 (see Figure 6.5).
3. Include an F8 EPROM as part of the STEPPER design. This EPROM is enabled by F8 addressing while NMI' is low. The STEPPER resides in Slot 0 and echoes $C08X commands via DMA to a RAM card or ROM card in a separate slot.
4. Include the STEPPER as a low cost add on to a Slot 0 ROM or RAM card.

The author utilizes the NMI STEPPER as part of a front panel for the Apple (see Figure 4.11). The front panel firmware resides in F8 of a modified firmware card in Slot 0. The F8 EPROM is selected by the NMI' line being low and is simply a merge between important Apple monitor routines and front panel utility routines. The NMI handler displays critical program flow (identically to an Apple BREAK) and steps, traces, or branches to the monitor as controlled by pushbuttons on the Apple game paddles. Using paddle buttons prevents the stepping process from interfering with keyboard inputs. The programming is quite simple, and maximum use is made of Apple monitor routines. Figure 4.12 is a listing showing the basics of STEPPER control.

Figure 4.13 shows a design concept which considerably enhances the NMI STEPPER. Figure 4.13 is a hardware breakpoint generator. Sixteen switches on a remote panel make up a breakpoint register.

```
SOURCE FILE: FIGURE 4.12
0000:              1 ********************************************************
0000:              2 *                                                      *
0000:              3 *                                                      *
0000:              4 *                NMI STEPPER ROUTINES                  *
0000:              5 *                                                      *
0000:              6 *                   BY JIM SATHER                      *
0000:              7 *                      3/28/83                         *
0000:              8 *                                                      *
0000:              9 *                                                      *
0000:             10 ********************************************************
0000:             11 *
0000:             12 *
0036:             13 CSWL     EQU  $36
0037:             14 CSWH     EQU  $37
003A:             15 PCL      EQU  $3A
003B:             16 PCH      EQU  $3B
0045:             17 ACC      EQU  $45
0046:             18 XREG     EQU  $46
0047:             19 YREG     EQU  $47
0048:             20 STATUS   EQU  $48
0049:             21 SPNT     EQU  $49
03F8:             22 CYVCTR   EQU  $3F8
03FB:             23 NVCTR    EQU  $3FB
C061:             24 BUTTON0  EQU  $C061
C062:             25 BUTTON1  EQU  $C062
C0C0:             26 NRESET   EQU  $C0C0      NMI STEPPER IN SLOT 4
F8D0:             27 INSDSP   EQU  $F8D0
FAD7:             28 RGDSP    EQU  $FAD7
FCA8:             29 WAIT     EQU  $FCA8
FD8B:             30 CROUT    EQU  $FD8B
FF3A:             31 BELL     EQU  $FF3A
FF69:             32 MONITOR  EQU  $FF69
0000:             33 *
0000:             34 *
----- NEXT OBJECT FILE NAME IS FIGURE 4.12.OBJ0
1F00:             35          ORG  $1F00
1F00:             36 *
1F00:             37 *
1F00:A9 4C        38 INIT     LDA  #$4C       INIT SETS NMI VECTOR.
1F02:8D FB 03     39          STA  NVCTR
1F05:A9 10        40          LDA  #>NMI
1F07:8D FC 03     41          STA  NVCTR+1
1F0A:A9 1F        42          LDA  #<NMI
1F0C:8D FD 03     43          STA  NVCTR+2
1F0F:60           44          RTS
1F10:             45 *
1F10:             46 *
1F10:85 45        47 NMI      STA  ACC        NMI IS THE ENTRY FOR THE
1F12:86 46        48          STX  XREG       HARDWARE STEP ROUTINE.
1F14:84 47        49          STY  YREG
1F16:68           50          PLA             SAVE 6502 REGISTERS AT
1F17:85 48        51          STA  STATUS     MONITOR SAVE LOCATIONS.
1F19:68           52          PLA             DON'T USE MONITOR SAVE ROUTINE
1F1A:85 3A        53          STA  PCL        BECAUSE IT CLEARS DECIMAL MODE.
1F1C:68           54          PLA
1F1D:85 3B        55          STA  PCH
1F1F:BA           56          TSX
```

Figure 4.12 Assembler Listing: NMI Stepper Routines. (1 of 2)

```
1F20:86 49      58          STX  SPNT
1F22:A5 36      59          LDA  CSWL       SAVE CSW ON STACK.
1F24:48         60          PHA
1F25:A5 37      61          LDA  CSWH
1F27:48         62          PHA
1F28:A9 F0      63          LDA  #$F0       USE SCREEN FOR DISPLAY.
1F2A:85 36      64          STA  CSWL
1F2C:A9 FD      65          LDA  #$FD
1F2E:85 37      66          STA  CSWH
1F30:20 D7 FA   67          JSR  RGDSP      DISPLAY REGISTERS AND INSTRUCTION.
1F33:20 8B FD   68          JSR  CROUT
1F36:20 D0 F8   69          JSR  INSDSP
1F39:           70 *
1F39:           71 *
1F39:AD 61 C0   72          LDA  BUTTON0    BUTTON0 ENTRY TO NMI CAUSES TRACE;
1F3C:30 2B      73          BMI  RESTORE
1F3E:48         74          PHA             ELSE, STOP AND WAIT FOR BUTTON.
1F3F:48         75          PHA
1F40:68         76 BEEP2    PLA
1F41:68         77          PLA
1F42:20 3A FF   78          JSR  BELL
1F45:20 3A FF   79          JSR  BELL
1F48:AD 61 C0   80 STOP     LDA  BUTTON0    BUTTON 0 CAUSES STEPPING.
1F4B:30 17      81          BMI  DBOUNCE
1F4D:AD 62 C0   82          LDA  BUTTON1
1F50:10 F6      83          BPL  STOP
1F52:           84 *
1F52:           85 *
1F52:A9 4C      86          LDA  #$4C       BUTTON 1: GO TO MONITOR.
1F54:8D F8 03   87          STA  CYVCTR     FIX SO CONTROL-Y RETURNS TO BEEP2.
1F57:A9 40      88          LDA  #>BEEP2
1F59:8D F9 03   89          STA  CYVCTR+1
1F5C:A9 1F      90          LDA  #<BEEP2
1F5E:8D FA 03   91          STA  CYVCTR+2
1F61:4C 69 FF   92          JMP  MONITOR
1F64:A9 00      93 DBOUNCE  LDA  #0
1F66:20 A8 FC   94          JSR  WAIT
1F69:           95 *
1F69:           96 *
1F69:68         97 RESTORE  PLA             RESTORE CSW.
1F6A:85 37      98          STA  CSWH
1F6C:68         99          PLA
1F6D:85 36     100          STA  CSWL
1F6F:A6 49     101          LDX  SPNT       RESTORE 6502 REGISTERS.
1F71:9A        102          TXS
1F72:A5 3B     103          LDA  PCH
1F74:48        104          PHA
1F75:A5 3A     105          LDA  PCL
1F77:48        106          PHA
1F78:A5 48     107          LDA  STATUS
1F7A:48        108          PHA
1F7B:A5 45     109          LDA  ACC
1F7D:A6 46     110          LDX  XREG
1F7F:A4 47     111          LDY  YREG
1F81:8D C0 C0  112          STA  NRESET     RESET NMI FLIP-FLOP.
1F84:40        113          RTI

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 4.12 Assembler Listing: NMI Stepper Routines. (2 of 2)

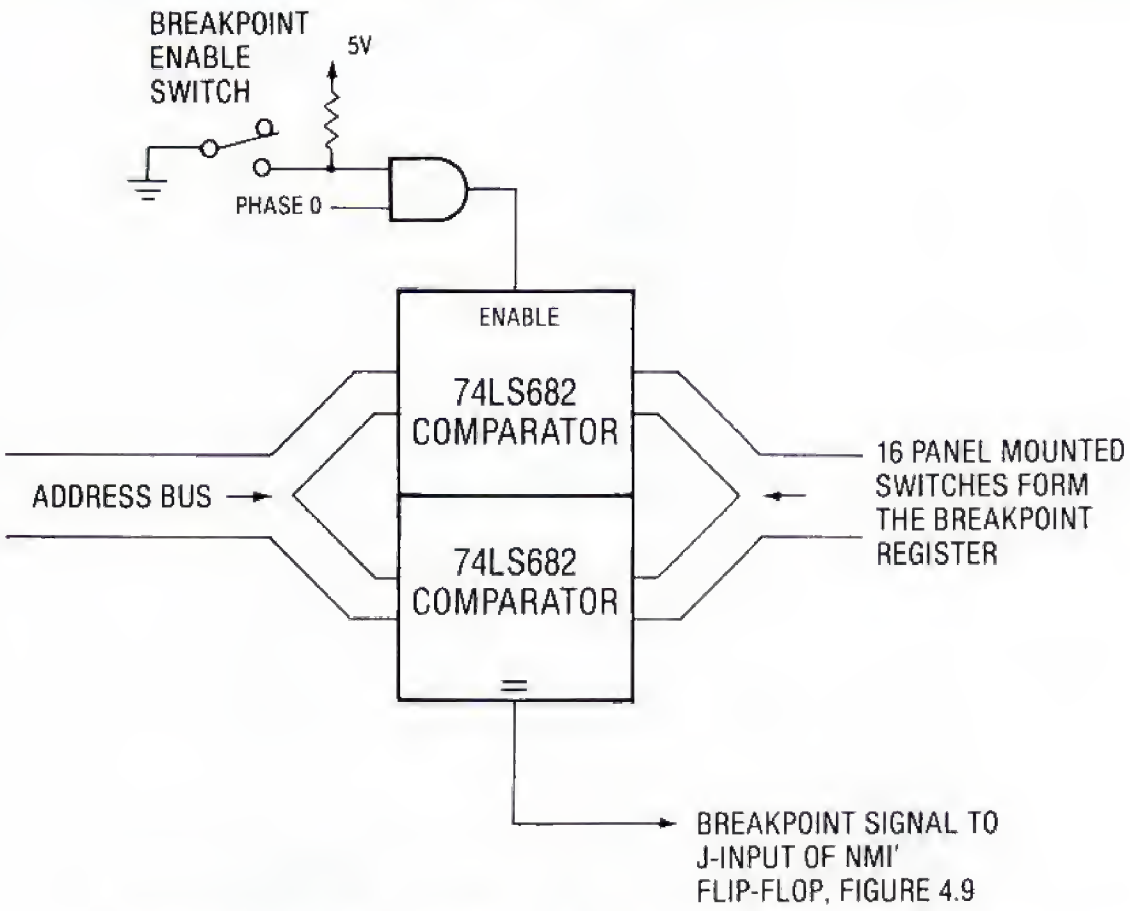
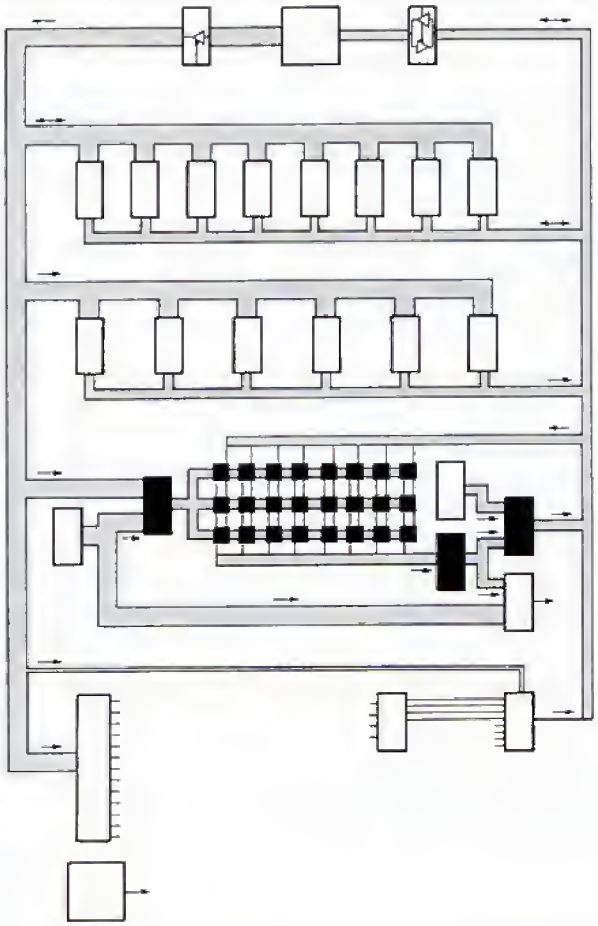


Figure 4.13 Design Concept for a Hardware Breakpoint Generator.

Any address can be placed in the breakpoint register. An NMI' is generated any time the address bus is equal to the breakpoint register. From that point, you can use the NMI STEPPER to step through the program under scrutiny. The breakpoint detection can be gated by R/W' high, R/W' low or 6502 SYNC if SYNC is jumpered from the 6502 to the NMI' generating card. The NMI STEPPER is a powerful investigative tool, alone or in combination with a breakpoint register.



chapter 5

RAM in the Apple II

One might think that RAM and its associated circuitry should be a relatively easy subject. You write to it and read from it. What else is there?

Well, the **MPU** does write to and read from memory, but the **video scanner** reads from memory too. And then there is the 16K dynamic RAM chip with its ROW address, COLUMN address, and refresh requirement. As was seen back in the chapter on bus structure, this all adds up to a lot of circuitry and complexity. In this chapter, we will examine the requirements of 16K dynamic RAM and how they are met in the Apple.

It should be mentioned here that earlier Apples could accommodate either 16K or 4K RAM chips. This was a throwback to the bad old days when 16K chips were a hot new expensive item. Today, one can fully populate an Apple with 48 Kilobytes of RAM for well under $50. In the Apple, 4K chips are obsolete, and there is no effort made in these pages to document operation with 4K chips. It is assumed that the Apple contains 48 Kilobytes of RAM.

THE 16K DYNAMIC RAM CHIP

16K RAM chips are 16,834 **bit** read/write memories. As is indicated in the bus structure diagram in the back of the book, it takes three groups of eight chips to make up the 49,152 bytes of read/write memory in the Apple. Figure 5.2 shows the pin assignments of the 16K RAM chips. This standard chip is available from a number of manufacturers in a variety of speeds. With a 2 MHz access rate, the Apple does not put a particularly stringent speed requirement on its RAM.

The RAM chip is a 16-pin device requiring four power supply inputs: +12V, +5V, -5V, and ground. There is a data input to accept write data, a tri-state data output to transfer read data, and a R/W' control input to identify read and write cycles.

It takes 14 bits to address 16K of RAM, but there are only seven address inputs to the RAM chip. The 14-bit address must be multiplexed on to the 7-bit RAM address input lines in the form of a **ROW address** followed by a **COLUMN address**. Think of

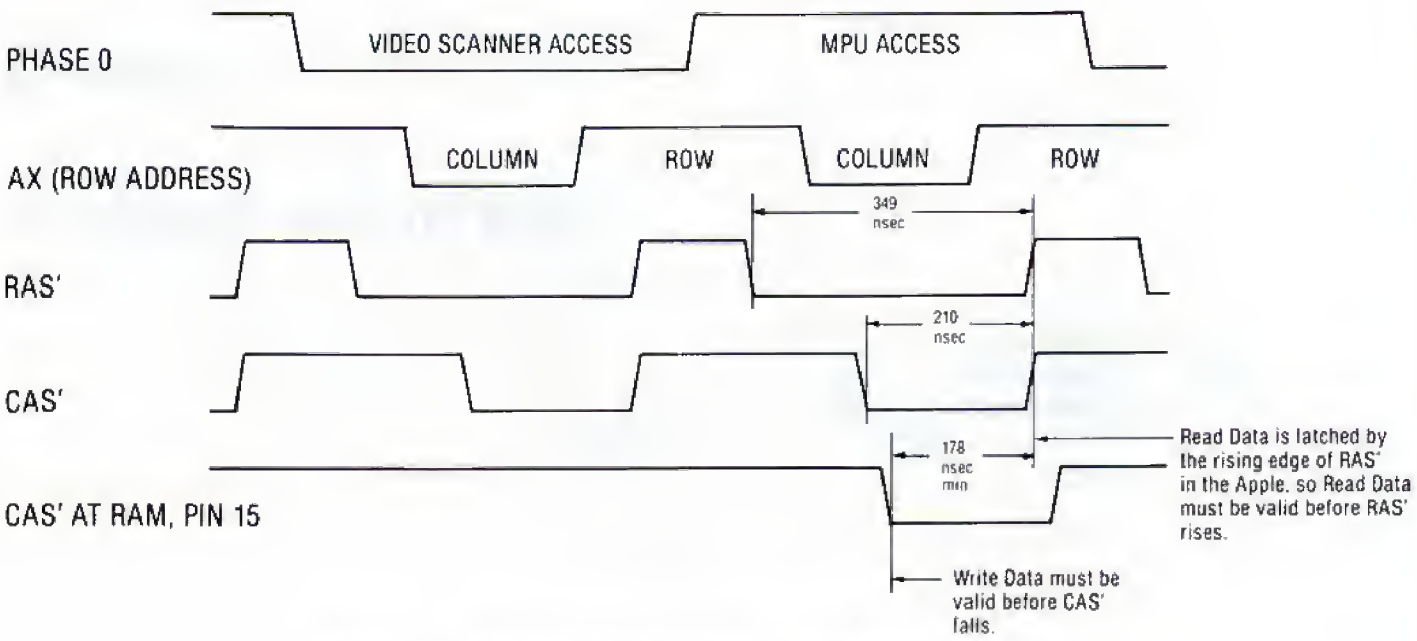


Figure 5.1 RAM Timing Signals from the Timing Generator.

the 16,384 memory cells as lying in a 128 by 128 matrix.* The first 7-bit address input to a RAM chip specifies which ROW the addressed cell lies in, and the second 7-bit address specifies the COLUMN. RAS' falling clocks the ROW address to RAM. CAS' falling clocks the COLUMN address to RAM and initiates the read or write action.

Figure 5.1 shows the **timing generator** signals which control RAM access in the Apple. The nature of these signals was dictated by 16K and 4K dynamic RAM chip requirements, 1 MHz 6502 timing requirements, and the alternating access between the MPU and the video scanner. PHASE 0 and AX provide the timing reference for scanner ROW/COLUMN addressing and for MPU ROW/COLUMN addressing. RAS' is applied directly to the RAS' input of all the RAM chips. CAS' is applied through logic gating to one of three rows of RAM chips when RAM is accessed.

*As mentioned in Chapter 2, this book refers to a unit of memory which stores a bit of information as a **cell**. Each RAM chip has 16,382 cells and is capable of storing 16,382 bits of information. The eight associated cells which store a byte of information in the Apple are referred to as a memory **location**. The Apple has 49,152 RAM locations.

RAS' falling clocks the ROW address to the RAM chip. The address input to the chip must contain the ROW address when RAS' falls and the COLUMN address when CAS' falls. Placing the correct addressing signals at the address input to the RAM chips is the function of the **RAM address multiplexor**. CAS' initiates the data transfer by dropping low after RAS' has already dropped low. RAM in the Apple must be capable of responding to a read access within 349 nanoseconds of RAS' falling. Read data must be valid 210 nanoseconds after CAS' falls. CAS' falling can be delayed by as much as 32 nanoseconds by logic gating before it gets to RAM, so RAM must be able to respond to a read access within 178 nanoseconds of CAS' falling at its input. These requirements are met by 250 nanosecond or faster RAM chips.

Dynamic RAM must be periodically **refreshed** for it to operate properly. The refresh requirement of 16K chips is that each of the 128 possible ROW addresses must be accessed every two milliseconds, or 500 times a second. This can be accomplished in RAS'/CAS' cycles or in RAS'-only cycles. CAS' is only active at one third of the Apple RAM chips

at any one time, but RAS' is active at all RAM chips. The refresh requirement in the Apple is met in the process of scanning RAM for video output. While some chips are refreshed by RAS'/CAS' access from the video scanner, other chips, not part of the screen memory being scanned, are refreshed by RAS'-only access. RAM in the Apple is refreshed every 1.85 milliseconds while HIRES memory is being scanned, and every 1.6 milliseconds while TEXT/LORES memory is being scanned.

RAM CONNECTIONS IN THE APPLE

The general flow of RAM data was discussed in the chapter on bus structure. The bus structure diagram in the back of the book should be reviewed to reinforce, in your mind, how RAM is tied into the overall scheme of things in the Apple. The basic features of RAM connections in the Apple are:

1. RAM data input is tied directly to the data bus for MPU writing.
2. RAM data output is saved in an eight-bit data latch when RAS' rises. The latched data is routed to the data bus through the tri-state RAM/keyboard data multiplexor.
3. The latched data is also routed to the video generator for processing.
4. The RAM address input is multiplexed among the address bus ROW address, address bus COLUMN address, video scanner ROW address and video scanner COLUMN address.
5. RAM latched data is gated to the data bus when the scanner or MPU is addressing RAM and R/W' is high.
6. Physically, the RAM is arranged into three rows, each containing eight RAM chips and 16,384 bytes of memory. Selection among the three rows is accomplished by enabling CAS' only to the row being accessed.

Figure 5.2 is a schematic diagram showing the connections of RAM, the **RAM data latch** and the **RAM/keyboard data multiplexor.** The RAM chips are connected together in a way that reminds you of the wiring of the peripheral slots. The majority of the RAM lines are just strung from chip to chip. This includes the address input (RA0-RA6), the four power supply inputs, RAM R/W', and RAS'. In older Apples, RA6 was distributed to all of RAM when 16K jumpers were installed. Newer Apples are hard-wired for 16K RAM, and RA6 is distributed like RA0 through RA5.

The RAM R/W' signal is not the same as system R/W' from the 6502. It is system R/W' gated by PHASE 0. The 6502 R/W' line drops low some time during PHASE 1 of write cycles. This would interfere with video scanner reading if 6502 R/W' were connected directly to RAM. Another way of looking at this is that the video scanner controls the RAM address and RAM R/W' during PHASE 1. The video scanner always reads, never writes.

Each row of RAM in the Apple is the equivalent of a 16,384 byte read/write memory with eight data inputs and eight data outputs. The RAM D7 inputs of rows C, D, and E are all tied to D7 of the data bus, and RAM D0-D6 inputs are similarly tied directly to the data bus. The RAM D7 outputs of rows C, D, and E are tied to the D7 input of the RAM data latch, and RAM D0-D6 outputs are similarly tied to the other seven inputs to the data latch. As mentioned before, CAS' row C, CAS' row D, and CAS' row E serve as the enabling signals for the three rows of RAM.

The RAM latch is made up of two hex flip-flop chips, but only eight of the flip-flops are needed to latch RAM output data. One of the remaining flip-flops is not used, and the other three are used in the video generator. RAM data is saved in the latch every time RAS' transits from low to high.

The RAM/keyboard data multiplexor provides a tri-state connection to the data bus for latched RAM data and for the keyboard input. Keyboard data is selected by the multiplexor when $C00X is on the address bus during PHASE 0. Enabling of multiplexor outputs to the data bus is controlled by the RAM SELECT' signal generated in the address multiplexor. The output is enabled when the 6502 R/W' line is high and when:

The scanner or MPU is addressing RAM
or
$C00X is on the address bus during PHASE 0.

The $C00X signal is developed in the **address decode** section of the Apple (Figure 7.2).

There are some subtle but important points about the handling of RAM output in the Apple. It only makes sense that RAM output data is isolated from the data bus during MPU write cycles. The MPU controls the data bus for the second half of PHASE 0 and slightly beyond during write cycles. However, nothing in a write cycle prevents the video scanner from reading RAM, and nothing prevents the latched RAM output from being processed in the video generator. Therefore, write cycles do not cause random flicker in the Apple's video display.

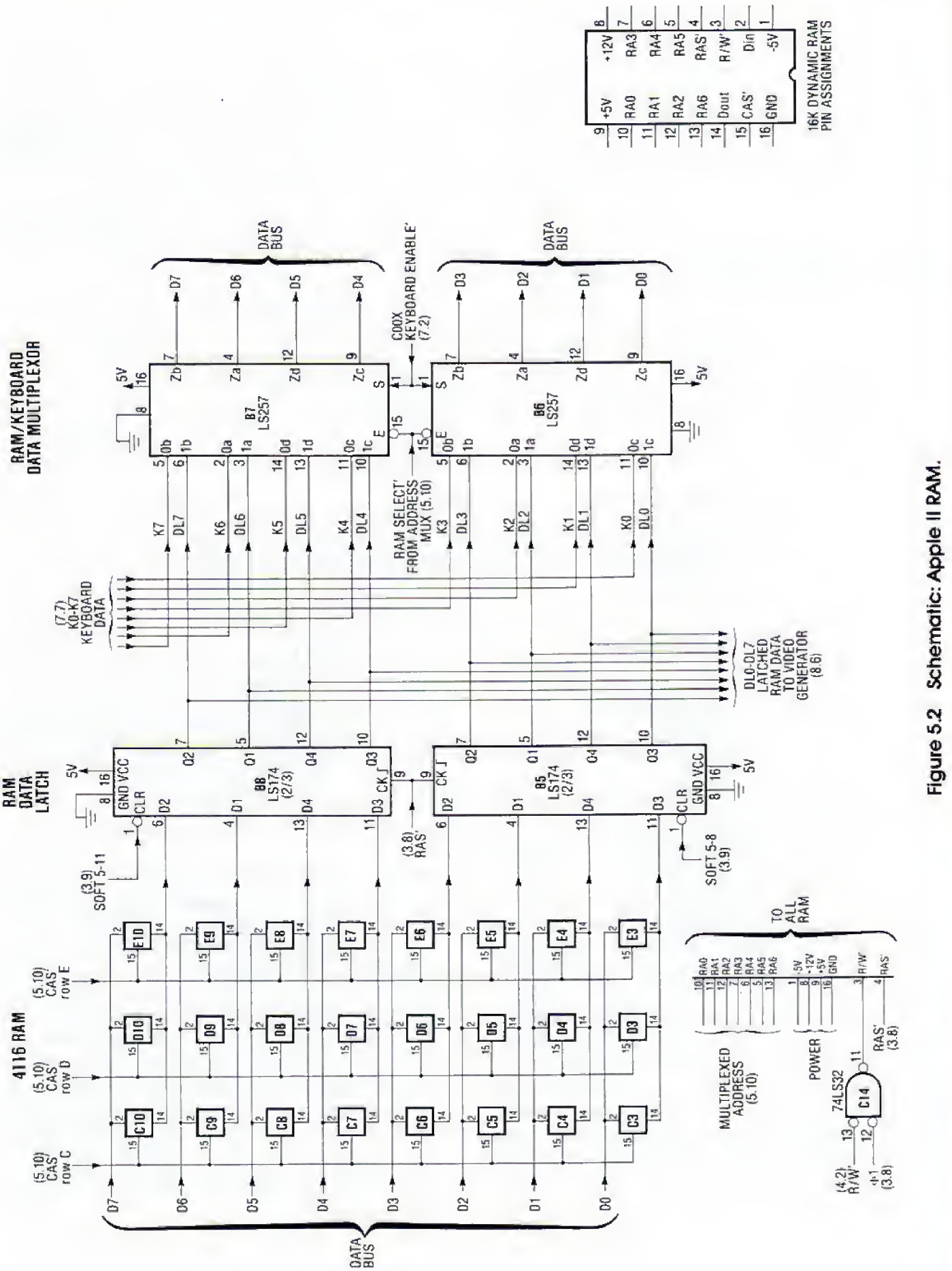


Figure 5.2 Schematic: Apple II RAM.

Another interesting point is that latched output from the video scanner access is gated to the data bus when the MPU is not writing. This data is not needed on the data bus by any motherboard device. Its presence on the data bus means that it can be read by peripheral cards when the MPU is not writing. Many conceivable peripheral designs could make use of this data. The programmable video scanner simulator, described in a Chapter 3 Application Note, is one such design (see Figure 3.13).

The propagation delay from RAS' rising to data valid on the data bus is significant. It typically takes 20 nanoseconds (30 max) for data to become valid at the latch after RAS' rises. It typically takes 12 nanoseconds (18 max) for data to propagate through the RAM/keyboard multiplexor. The total delay in the author's Apple is 35 nanoseconds. Because of this delay, the Apple does not meet the 100 nanosecond minimum read data setup time supposedly required by Synertek and MOS Technology 6502s.

THE RAM ADDRESS MULTIPLEXOR

The RAM address multiplexor can be functionally divided into two parts. The larger part develops RA0 through RA6, the multiplexed RAM address input. The smaller part develops the RAM SELECT' signal and CAS' for rows C, D, and E. The generation of RA0-RA6 is a 4 to 1 multiplex operation controlled by AX and PHASE 0. The generation of CAS'-C, CAS'-D, CAS'-E and RAM SELECT' is a 2 to 1 multiplex operation controlled by PHASE 0.

The functions of the address multiplexor are summarized in Figure 5.3. The CAS' table and MULTIPLEXED RAM ADDRESS table show which address bus and video scanner outputs control CAS' and the RAM address at any moment. The various blocks show the general hardware elements of the address multiplexor.

The gating of CAS' and the gating of RAM data to the data bus are closely related. The high order address bits contain the information that RAM is being accessed, and they also contain the information that row C or D or E is being accessed. The CAS' table in Figure 5.3 shows that during PHASE 0, CAS' falls low at row C, row D, or row E depending on A15 and A14 of the address bus. In a sense, CAS' is part of the RAM address. Addresses $0000-$3FFF access row C; address $4000-$7FFF access row D; and address $8000-$BFFF access row E.

The CAS' gating partially defines screen memory in the Apple. During scanner access, CAS' never falls in row E. It falls in row D when HIRES PAGE 2 is being scanned, and it falls in row C when HIRES PAGE 2 is not being scanned. This means that all screen memory is between $0000 and $3FFF except HIRES PAGE 2 which is between $4000 and $7FFF. Notice that CAS' is gated during PHASE 1 with no input from the video scanner. The screen modes are an extension of the video scanner when it comes to addressing RAM. Generally, the screen modes control the high order address bits, thus determining what areas of RAM the video scanner reads.

The RAM SELECT' term is one of the more important signals in the Apple, being the primary data bus management signal. When it is low, the latched RAM data or the keyboard data is gated to the data bus. RAM SELECT' goes low when R/W' is high and the keyboard or RAM is accessed. The video scanner always accesses RAM during PHASE 1, so latched RAM data is always on the data bus shortly after PHASE 0 drops low unless R/W' is low.

The gating of video data to the data bus is an interesting feature, because it is not necessary for generation of Apple video. The video data is not read by any motherboard device on the data bus, but it can be read by peripheral cards. Additionally, the ability of the MPU to read the video data when the data bus is floating is a very useful design accident.

The HIRES TIME signal which is used in RAM addressing is developed in the video generator. It is high when the Apple is in HIRES, GRAPHICS, NO MIX mode or in HIRES, GRAPHICS, MIX mode when (V4 • V2) is false. The V4 • V2 gating switches the scanned memory over to TEXT memory for four lines of text at the bottom of the screen. Naturally, the MIXED mode requires switching between GRAPHICS and TEXT in sync with the video scanner.

The RAM address inputs are selected from the address bus, the video scanner state, and the screen mode. The MULTIPLEXED RAM ADDRESS table in Figure 5.3 shows the way address bus lines and video scanner output lines are assigned to RA0-RA6 ROW and RA0-RA6 COLUMN. There are some significant aspects to these address assignments:

1. The scanner low order bits are assigned to RAM ROW address inputs so the RAM will be refreshed by the video scanner.
2. The address bus bit which controls a given RAM address will be equivalent to the scanner bit which controls the same RAM address. For example, A0 controls RA2 during an MPU ROW access, and H0 controls RA2 during a

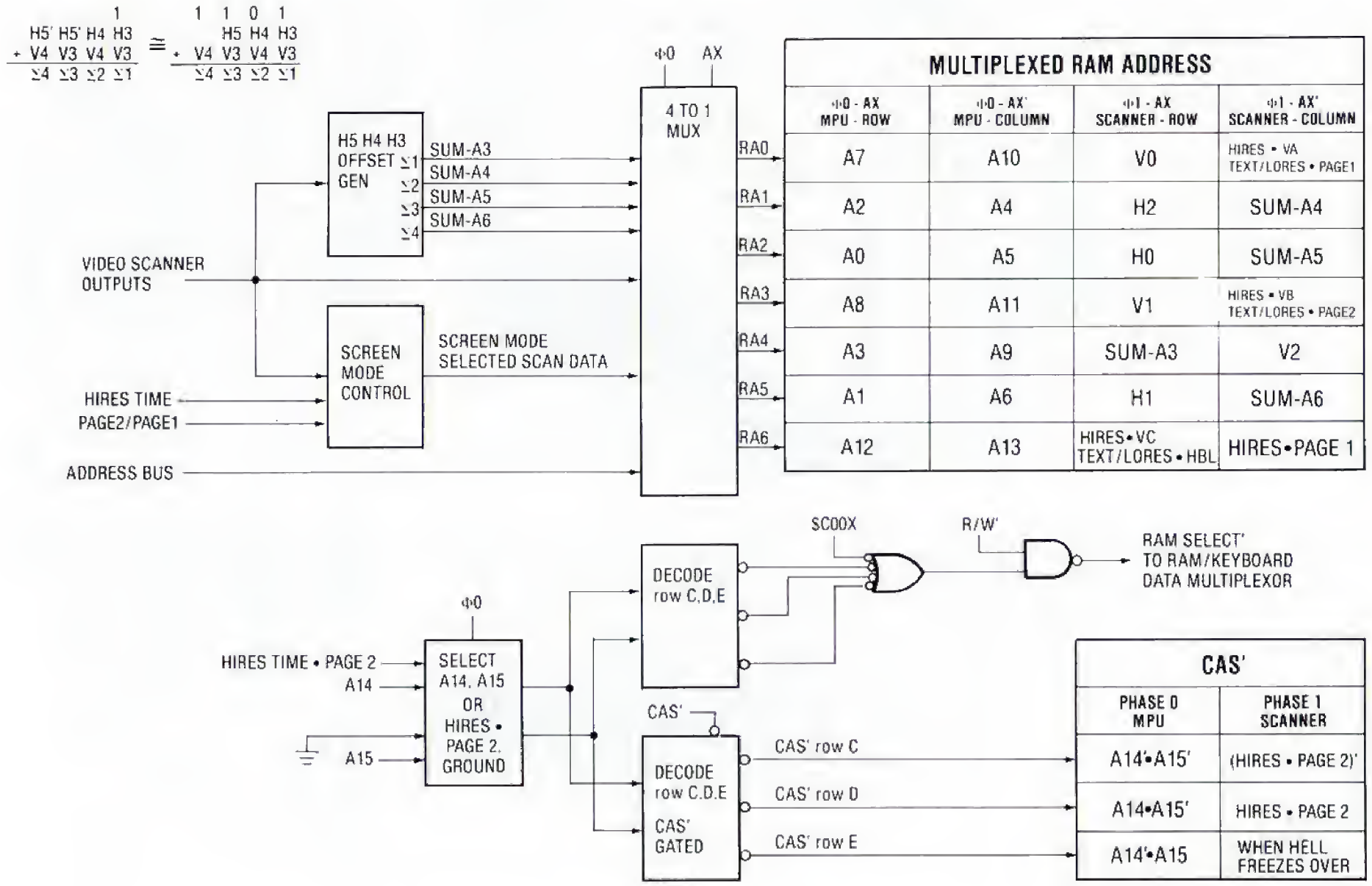


Figure 5.3 The Functions of the Address Multiplexer.

scanner ROW access. This means that A0 and H0 perform the equivalent RAM addressing function.

3. A12 and A13 are assigned to RA6 as part of the Apple scheme for operating with 4K or 16K RAM chips.
4. The other assignment features were probably determined by convenience of mechanical layout.

Table 5.1 shows the equivalent address bus/video scanner address bits. You can use this table to take any screen mode and video scanner state and convert them to an equivalent MPU address. If you store a byte of data at the equivalent MPU address, it will be driven out of RAM during PHASE 1 when the video scanner reaches the chosen state.

The Arithmetic of Video Scanner Memory Addressing

If the Apple isn't famous for the encrypted nature of its screen memory addressing, it should be. The programmer has a very heavy burden in computing or looking up seemingly illogical addresses. There is logic to the Apple screen memory addressing. It is the logic of binary manipulation. The way to understand it is to look at the Apple from the designer's viewpoint. In 1975, how would you have gotten the Apple to display HIRES color graphics, LORES color graphics, and 40 columns of text?

Forty columns? Two strikes against you to start with. Didn't Wozniak ever hear of powers of two? Digital computers are based in binary numbers. Use 32, 64, or 128 columns. This is as bad as the guys who designed 80-column typewritten page widths and 10-digit humans.

The problem is that you want to address memory sequentially with the output lines of the video scanner. If the Apple line width had been 32 columns, you could just tie H0-H5 and V0-V4 directly to the 4 to 1 address multiplexor. Memory would be very neatly divided up into 32 X 24 bytes. With 40 columns, you could still tie H0-H5 directly to the 4 to 1 address multiplexor, and this would create an easy hardware connection, but there would be unused gaps in memory 24 bytes long for every 40 bytes used. This would waste 576 bytes of memory in TEXT/LORES modes and 4608 bytes in HIRES mode. What good are 4608 bytes of memory divided up into 192 noncontiguous groups of 24 bytes?

In the Apple, it was accepted that there would be some waste of memory caused by the 40 character lines, but the waste was minimized at the expense of a little hardware complexity. Instead of using 40 bytes out of each 64-byte memory segment, 120 bytes out of each 128-byte memory segment are used. This creates eight bytes of wasted memory for every three horizontal scans in HIRES or every three lines of characters in TEXT. This results in a

Table 5.1 MPU/Scanner Equivalent Address Bits.

MPU	VIDEO SCANNER	
A0	H0	
A1	H1	
A2	H2	
A3	SUM-A3	
A4	SUM-A4	
A5	SUM-A5	
A6	SUM-A6	
A7	V0	
A8	V1	
A9	V2	
A10	HIRES • VA	TEXT/LORES • PAGE 1
A11	HIRES • VB	TEXT/LORES • PAGE 2
A12	HIRES • VC	TEXT/LORES • HBL
A13	HIRES • PAGE 1	—
A14	HIRES • PAGE 2	—
A15	—	

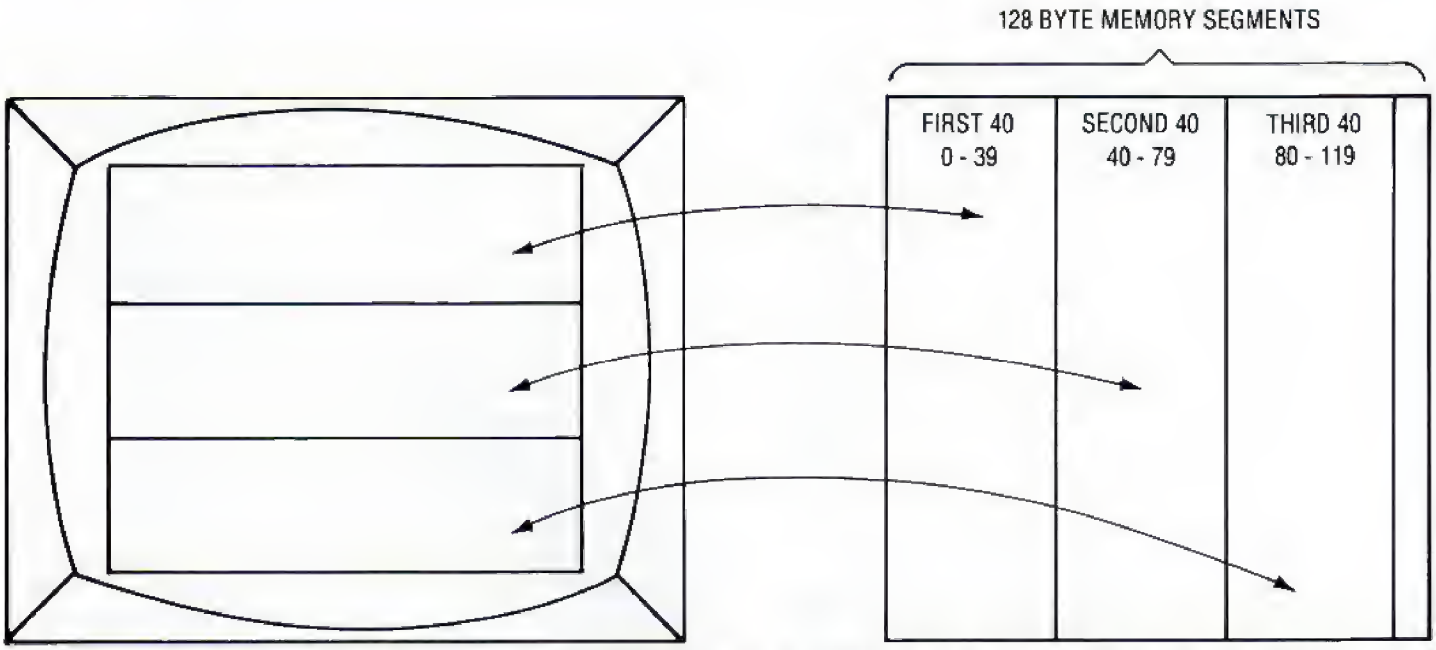


Figure 5.4 128-Byte Video Memory Segments Consist of Three 40-Byte Sections, Each Mapped Into a Different Part of the Video Screen.

total wastage of 512 bytes in HIRES and 64 bytes in TEXT/LORES.

How do you implement this in hardware? Screen memory is divided into **128-byte segments**. Each segment is divided into the **FIRST 40**, the **SECOND 40**, the **THIRD 40**, and eight bytes of no man's memory (**UNUSED 8**). It so happens that the displayed television scan is neatly divided into three sections by V3 and V4 from the video scanner as follows:

V4' V3' -	Top third of television screen
V4' V3 -	Middle third of television screen
V4 V3' -	Bottom third of television screen
V4 V3 -	Undisplayed (VBL)

Because the three displayed portions of the screen can be so easily detected, they are mapped into the three 40-byte sections of each 128-byte memory segment as follows:

LOCATION ON TV SCREEN	LEAST SIGNIFICANT BITS OF ADDRESS
Top	0000000 - 0100111 (FIRST 40)
Middle	0101000 - 1001111 (SECOND 40)
Bottom	1010000 - 1110111 (THIRD 40)

It can be seen that in the binary representations of the 40-byte address sections, the lower three bits cross over from 111 to 000 at all section boundaries. This means that these three bits can be identically addressed in the FIRST 40, SECOND 40, or THIRD 40. For example, the lowest three bits of the address of the left most character in any section is 000. For this reason H0, H1, and H2 are direct address inputs to the 4 to 1 multiplexor and are address equivalents of A0, A1, and A2.

The next four address bits are different depending on the 40-byte section that is being addressed. They are 0000 through 0100 in the FIRST 40, 0101 through 1001 in the SECOND 40, and 1010 through 1110 in the THIRD 40. These four bits are addressed by H5-H4-H3 plus an offset. The offset value is selected by V4 and V3 to place the scanned memory address in the FIRST 40, SECOND 40, or THIRD 40 of the current 128-byte segment. The offset is added to H5-H4-H3 in a single chip 4-bit adder, and the four bits of the resulting SUM become the scanning address bits equivalent to A3, A4, A5, and A6. This book refers to the SUM bits as SUM-A3, SUM-A4, SUM-A5, and SUM-A6.

There are eight states of H5-H4-H3 but only five of the states are displayed. 000 through 010 are undisplayed and occur during the right margin, horizontal retrace, and left margin of the television

scan. 011 is the first displayed count, and when H5-H4-H3 reaches 011, it is time to address the first byte of a 40-byte section. Suppose you built the following summing circuit:

```
                H5          H4          H3
+   V4          V3          V4          V3
  SUM-A6      SUM-A5      SUM-A4      SUM-A3
```

This would create the three offsets 0, 101, and 1010 which are 40-byte offsets. This circuit would work, but it would make screen memory address assignments even more complex than they are for the Apple programmer. Since the display starts at H5-H4-H3 = 011, we need to subtract 011 from the offsets 000, 101, and 1010 to make the FIRST 40 start at a natural 128-byte segment boundary. The required offsets are 1101, 010, and 111 (-3, 2, and 7 in decimal). You need to address A6-A5-A4-A3 with the values H5-H4-H3 minus 011 in the FIRST 40, H5-H4-H3 plus 010 in the SECOND 40, and H5-H4-H3 plus 111 in the THIRD 40. These sums are cleverly created in the Apple by the following addition:

```
                                    1
    H5'         H5'         H4      H3
+   V4          V3          V4      V3
  SUM-A6      SUM-A5      SUM-A4      SUM-A3
```

H5'-H5'-H4-H3 is equal to H5-H4-H3 minus 100 in 4-bit signed binary arithmetic. 001 minus 100 is -011, so the needed offset is developed. It is easy to add 1 because of the carry input to the 4-bit adder. The equivalent adding circuit is:

```
    1           1           0       1
                H5          H4      H3
+   V4          V3          V4      V3
  SUM-A6      SUM-A5      SUM-A4      SUM-A3
```

TEXT/LORES Scanning

Beyond A6, scanning address assignments determine the memory blocks scanned in the various screen modes. V0, V1, and V2 are equivalent to A7, A8, and A9 in all screen modes. A TEXT/LORES screen memory page is made up of eight adjacent 128-byte segments. These eight adjacent segments are defined by V0, V1, and V2 in the scanner and by A7, A8, and A9 on the address bus. VA, VB, and VC play no part in addressing TEXT/LORES memory. Rather, the same 40-byte section of memory is scanned for eight adjacent horizontal television lines. It takes eight horizontal television lines to paint a line of text or two rows of LORES blocks. In the video generator, VA, VB, and VC define which vertical part of a text character it is time to draw, and VC defines which of two LORES blocks it is time to draw.

In TEXT/LORES, A15, A14, and A13 equivalents are false (low, ground, zip). A12 equivalents are false during display and true during HBL (Horizontal BLanking gate).* The A11 equivalent is PAGE 2 and the A10 equivalent is PAGE 1. This results in the memory scanned areas for TEXT/LORES shown at the bottom of this page. The reason for scanning different memory during HBL has nothing to do with video display. HBL on the RA6 ROW address line helps in the task of refreshing RAM. If not for the refresh requirement, the TEXT/LORES equivalent of A12 would be false and the memory scanned during HBL would be in the same range as the memory scanned during HBL'.

Figure 5.5 is the TEXT/LORES displayed memory map. This map shows the same information as the maps of the *Apple II Reference Manual*, but there is a difference in layout. The reference manual maps accent the 24 lines of text or 48 lines of LORES blocks, but Figure 5.5 accents the division of screen memory into 128-byte memory segments. This should give the reader a second perspective from which to view the screen mapping.

*HBL and VBL are signals produced in the video generator. HBL (the horizontal blanking gate) is high during the right margin, horizontal retrace, and left margin of the Apple video display. VBL (vertical blanking gate) is high during the bottom margin, vertical retrace, and top margin of the Apple video display. Refer to Chapter 8 for more information about these signals.

SCREEN MODE	BINARY	HEXADECIMAL
PAGE 1 Displayed	0000 01XX XXXX XXXX	$04XX-$07XX
PAGE 1 HBL	0001 01XX XXXX XXXX	$14XX-$17XX
PAGE 2 Displayed	0000 10XX XXXX XXXX	$08XX-$0BXX
PAGE 2 HBL	0001 10XX XXXX XXXX	$18XX-$1BXX

	BASE	TOP SCREEN/ FIRST 40		MIDDLE SCREEN/ SECOND 40		BOTTOM SCREEN/ THIRD 40		UNUSED 8
	ADDRESS	LIN#	RANGE	LIN#	RANGE	LIN#	RANGE	RANGE
	$400	00	$400-$427	08	$428-$44F	16	$450-$477	$478-$47F
	1024		1024-1063		1064-1103		1104-1143	1144-1151
	$480	01	$480-$4A7	09	$4A8-$4CF	17	$4D0-$4F7	$4F8-$4FF
	1152		1152-1191		1192-1231		1232-1271	1272-1279
	$500	02	$500-$527	10	$528-$54F	18	$550-$577	$578-$57F
	1280		1280-1319		1320-1359		1360-1399	1400-1407
	$580	03	$580-$5A7	11	$5A8-$5CF	19	$5D0-$5F7	$5F8-$5FF
PAGE 1	1408		1408-1447		1448-1487		1488-1527	1528-1535
	$600	04	$600-$627	12	$628-$64F	20	$650-$677	$678-$67F
	1536		1536-1575		1576-1615		1616-1655	1656-1663
	$680	05	$680-$6A7	13	$6A8-$6CF	21	$6D0-$6F7	$6F8-$6FF
	1664		1664-1703		1704-1743		1744-1783	1784-1791
	$700	06	$700-$727	14	$728-$74F	22	$750-$777	$778-$77F
	1792		1792-1831		1832-1871		1872-1911	1912-1919
	$780	07	$780-$7A7	15	$7A8-$7CF	23	$7D0-$7F7	$7F8-$7FF
	1920		1920-1959		1960-1999		2000-2039	2040-2047
	$800	00	$800-$827	08	$828-$84F	16	$850-$877	$878-$87F
	2048		2048-2087		2088-2127		2128-2167	2168-2175
	$880	01	$880-$8A7	09	$8A8-$8CF	17	$8D0-$8F7	$8F8-$8FF
	2176		2176-2215		2216-2255		2256-2295	2296-2303
	$900	02	$900-$927	10	$928-$94F	18	$950-$977	$978-$97F
	2304		2304-2343		2344-2383		2384-2423	2424-2431
	$980	03	$980-$9A7	11	$9A8-$9CF	19	$9D0-$9F7	$9F8-$9FF
PAGE 2	2432		2432-2471		2472-2511		2512-2551	2552-2559
	$A00	04	$A00-$A27	12	$A28-$A4F	20	$A50-$A77	$A78-$A7F
	2560		2560-2599		2600-2639		2640-2679	2680-2687
	$A80	05	$A80-$AA7	13	$AA8-$ACF	21	$AD0-$AF7	$AF8-$AFF
	2688		2688-2727		2728-2767		2768-2807	2808-2815
	$B00	06	$B00-$B27	14	$B28-$B4F	22	$B50-$B77	$B78-$B7F
	2816		2816-2855		2856-2895		2896-2935	2936-2943
	$B80	07	$B80-$BA7	15	$BA8-$BCF	23	$BD0-$BF7	$BF8-$BFF
	2944		2944-2983		2984-3023		3024-3063	3064-3071

Figure 5.5 TEXT/LORES Displayed Memory Map.

In addition to the displayed memory locations, there is reason to know what areas of memory are being scanned while nothing is being displayed. This knowledge has applications when software or hardware syncs to the video scan by detecting the scanned memory output on the data bus. This is possible in software by reading an address which does not result in data being placed on the data bus. For example, if you zero out all scanned memory except for the bytes in the blanking period preceding a given horizontal display line, you can detect the beginning of that horizontal scan with the following loop.

```
KBDSTRB    EQU    $C010
WAIT       LDA    KBDSTRB
           BPL    WAIT
```

Some techniques of exploiting this capability are discussed in an Application Note at the end of this chapter. The point is that it is sometimes useful to know what areas of memory are being scanned during blanking periods.

Figure 5.6 is a TEXT/LORES map showing the areas of memory scanned during displayed and undisplayed periods. The layout is similar to the

maps in the *Apple II Reference Manual*. The area of memory scanned previous to every horizontal display period is shown directly to the left of the memory scanned during that display period. The vertical blanking period is shown at the bottom. The considerations which determine the memory scanned during the blanking periods are as follows:

1. HBL is equivalent to A12 in TEXT/LORES mode, so the memory scanned during HBL is completely separate from the memory scanned during HBL'. This is not true in HIRES mode.
2. HBL scanned memory begins $18 bytes before display scanned memory plus $1000. The HBL base address can be computed from the displayed base address using this Applesoft program sequence:

```
10 HBL = BASE-24
20 IF INT(HBL/128)<>INT(BASE/128)
   THEN HBL = HBL+128
30 HBL = HBL+1024.
```

   Step 20 of the above program is necessary because horizontal memory addressing wraps around at the 128-byte segment boundaries.
3. The first address of HBL is always addressed twice consecutively, because H0-H5 is in the all zero state for two consecutive scans.
4. During VBL (Vertical BLanking), V3 and V4 are both true. The horizontal offset sum becomes H5-H4-H3 minus 0100. This is almost the same as the top of the displayed screen (H4-H3-H2 minus 0011). The VBL base addresses are equal to the FIRST 40 base addresses minus eight bytes using 128-byte wraparound subtraction. Example: $400 minus $8 gives $478; not $3F8.
5. Horizontal scanning wraps around at the 128-byte segment boundaries. Example: the address scanned before address $400 is $47F during VBL. The address scanned before $400 when VBL is false is $147F.

		HORIZONTAL BLANKING (HBL)			HORIZONTAL DISPLAY ENABLE		
	LINE NUM	PAGE 1	PAGE 2	11111111 00123456789ABCDEF01234567	PAGE 1	PAGE 2	1111111111111111122222222 0123456789ABCDEF0123456789ABCDEF01234567
SCREEN TOP	0	$1468 5224	$1868 6248	++++++++++########++++++++++	$400 1024	$800 2048	++
	1	$14E8 5352	$18E8 6376	++++++++++########++++++++++	$480 1152	$880 2176	++
	2	$1568 5480	$1968 6504	++++++++++########++++++++++	$500 1280	$900 2304	++
	3	$15E8 5608	$19E8 6632	++++++++++########++++++++++	$580 1408	$980 2432	++
	4	$1668 5736	$1A68 6760	++++++++++########++++++++++	$600 1536	$A00 2560	++
	5	$16E8 5864	$1AE8 6888	++++++++++########++++++++++	$680 1664	$A80 2688	++
	6	$1768 5992	$1B68 7016	++++++++++########++++++++++	$700 1792	$B00 2816	++
	7	$17E8 6120	$1BE8 7144	++++++++++########++++++++++	$780 1920	$B80 2944	++
SCREEN MIDDLE	8	$1410 5136	$1810 6160	++++++++++########++++++++++	$428 1064	$828 2088	++
	9	$1490 5264	$1890 6288	++++++++++########++++++++++	$4A8 1192	$8A8 2216	++
	10	$1510 5392	$1910 6416	++++++++++########++++++++++	$528 1320	$928 2344	++
	11	$1590 5520	$1990 6544	++++++++++########++++++++++	$5A8 1448	$9A8 2472	++
	12	$1610 5648	$1A10 6672	++++++++++########++++++++++	$628 1576	$A28 2600	++
	13	$1690 5776	$1A90 6800	++++++++++########++++++++++	$6A8 1704	$AA8 2728	++
	14	$1710 5904	$1B10 6928	++++++++++########++++++++++	$728 1832	$B28 2856	++
	15	$1790 6032	$1B90 7056	++++++++++########++++++++++	$7A8 1960	$BA8 2984	++
SCREEN BOTTOM	16	$1438 5176	$1838 6200	++++++++++########++++++++++	$450 1104	$850 2128	++
	17	$14B8 5304	$18B8 6328	++++++++++########++++++++++	$4D0 1232	$8D0 2256	++
	18	$1538 5432	$1938 6456	++++++++++########++++++++++	$550 1360	$950 2384	++
	19	$15B8 5560	$19B8 6584	++++++++++########++++++++++	$5D0 1488	$9D0 2512	++
	20	$1638 5688	$1A38 6712	++++++++++########++++++++++	$650 1616	$A50 2640	++
	21	$16B8 5816	$1AB8 6840	++++++++++########++++++++++	$6D0 1744	$AD0 2768	++
	22	$1738 5944	$1B38 6968	++++++++++########++++++++++	$750 1872	$B50 2896	++
	23	$17B8 6072	$1BB8 7096	++++++++++########++++++++++	$7D0 2000	$BD0 3024	++
VERTICAL BLANKING	24	$1460 5216	$1860 6240	++++++++++########++++++++++	$478 1144	$878 2168	++
	25	$14E0 5344	$18E0 6368	++++++++++########++++++++++	$4F8 1272	$8F8 2296	++
	26	$1560 5472	$1960 6496	++++++++++########++++++++++	$578 1400	$978 2424	++
	27	$15E0 5600	$19E0 6624	++++++++++########++++++++++	$5F8 1528	$9F8 2552	++
	28	$1660 5728	$1A60 6752	++++++++++##################	$678 1656	$A78 2680	##
	29	$16E0 5856	$1AE0 6880	++++++++++########++++++++++	$6F8 1784	$AF8 2808	++
	30	$1760 5984	$1B60 7008	++++++++++########++++++++++	$778 1912	$B78 2936	++
	31	$17E0 6112	$1BE0 7136	++++++++++########++++++++++	$7F8 2040	$BF8 3064	++

The last row of memory is scanned 14 consecutive times. All other rows are scanned 8 consecutive times.

HORIZONTAL SYNC (REV-7 & LATER)

VERTICAL SYNC

Figure 5.6 TEXT/LORES Video Scanning Map.

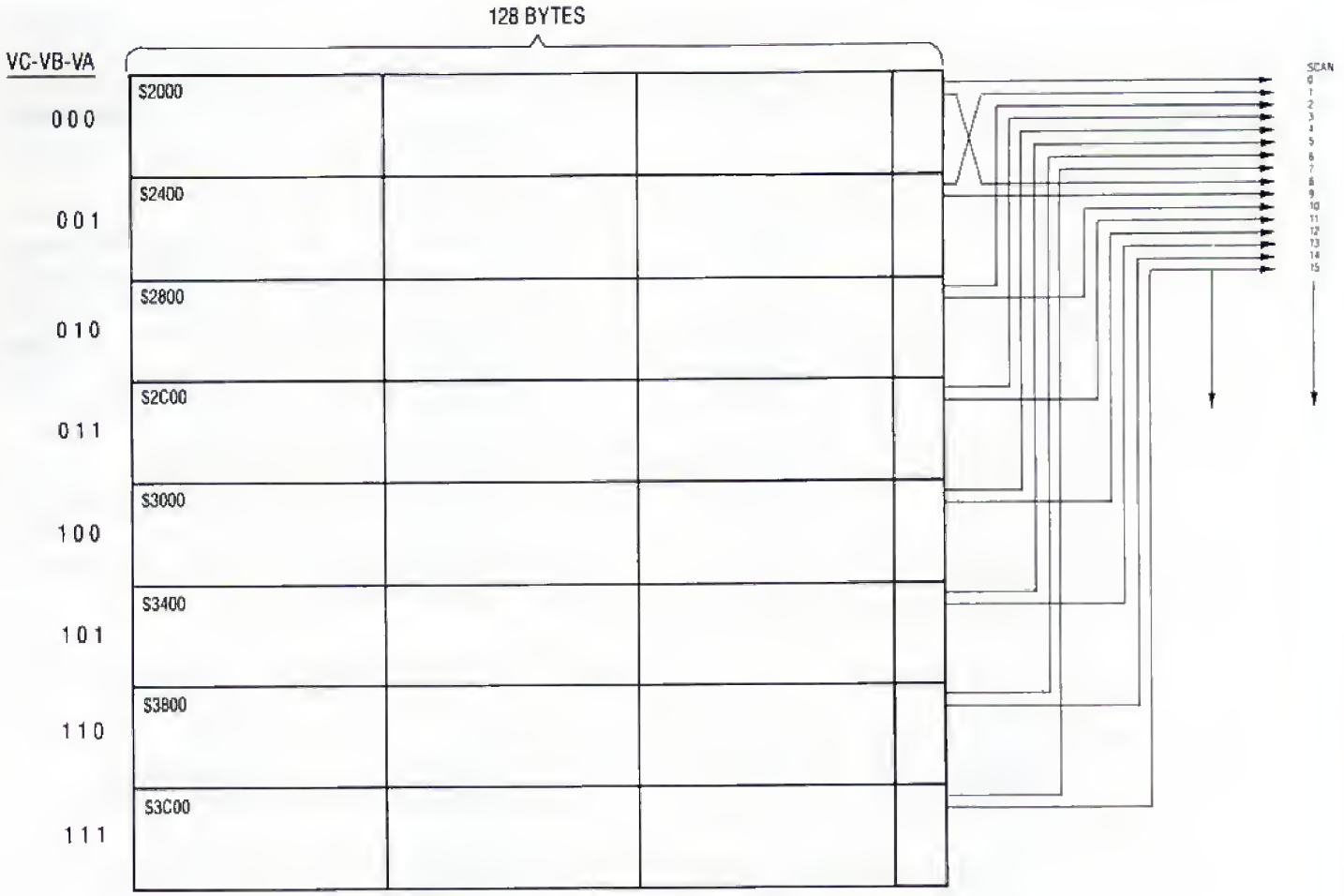


Figure 5.7 HIRES Memory Areas (Page 1).

HIRES Scanning

Table 5.1 shows that HIRES video scanner addressing is identical to TEXT/LORES addressing on bits A0-A9 and A15. The differences in bits A10-A14 reflect the facts that HIRES memory is eight times as big as TEXT/LORES memory, and that HIRES memory is in a different location than TEXT/LORES memory.

During HIRES scanning, A13 is equivalent to PAGE 1, and A14 is equivalent to PAGE 2. This results in a PAGE 1 base address of $2000 and a PAGE 2 base address of $4000.

VA, VB, and VC are equivalent to A10, A11, and A12 in HIRES. This is the most important point, because it represents the great difference between HIRES and TEXT/LORES. In TEXT/LORES, 40 bytes contain the display intelligence for eight horizontal scans. In HIRES, 40 bytes contain the display intelligence for one horizontal scan. The HIRES scan must address a different 40-byte section every scan. This is accomplished by letting VA, VB, and VC affect the memory address in the HIRES scan.

Notice in Table 5.1 the oddity that VA, VB, and VC address higher order bits of memory than V0, V1, and V2. This causes the base addresses of adjacent HIRES lines to be separated by 1024 bytes rather than the logical 128 bytes. This extra complication of HIRES address computation could have been eliminated in the Apple design by the addition of one chip. It wasn't, so the user suffers an extra operational distraction. One way to look at the HIRES memory layout is as eight adjacent areas with each area the memory equivalent of a single TEXT/LORES page. VA, VB, and VC determine which of the eight areas is being addressed. As eight adjacent horizontal lines are scanned, one 64-byte (40 bytes displayed) section from each of the eight memory areas is scanned. As in TEXT/LORES, the top, middle, and bottom thirds of the screen are accompanied by memory scanning of the FIRST 40, SECOND 40, and THIRD 40 sections respectively.

One way to gain insight into the overall layout of HIRES memory is to run the following BASIC program:

```
10 HGR : POKE -16302,0 :
   REM HIRES, NO MIX
20 FOR A = 8192 TO 16383
30 POKE A,255
40 FOR B = 0 TO 100 : NEXT B :
   REM DO IT SLOWLY
50 NEXT A : GO TO 10
```

This program fills the consecutive memory locations of HIRES, PAGE 1 with $FF; slowly so you can watch the screen fill. When the program runs, the screen should fill as follows:

1. Forty short lines are strung together at the top of the screen to form a long horizontal line.
2. The same thing happens at the first line of the middle third of the screen.
3. The same thing happens at the first line of the bottom third of the screen. $2000-$2077 (120 bytes) has now been filled.
4. The same thing happens on the ninth line from the top. The second line from the top will not be formed until the program gets to address $2400 (9216 decimal).
5. The pattern of lines continues to be drawn until there are 24 evenly spaced lines on the screen. At this point the screen is one eighth full, a 128-byte segment from each of the eight memory areas has been filled, and the program has reached $2400.
6. The program continues to add patterns of 24 lines in a similar fashion until the screen is filled.

Figure 5.8 is a HIRES displayed memory map accenting the division of screen memory into 128-byte memory segments. This figure was printed out using an Applesoft program listed in Appendix D. Like the TEXT/LORES map of Figure 5.5, this figure gives a different perspective for viewing HIRES memory usage. Figure 5.9 is the full HIRES memory map showing addresses scanned during HBL and VBL as well as the displayed map. It was also made by an Applesoft program listed in Appendix D. For reference, "#" is used in Figure 5.9 to show when horizontal or vertical television sync is output by the video generator. The #s in the middle of every HBL period represent the horizontal sync which causes the horizontal retrace. The long strings of #s in lines 224-227 represent the vertical sync which causes the vertical retrace. Please note that video sync in the Apple was changed in Revision 1, Revision 7, and the RFI Revision. The sync generation depicted in Figures 5.6 and 5.9 is that of the RFI Revision.

The scanning during blanking periods in HIRES is very similar to that in TEXT/LORES with one major exception. HBL has no effect on memory addressing in HIRES, so during HBL, the memory locations that are scanned are in the displayed memory area. The memory locations scanned during HBL prior to a displayed line are the 24 bytes just below the displayed area, using 128-byte wrap-around addressing.

VBL scanned memory is addressed in HIRES, just as in TEXT/LORES. The area scanned during VBL is the same as the top third of the screen minus eight bytes. Memory scanned by lines 256 through 261 is identical to memory scanned by lines 250 through 255, so those six 64-byte sections are scanned twice, as shown in Figure 5.9. The memory scanning areas are summarized in Table 5.2. This same information is displayed graphically in Figure 5.17.

Mixed Mode Scanning

HIRES graphics mixed with TEXT is a special case when it comes to video scanner addressing. Part of HIRES memory and part of TEXT/LORES memory must be scanned in this mode. The problem does not arise with LORES graphics mixed with TEXT, because TEXT memory scanning is identical to LORES memory scanning.

The HIRES TIME term that is used to scan memory addresses is not a direct input from the $C056/$C057 LORES/HIRES soft switch. Rather, it is a term developed in the video generator which is active when it is actually time to scan HIRES screen memory. In HIRES MIXED mode, the HIRES TIME term switches low on the third RAS' (rising) after V4 • V2 becomes true (see Figures 8.6 and 8.13). HIRES TIME switches high on the third RAS' (rising) after V4 • V2 becomes false. V4 • V2 identifies the last four lines of the TEXT display, and the three RAS' delay prevents switching from HIRES to TEXT mode in the video generator before the last seven dots in line 159 of HIRES have been output to the television.

V4 • V2 actually identifies scan lines 160 through 191 and 224 through 261. The scanned memory switches to HIRES during the first part of VBL, back to TEXT for the second part of VBL, then back to HIRES for the top of the screen. The switching during VBL is, of course, not visible on the screen. This information is only important to those special applications where it is important to know what is scanned during the blanking periods.

		TOP SCREEN/ FIRST 40		MIDDLE SCREEN/ SECOND 40		BOTTOM SCREEN/ THIRD 40		
PAGE 1	PAGE 2	LIN#	PAGE 1 RANGE	LIN#	PAGE 1 RANGE	LIN#	PAGE 1 RANGE	UNUSED 8 PAGE 1 RANGE
$2000 8192	$4000 16384	000	$2000-$2027	064	$2028-$204F	128	$2050-$2077	$2078-$207F
$2400 9216	$4400 17408	001	$2400-$2427	065	$2428-$244F	129	$2450-$2477	$2478-$247F
$2800 10240	$4800 18432	002	$2800-$2827	066	$2828-$284F	130	$2850-$2877	$2878-$287F
$2C00 11264	$4C00 19456	003	$2C00-$2C27	067	$2C28-$2C4F	131	$2C50-$2C77	$2C78-$2C7F
$3000 12288	$5000 20480	004	$3000-$3027	068	$3028-$304F	132	$3050-$3077	$3078-$307F
$3400 13312	$5400 21504	005	$3400-$3427	069	$3428-$344F	133	$3450-$3477	$3478-$347F
$3800 14336	$5800 22528	006	$3800-$3827	070	$3828-$384F	134	$3850-$3877	$3878-$387F
$3C00 15360	$5C00 23552	007	$3C00-$3C27	071	$3C28-$3C4F	135	$3C50-$3C77	$3C78-$3C7F
$2080 8320	$4080 16512	008	$2080-$20A7	072	$20A8-$20CF	136	$20D0-$20F7	$20F8-$20FF
$2480 9344	$4480 17536	009	$2480-$24A7	073	$24A8-$24CF	137	$24D0-$24F7	$24F8-$24FF
$2880 10368	$4880 18560	010	$2880-$28A7	074	$28A8-$28CF	138	$28D0-$28F7	$28F8-$28FF
$2C80 11392	$4C80 19584	011	$2C80-$2CA7	075	$2CA8-$2CCF	139	$2CD0-$2CF7	$2CF8-$2CFF
$3080 12416	$5080 20608	012	$3080-$30A7	076	$30A8-$30CF	140	$30D0-$30F7	$30F8-$30FF
$3480 13440	$5480 21632	013	$3480-$34A7	077	$34A8-$34CF	141	$34D0-$34F7	$34F8-$34FF
$3880 14464	$5880 22656	014	$3880-$38A7	078	$38A8-$38CF	142	$38D0-$38F7	$38F8-$38FF
$3C80 15488	$5C80 23680	015	$3C80-$3CA7	079	$3CA8-$3CCF	143	$3CD0-$3CF7	$3CF8-$3CFF
$2100 8448	$4100 16640	016	$2100-$2127	080	$2128-$214F	144	$2150-$2177	$2178-$217F
$2500 9472	$4500 17664	017	$2500-$2527	081	$2528-$254F	145	$2550-$2577	$2578-$257F
$2900 10496	$4900 18688	018	$2900-$2927	082	$2928-$294F	146	$2950-$2977	$2978-$297F
$2D00 11520	$4D00 19712	019	$2D00-$2D27	083	$2D28-$2D4F	147	$2D50-$2D77	$2D78-$2D7F
$3100 12544	$5100 20736	020	$3100-$3127	084	$3128-$314F	148	$3150-$3177	$3178-$317F
$3500 13568	$5500 21760	021	$3500-$3527	085	$3528-$354F	149	$3550-$3577	$3578-$357F
$3900 14592	$5900 22784	022	$3900-$3927	086	$3928-$394F	150	$3950-$3977	$3978-$397F
$3D00 15616	$5D00 23808	023	$3D00-$3D27	087	$3D28-$3D4F	151	$3D50-$3D77	$3D78-$3D7F
$2180 8576	$4180 16768	024	$2180-$21A7	088	$21A8-$21CF	152	$21D0-$21F7	$21F8-$21FF
$2580 9600	$4580 17792	025	$2580-$25A7	089	$25A8-$25CF	153	$25D0-$25F7	$25F8-$25FF
$2980 10624	$4980 18816	026	$2980-$29A7	090	$29A8-$29CF	154	$29D0-$29F7	$29F8-$29FF
$2D80 11648	$4D80 19840	027	$2D80-$2DA7	091	$2DA8-$2DCF	155	$2DD0-$2DF7	$2DF8-$2DFF
$3180 12672	$5180 20864	028	$3180-$31A7	092	$31A8-$31CF	156	$31D0-$31F7	$31F8-$31FF
$3580 13696	$5580 21888	029	$3580-$35A7	093	$35A8-$35CF	157	$35D0-$35F7	$35F8-$35FF
$3980 14720	$5980 22912	030	$3980-$39A7	094	$39A8-$39CF	158	$39D0-$39F7	$39F8-$39FF
$3D80 15744	$5D80 23936	031	$3D80-$3DA7	095	$3DA8-$3DCF	159	$3DD0-$3DF7	$3DF8-$3DFF
$2200 8704	$4200 16896	032	$2200-$2227	096	$2228-$224F	160	$2250-$2277	$2278-$227F
$2600 9728	$4600 17920	033	$2600-$2627	097	$2628-$264F	161	$2650-$2677	$2678-$267F
$2A00 10752	$4A00 18944	034	$2A00-$2A27	098	$2A28-$2A4F	162	$2A50-$2A77	$2A78-$2A7F
$2E00 11776	$4E00 19968	035	$2E00-$2E27	099	$2E28-$2E4F	163	$2E50-$2E77	$2E78-$2E7F
$3200 12800	$5200 20992	036	$3200-$3227	100	$3228-$324F	164	$3250-$3277	$3278-$327F
$3600 13824	$5600 22016	037	$3600-$3627	101	$3628-$364F	165	$3650-$3677	$3678-$367F
$3A00 14848	$5A00 23040	038	$3A00-$3A27	102	$3A28-$3A4F	166	$3A50-$3A77	$3A78-$3A7F
$3E00 15872	$5E00 24064	039	$3E00-$3E27	103	$3E28-$3E4F	167	$3E50-$3E77	$3E78-$3E7F
$2280 8832	$4280 17024	040	$2280-$22A7	104	$22A8-$22CF	168	$22D0-$22F7	$22F8-$22FF
$2680 9856	$4680 18048	041	$2680-$26A7	105	$26A8-$26CF	169	$26D0-$26F7	$26F8-$26FF
$2A80 10880	$4A80 19072	042	$2A80-$2AA7	106	$2AA8-$2ACF	170	$2AD0-$2AF7	$2AF8-$2AFF
$2E80 11904	$4E80 20096	043	$2E80-$2EA7	107	$2EA8-$2ECF	171	$2ED0-$2EF7	$2EF8-$2EFF
$3280 12928	$5280 21120	044	$3280-$32A7	108	$32A8-$32CF	172	$32D0-$32F7	$32F8-$32FF
$3680 13952	$5680 22144	045	$3680-$36A7	109	$36A8-$36CF	173	$36D0-$36F7	$36F8-$36FF
$3A80 14976	$5A80 23168	046	$3A80-$3AA7	110	$3AA8-$3ACF	174	$3AD0-$3AF7	$3AF8-$3AFF
$3E80 16000	$5E80 24192	047	$3E80-$3EA7	111	$3EA8-$3ECF	175	$3ED0-$3EF7	$3EF8-$3EFF
$2300 8960	$4300 17152	048	$2300-$2327	112	$2328-$234F	176	$2350-$2377	$2378-$237F
$2700 9984	$4700 18176	049	$2700-$2727	113	$2728-$274F	177	$2750-$2777	$2778-$277F
$2B00 11008	$4B00 19200	050	$2B00-$2B27	114	$2B28-$2B4F	178	$2B50-$2B77	$2B78-$2B7F
$2F00 12032	$4F00 20224	051	$2F00-$2F27	115	$2F28-$2F4F	179	$2F50-$2F77	$2F78-$2F7F
$3300 13056	$5300 21248	052	$3300-$3327	116	$3328-$334F	180	$3350-$3377	$3378-$337F
$3700 14080	$5700 22272	053	$3700-$3727	117	$3728-$374F	181	$3750-$3777	$3778-$377F
$3B00 15104	$5B00 23296	054	$3B00-$3B27	118	$3B28-$3B4F	182	$3B50-$3B77	$3B78-$3B7F
$3F00 16128	$5F00 24320	055	$3F00-$3F27	119	$3F28-$3F4F	183	$3F50-$3F77	$3F78-$3F7F
$2380 9088	$4380 17280	056	$2380-$23A7	120	$23A8-$23CF	184	$23D0-$23F7	$23F8-$23FF
$2780 10112	$4780 18304	057	$2780-$27A7	121	$27A8-$27CF	185	$27D0-$27F7	$27F8-$27FF
$2B80 11136	$4B80 19328	058	$2B80-$2BA7	122	$2BA8-$2BCF	186	$2BD0-$2BF7	$2BF8-$2BFF
$2F80 12160	$4F80 20352	059	$2F80-$2FA7	123	$2FA8-$2FCF	187	$2FD0-$2FF7	$2FF8-$2FFF
$3380 13184	$5380 21376	060	$3380-$33A7	124	$33A8-$33CF	188	$33D0-$33F7	$33F8-$33FF
$3780 14208	$5780 22400	061	$3780-$37A7	125	$37A8-$37CF	189	$37D0-$37F7	$37F8-$37FF
$3B80 15232	$5B80 23424	062	$3B80-$3BA7	126	$3BA8-$3BCF	190	$3BD0-$3BF7	$3BF8-$3BFF
$3F80 16256	$5F80 24448	063	$3F80-$3FA7	127	$3FA8-$3FCF	191	$3FD0-$3FF7	$3FF8-$3FFF

Figure 5.8 HIRES Displayed Memory Map.

SCREEN TOP							
	HORIZONTAL BLANKING (HBL)			HORIZONTAL DISPLAY ENABLE			
LINE NUM	PAGE 1	PAGE 2	11111111 00123456789ABCDEF01234567	PAGE 1	PAGE 2	1111111111111111122222222 0123456789ABCDEF0123456789ABCDEF01234567
0	$2068 8296	$4068 16488	++++++++++####++++++++++++	$2000 8192	$4000 16384	++
1	$2468 9320	$4468 17512	++++++++++####++++++++++++	$2400 9216	$4400 17408	++
2	$2868 10344	$4868 18536	++++++++++####++++++++++++	$2800 10240	$4800 18432	++
3	$2C68 11368	$4C68 19560	++++++++++####++++++++++++	$2C00 11264	$4C00 19456	++
4	$3068 12392	$5068 20584	++++++++++####++++++++++++	$3000 12288	$5000 20480	++
5	$3468 13416	$5468 21608	++++++++++####++++++++++++	$3400 13312	$5400 21504	++
6	$3868 14440	$5868 22632	++++++++++####++++++++++++	$3800 14336	$5800 22528	++
7	$3C68 15464	$5C68 23656	++++++++++####++++++++++++	$3C00 15360	$5C00 23552	++
8	$20E8 8424	$40E8 16616	++++++++++####++++++++++++	$2080 8320	$4080 16512	++
9	$24E8 9448	$44E8 17640	++++++++++####++++++++++++	$2480 9344	$4480 17536	++
10	$28E8 10472	$48E8 18664	++++++++++####++++++++++++	$2880 10368	$4880 18560	++
11	$2CE8 11496	$4CE8 19688	++++++++++####++++++++++++	$2C80 11392	$4C80 19584	++
12	$30E8 12520	$50E8 20712	++++++++++####++++++++++++	$3080 12416	$5080 20608	++
13	$34E8 13544	$54E8 21736	++++++++++####++++++++++++	$3480 13440	$5480 21632	++
14	$38E8 14568	$58E8 22760	++++++++++####++++++++++++	$3880 14464	$5880 22656	++
15	$3CE8 15592	$5CE8 23784	++++++++++####++++++++++++	$3C80 15488	$5C80 23680	++
16	$2168 8552	$4168 16744	++++++++++####++++++++++++	$2100 8448	$4100 16640	++
17	$2568 9576	$4568 17768	++++++++++####++++++++++++	$2500 9472	$4500 17664	++
18	$2968 10600	$4968 18792	++++++++++####++++++++++++	$2900 10496	$4900 18688	++
19	$2D68 11624	$4D68 19816	++++++++++####++++++++++++	$2D00 11520	$4D00 19712	++
20	$3168 12648	$5168 20840	++++++++++####++++++++++++	$3100 12544	$5100 20736	++
21	$3568 13672	$5568 21864	++++++++++####++++++++++++	$3500 13568	$5500 21760	++
22	$3968 14696	$5968 22888	++++++++++####++++++++++++	$3900 14592	$5900 22784	++
23	$3D68 15720	$5D68 23912	++++++++++####++++++++++++	$3D00 15616	$5D00 23808	++
24	$21E8 8680	$41E8 16872	++++++++++####++++++++++++	$2180 8576	$4180 16768	++
25	$25E8 9704	$45E8 17896	++++++++++####++++++++++++	$2580 9600	$4580 17792	++
26	$29E8 10728	$49E8 18920	++++++++++####++++++++++++	$2980 10624	$4980 18816	++
27	$2DE8 11752	$4DE8 19944	++++++++++####++++++++++++	$2D80 11648	$4D80 19840	++
28	$31E8 12776	$51E8 20968	++++++++++####++++++++++++	$3180 12672	$5180 20864	++
29	$35E8 13800	$55E8 21992	++++++++++####++++++++++++	$3580 13696	$5580 21888	++
30	$39E8 14824	$59E8 23016	++++++++++####++++++++++++	$3980 14720	$5980 22912	++
31	$3DE8 15848	$5DE8 24040	++++++++++####++++++++++++	$3D80 15744	$5D80 23936	++
32	$2268 8808	$4268 17000	++++++++++####++++++++++++	$2200 8704	$4200 16896	++
33	$2668 9832	$4668 18024	++++++++++####++++++++++++	$2600 9728	$4600 17920	++
34	$2A68 10856	$4A68 19048	++++++++++####++++++++++++	$2A00 10752	$4A00 18944	++
35	$2E68 11880	$4E68 20072	++++++++++####++++++++++++	$2E00 11776	$4E00 19968	++
36	$3268 12904	$5268 21096	++++++++++####++++++++++++	$3200 12800	$5200 20992	++
37	$3668 13928	$5668 22120	++++++++++####++++++++++++	$3600 13824	$5600 22016	++
38	$3A68 14952	$5A68 23144	++++++++++####++++++++++++	$3A00 14848	$5A00 23040	++
39	$3E68 15976	$5E68 24168	++++++++++####++++++++++++	$3E00 15872	$5E00 24064	++
40	$22E8 8936	$42E8 17128	++++++++++####++++++++++++	$2280 8832	$4280 17024	++
41	$26E8 9960	$46E8 18152	++++++++++####++++++++++++	$2680 9856	$4680 18048	++
42	$2AE8 10984	$4AE8 19176	++++++++++####++++++++++++	$2A80 10880	$4A80 19072	++
43	$2EE8 12008	$4EE8 20200	++++++++++####++++++++++++	$2E80 11904	$4E80 20096	++
44	$32E8 13032	$52E8 21224	++++++++++####++++++++++++	$3280 12928	$5280 21120	++
45	$36E8 14056	$56E8 22248	++++++++++####++++++++++++	$3680 13952	$5680 22144	++
46	$3AE8 15080	$5AE8 23272	++++++++++####++++++++++++	$3A80 14976	$5A80 23168	++
47	$3EE8 16104	$5EE8 24296	++++++++++####++++++++++++	$3E80 16000	$5E80 24192	++
48	$2368 9064	$4368 17256	++++++++++####++++++++++++	$2300 8960	$4300 17152	++
49	$2768 10088	$4768 18280	++++++++++####++++++++++++	$2700 9984	$4700 18176	++
50	$2B68 11112	$4B68 19304	++++++++++####++++++++++++	$2B00 11008	$4B00 19200	++
51	$2F68 12136	$4F68 20328	++++++++++####++++++++++++	$2F00 12032	$4F00 20224	++
52	$3368 13160	$5368 21352	++++++++++####++++++++++++	$3300 13056	$5300 21248	++
53	$3768 14184	$5768 22376	++++++++++####++++++++++++	$3700 14080	$5700 22272	++
54	$3B68 15208	$5B68 23400	++++++++++####++++++++++++	$3B00 15104	$5B00 23296	++
55	$3F68 16232	$5F68 24424	++++++++++####++++++++++++	$3F00 16128	$5F00 24320	++
56	$23E8 9192	$43E8 17384	++++++++++####++++++++++++	$2380 9088	$4380 17280	++
57	$27E8 10216	$47E8 18408	++++++++++####++++++++++++	$2780 10112	$4780 18304	++
58	$2BE8 11240	$4BE8 19432	++++++++++####++++++++++++	$2B80 11136	$4B80 19328	++
59	$2FE8 12264	$4FE8 20456	++++++++++####++++++++++++	$2F80 12160	$4F80 20352	++
60	$33E8 13288	$53E8 21480	++++++++++####++++++++++++	$3380 13184	$5380 21376	++
61	$37E8 14312	$57E8 22504	++++++++++####++++++++++++	$3780 14208	$5780 22400	++
62	$3BE8 15336	$5BE8 23528	++++++++++####++++++++++++	$3B80 15232	$5B80 23424	++
63	$3FE8 16360	$5FE8 24552	++++++++++####++++++++++++	$3F80 16256	$5F80 24448	++

Figure 5.9 HIRES Video Scanning Map. (1 of 4)

SCREEN MIDDLE						
	HORIZONTAL BLANKING (HBL)			HORIZONTAL DISPLAY ENABLE		
LINE NUM	PAGE 1	PAGE 2	11111111 00123456789ABCDEF01234567	PAGE 1	PAGE 2	1111111111111111122222222 0123456789ABCDEF0123456789ABCDEF01234567
64	$2010 8208	$4010 16400	+++++++++########++++++++++++++	$2028 8232	$4028 16424	++
65	$2410 9232	$4410 17424	+++++++++########++++++++++++++	$2428 9256	$4428 17448	++
66	$2810 10256	$4810 18448	+++++++++########++++++++++++++	$2828 10280	$4828 18472	++
67	$2C10 11280	$4C10 19472	+++++++++########++++++++++++++	$2C28 11304	$4C28 19496	++
68	$3010 12304	$5010 20496	+++++++++########++++++++++++++	$3028 12328	$5028 20520	++
69	$3410 13328	$5410 21520	+++++++++########++++++++++++++	$3428 13352	$5428 21544	++
70	$3810 14352	$5810 22544	+++++++++########++++++++++++++	$3828 14376	$5828 22568	++
71	$3C10 15376	$5C10 23568	+++++++++########++++++++++++++	$3C28 15400	$5C28 23592	++
72	$2090 8336	$4090 16528	+++++++++########++++++++++++++	$20A8 8360	$40A8 16552	++
73	$2490 9360	$4490 17552	+++++++++########++++++++++++++	$24A8 9384	$44A8 17576	++
74	$2890 10384	$4890 18576	+++++++++########++++++++++++++	$28A8 10408	$48A8 18600	++
75	$2C90 11408	$4C90 19600	+++++++++########++++++++++++++	$2CA8 11432	$4CA8 19624	++
76	$3090 12432	$5090 20624	+++++++++########++++++++++++++	$30A8 12456	$50A8 20648	++
77	$3490 13456	$5490 21648	+++++++++########++++++++++++++	$34A8 13480	$54A8 21672	++
78	$3890 14480	$5890 22672	+++++++++########++++++++++++++	$38A8 14504	$58A8 22696	++
79	$3C90 15504	$5C90 23696	+++++++++########++++++++++++++	$3CA8 15528	$5CA8 23720	++
80	$2110 8464	$4110 16656	+++++++++########++++++++++++++	$2128 8488	$4128 16680	++
81	$2510 9488	$4510 17680	+++++++++########++++++++++++++	$2528 9512	$4528 17704	++
82	$2910 10512	$4910 18704	+++++++++########++++++++++++++	$2928 10536	$4928 18728	++
83	$2D10 11536	$4D10 19728	+++++++++########++++++++++++++	$2D28 11560	$4D28 19752	++
84	$3110 12560	$5110 20752	+++++++++########++++++++++++++	$3128 12584	$5128 20776	++
85	$3510 13584	$5510 21776	+++++++++########++++++++++++++	$3528 13608	$5528 21800	++
86	$3910 14608	$5910 22800	+++++++++########++++++++++++++	$3928 14632	$5928 22824	++
87	$3D10 15632	$5D10 23824	+++++++++########++++++++++++++	$3D28 15656	$5D28 23848	++
88	$2190 8592	$4190 16784	+++++++++########++++++++++++++	$21A8 8616	$41A8 16808	++
89	$2590 9616	$4590 17808	+++++++++########++++++++++++++	$25A8 9640	$45A8 17832	++
90	$2990 10640	$4990 18832	+++++++++########++++++++++++++	$29A8 10664	$49A8 18856	++
91	$2D90 11664	$4D90 19856	+++++++++########++++++++++++++	$2DA8 11688	$4DA8 19880	++
92	$3190 12688	$5190 20880	+++++++++########++++++++++++++	$31A8 12712	$51A8 20904	++
93	$3590 13712	$5590 21904	+++++++++########++++++++++++++	$35A8 13736	$55A8 21928	++
94	$3990 14736	$5990 22928	+++++++++########++++++++++++++	$39A8 14760	$59A8 22952	++
95	$3D90 15760	$5D90 23952	+++++++++########++++++++++++++	$3DA8 15784	$5DA8 23976	++
96	$2210 8720	$4210 16912	+++++++++########++++++++++++++	$2228 8744	$4228 16936	++
97	$2610 9744	$4610 17936	+++++++++########++++++++++++++	$2628 9768	$4628 17960	++
98	$2A10 10768	$4A10 18960	+++++++++########++++++++++++++	$2A28 10792	$4A28 18984	++
99	$2E10 11792	$4E10 19984	+++++++++########++++++++++++++	$2E28 11816	$4E28 20008	++
100	$3210 12816	$5210 21008	+++++++++########++++++++++++++	$3228 12840	$5228 21032	++
101	$3610 13840	$5610 22032	+++++++++########++++++++++++++	$3628 13864	$5628 22056	++
102	$3A10 14864	$5A10 23056	+++++++++########++++++++++++++	$3A28 14888	$5A28 23080	++
103	$3E10 15888	$5E10 24080	+++++++++########++++++++++++++	$3E28 15912	$5E28 24104	++
104	$2290 8848	$4290 17040	+++++++++########++++++++++++++	$22A8 8872	$42A8 17064	++
105	$2690 9872	$4690 18064	+++++++++########++++++++++++++	$26A8 9896	$46A8 18088	++
106	$2A90 10896	$4A90 19088	+++++++++########++++++++++++++	$2AA8 10920	$4AA8 19112	++
107	$2E90 11920	$4E90 20112	+++++++++########++++++++++++++	$2EA8 11944	$4EA8 20136	++
108	$3290 12944	$5290 21136	+++++++++########++++++++++++++	$32A8 12968	$52A8 21160	++
109	$3690 13968	$5690 22160	+++++++++########++++++++++++++	$36A8 13992	$56A8 22184	++
110	$3A90 14992	$5A90 23184	+++++++++########++++++++++++++	$3AA8 15016	$5AA8 23208	++
111	$3E90 16016	$5E90 24208	+++++++++########++++++++++++++	$3EA8 16040	$5EA8 24232	++
112	$2310 8976	$4310 17168	+++++++++########++++++++++++++	$2328 9000	$4328 17192	++
113	$2710 10000	$4710 18192	+++++++++########++++++++++++++	$2728 10024	$4728 18216	++
114	$2B10 11024	$4B10 19216	+++++++++########++++++++++++++	$2B28 11048	$4B28 19240	++
115	$2F10 12048	$4F10 20240	+++++++++########++++++++++++++	$2F28 12072	$4F28 20264	++
116	$3310 13072	$5310 21264	+++++++++########++++++++++++++	$3328 13096	$5328 21288	++
117	$3710 14096	$5710 22288	+++++++++########++++++++++++++	$3728 14120	$5728 22312	++
118	$3B10 15120	$5B10 23312	+++++++++########++++++++++++++	$3B28 15144	$5B28 23336	++
119	$3F10 16144	$5F10 24336	+++++++++########++++++++++++++	$3F28 16168	$5F28 24360	++
120	$2390 9104	$4390 17296	+++++++++########++++++++++++++	$23A8 9128	$43A8 17320	++
121	$2790 10128	$4790 18320	+++++++++########++++++++++++++	$27A8 10152	$47A8 18344	++
122	$2B90 11152	$4B90 19344	+++++++++########++++++++++++++	$2BA8 11176	$4BA8 19368	++
123	$2F90 12176	$4F90 20368	+++++++++########++++++++++++++	$2FA8 12200	$4FA8 20392	++
124	$3390 13200	$5390 21392	+++++++++########++++++++++++++	$33A8 13224	$53A8 21416	++
125	$3790 14224	$5790 22416	+++++++++########++++++++++++++	$37A8 14248	$57A8 22440	++
126	$3B90 15248	$5B90 23440	+++++++++########++++++++++++++	$3BA8 15272	$5BA8 23464	++
127	$3F90 16272	$5F90 24464	+++++++++########++++++++++++++	$3FA8 16296	$5FA8 24488	++

Figure 5.9 HIRES Video Scanning Map. (2 of 4)

SCREEN BOTTOM						
	HORIZONTAL BLANKING (HBL)			HORIZONTAL DISPLAY ENABLE		
LINE NUM	PAGE 1	PAGE 2	00123456789ABCDEF01234567	PAGE 1	PAGE 2	0123456789ABCDEF0123456789ABCDEF01234567
128	$2038 8248	$4038 16440	++++++++++####++++++++++++	$2050 8272	$4050 16464	++
129	$2438 9272	$4438 17464	++++++++++####++++++++++++	$2450 9296	$4450 17488	++
130	$2838 10296	$4838 18488	++++++++++####++++++++++++	$2850 10320	$4850 18512	++
131	$2C38 11320	$4C38 19512	++++++++++####++++++++++++	$2C50 11344	$4C50 19536	++
132	$3038 12344	$5038 20536	++++++++++####++++++++++++	$3050 12368	$5050 20560	++
133	$3438 13368	$5438 21560	++++++++++####++++++++++++	$3450 13392	$5450 21584	++
134	$3838 14392	$5838 22584	++++++++++####++++++++++++	$3850 14416	$5850 22608	++
135	$3C38 15416	$5C38 23608	++++++++++####++++++++++++	$3C50 15440	$5C50 23632	++
136	$20B8 8376	$40B8 16568	++++++++++####++++++++++++	$20D0 8400	$40D0 16592	++
137	$24B8 9400	$44B8 17592	++++++++++####++++++++++++	$24D0 9424	$44D0 17616	++
138	$28B8 10424	$48B8 18616	++++++++++####++++++++++++	$28D0 10448	$48D0 18640	++
139	$2CB8 11448	$4CB8 19640	++++++++++####++++++++++++	$2CD0 11472	$4CD0 19664	++
140	$30B8 12472	$50B8 20664	++++++++++####++++++++++++	$30D0 12496	$50D0 20688	++
141	$34B8 13496	$54B8 21688	++++++++++####++++++++++++	$34D0 13520	$54D0 21712	++
142	$38B8 14520	$58B8 22712	++++++++++####++++++++++++	$38D0 14544	$58D0 22736	++
143	$3CB8 15544	$5CB8 23736	++++++++++####++++++++++++	$3CD0 15568	$5CD0 23760	++
144	$2138 8504	$4138 16696	++++++++++####++++++++++++	$2150 8528	$4150 16720	++
145	$2538 9528	$4538 17720	++++++++++####++++++++++++	$2550 9552	$4550 17744	++
146	$2938 10552	$4938 18744	++++++++++####++++++++++++	$2950 10576	$4950 18768	++
147	$2D38 11576	$4D38 19768	++++++++++####++++++++++++	$2D50 11600	$4D50 19792	++
148	$3138 12600	$5138 20792	++++++++++####++++++++++++	$3150 12624	$5150 20816	++
149	$3538 13624	$5538 21816	++++++++++####++++++++++++	$3550 13648	$5550 21840	++
150	$3938 14648	$5938 22840	++++++++++####++++++++++++	$3950 14672	$5950 22864	++
151	$3D38 15672	$5D38 23864	++++++++++####++++++++++++	$3D50 15696	$5D50 23888	++
152	$21B8 8632	$41B8 16824	++++++++++####++++++++++++	$21D0 8656	$41D0 16848	++
153	$25B8 9656	$45B8 17848	++++++++++####++++++++++++	$25D0 9680	$45D0 17872	++
154	$29B8 10680	$49B8 18872	++++++++++####++++++++++++	$29D0 10704	$49D0 18896	++
155	$2DB8 11704	$4DB8 19896	++++++++++####++++++++++++	$2DD0 11728	$4DD0 19920	++
156	$31B8 12728	$51B8 20920	++++++++++####++++++++++++	$31D0 12752	$51D0 20944	++
157	$35B8 13752	$55B8 21944	++++++++++####++++++++++++	$35D0 13776	$55D0 21968	++
158	$39B8 14776	$59B8 22968	++++++++++####++++++++++++	$39D0 14800	$59D0 22992	++
159	$3DB8 15800	$5DB8 23992	++++++++++####++++++++++++	$3DD0 15824	$5DD0 24016	++
160	$2238 8760	$4238 16952	++++++++++####++++++++++++	$2250 8784	$4250 16976	++
161	$2638 9784	$4638 17976	++++++++++####++++++++++++	$2650 9808	$4650 18000	++
162	$2A38 10808	$4A38 19000	++++++++++####++++++++++++	$2A50 10832	$4A50 19024	++
163	$2E38 11832	$4E38 20024	++++++++++####++++++++++++	$2E50 11856	$4E50 20048	++
164	$3238 12856	$5238 21048	++++++++++####++++++++++++	$3250 12880	$5250 21072	++
165	$3638 13880	$5638 22072	++++++++++####++++++++++++	$3650 13904	$5650 22096	++
166	$3A38 14904	$5A38 23096	++++++++++####++++++++++++	$3A50 14928	$5A50 23120	++
167	$3E38 15928	$5E38 24120	++++++++++####++++++++++++	$3E50 15952	$5E50 24144	++
168	$22B8 8888	$42B8 17080	++++++++++####++++++++++++	$22D0 8912	$42D0 17104	++
169	$26B8 9912	$46B8 18104	++++++++++####++++++++++++	$26D0 9936	$46D0 18128	++
170	$2AB8 10936	$4AB8 19128	++++++++++####++++++++++++	$2AD0 10960	$4AD0 19152	++
171	$2EB8 11960	$4EB8 20152	++++++++++####++++++++++++	$2ED0 11984	$4ED0 20176	++
172	$32B8 12984	$52B8 21176	++++++++++####++++++++++++	$32D0 13008	$52D0 21200	++
173	$36B8 14008	$56B8 22200	++++++++++####++++++++++++	$36D0 14032	$56D0 22224	++
174	$3AB8 15032	$5AB8 23224	++++++++++####++++++++++++	$3AD0 15056	$5AD0 23248	++
175	$3EB8 16056	$5EB8 24248	++++++++++####++++++++++++	$3ED0 16080	$5ED0 24272	++
176	$2338 9016	$4338 17208	++++++++++####++++++++++++	$2350 9040	$4350 17232	++
177	$2738 10040	$4738 18232	++++++++++####++++++++++++	$2750 10064	$4750 18256	++
178	$2B38 11064	$4B38 19256	++++++++++####++++++++++++	$2B50 11088	$4B50 19280	++
179	$2F38 12088	$4F38 20280	++++++++++####++++++++++++	$2F50 12112	$4F50 20304	++
180	$3338 13112	$5338 21304	++++++++++####++++++++++++	$3350 13136	$5350 21328	++
181	$3738 14136	$5738 22328	++++++++++####++++++++++++	$3750 14160	$5750 22352	++
182	$3B38 15160	$5B38 23352	++++++++++####++++++++++++	$3B50 15184	$5B50 23376	++
183	$3F38 16184	$5F38 24376	++++++++++####++++++++++++	$3F50 16208	$5F50 24400	++
184	$23B8 9144	$43B8 17336	++++++++++####++++++++++++	$23D0 9168	$43D0 17360	++
185	$27B8 10168	$47B8 18360	++++++++++####++++++++++++	$27D0 10192	$47D0 18384	++
186	$2BB8 11192	$4BB8 19384	++++++++++####++++++++++++	$2BD0 11216	$4BD0 19408	++
187	$2FB8 12216	$4FB8 20408	++++++++++####++++++++++++	$2FD0 12240	$4FD0 20432	++
188	$33B8 13240	$53B8 21432	++++++++++####++++++++++++	$33D0 13264	$53D0 21456	++
189	$37B8 14264	$57B8 22456	++++++++++####++++++++++++	$37D0 14288	$57D0 22480	++
190	$3BB8 15288	$5BB8 23480	++++++++++####++++++++++++	$3BD0 15312	$5BD0 23504	++
191	$3FB8 16312	$5FB8 24504	++++++++++####++++++++++++	$3FD0 16336	$5FD0 24528	++

Figure 5.9 HIRES Video Scanning Map. (3 of 4)

VERTICAL BLANKING PERIOD (VBL)						
	HORIZONTAL BLANKING (HBL)			HORIZONTAL DISPLAY ENABLE		
LINE NUM	PAGE 1	PAGE 2	11111111 00123456789ABCDEF01234567	PAGE 1	PAGE 2	1111111111111111222222222 0123456789ABCDEF0123456789ABCDEF01234567
192	$2060 8288	$4060 16480	++++++++++####++++++++++++	$2078 8312	$4078 16504	++
193	$2460 9312	$4460 17504	++++++++++####++++++++++++	$2478 9336	$4478 17528	++
194	$2860 10336	$4860 18528	++++++++++####++++++++++++	$2878 10360	$4878 18552	++
195	$2C60 11360	$4C60 19552	++++++++++####++++++++++++	$2C78 11384	$4C78 19576	++
196	$3060 12384	$5060 20576	++++++++++####++++++++++++	$3078 12408	$5078 20600	++
197	$3460 13408	$5460 21600	++++++++++####++++++++++++	$3478 13432	$5478 21624	++
198	$3860 14432	$5860 22624	++++++++++####++++++++++++	$3878 14456	$5878 22648	++
199	$3C60 15456	$5C60 23648	++++++++++####++++++++++++	$3C78 15480	$5C78 23672	++
200	$20E0 8416	$40E0 16608	++++++++++####++++++++++++	$20F8 8440	$40F8 16632	++
201	$24E0 9440	$44E0 17632	++++++++++####++++++++++++	$24F8 9464	$44F8 17656	++
202	$28E0 10464	$48E0 18656	++++++++++####++++++++++++	$28F8 10488	$48F8 18680	++
203	$2CE0 11488	$4CE0 19680	++++++++++####++++++++++++	$2CF8 11512	$4CF8 19704	++
204	$30E0 12512	$50E0 20704	++++++++++####++++++++++++	$30F8 12536	$50F8 20728	++
205	$34E0 13536	$54E0 21728	++++++++++####++++++++++++	$34F8 13560	$54F8 21752	++
206	$38E0 14560	$58E0 22752	++++++++++####++++++++++++	$38F8 14584	$58F8 22776	++
207	$3CE0 15584	$5CE0 23776	++++++++++####++++++++++++	$3CF8 15608	$5CF8 23800	++
208	$2160 8544	$4160 16736	++++++++++####++++++++++++	$2178 8568	$4178 16760	++
209	$2560 9568	$4560 17760	++++++++++####++++++++++++	$2578 9592	$4578 17784	++
210	$2960 10592	$4960 18784	++++++++++####++++++++++++	$2978 10616	$4978 18808	++
211	$2D60 11616	$4D60 19808	++++++++++####++++++++++++	$2D78 11640	$4D78 19832	++
212	$3160 12640	$5160 20832	++++++++++####++++++++++++	$3178 12664	$5178 20856	++
213	$3560 13664	$5560 21856	++++++++++####++++++++++++	$3578 13688	$5578 21880	++
214	$3960 14688	$5960 22880	++++++++++####++++++++++++	$3978 14712	$5978 22904	++
215	$3D60 15712	$5D60 23904	++++++++++####++++++++++++	$3D78 15736	$5D78 23928	++
216	$21E0 8672	$41E0 16864	++++++++++####++++++++++++	$21F8 8696	$41F8 16888	++
217	$25E0 9696	$45E0 17888	++++++++++####++++++++++++	$25F8 9720	$45F8 17912	++
218	$29E0 10720	$49E0 18912	++++++++++####++++++++++++	$29F8 10744	$49F8 18936	++
219	$2DE0 11744	$4DE0 19936	++++++++++####++++++++++++	$2DF8 11768	$4DF8 19960	++
220	$31E0 12768	$51E0 20960	++++++++++####++++++++++++	$31F8 12792	$51F8 20984	++
221	$35E0 13792	$55E0 21984	++++++++++####++++++++++++	$35F8 13816	$55F8 22008	++
222	$39E0 14816	$59E0 23008	++++++++++####++++++++++++	$39F8 14840	$59F8 23032	++
223	$3DE0 15840	$5DE0 24032	++++++++++####++++++++++++	$3DF8 15864	$5DF8 24056	++
224	$2260 8800	$4260 16992	++++++++++########++++++++	$2278 8824	$4278 17016	##
225	$2660 9824	$4660 18016	++++++++++########++++++++	$2678 9848	$4678 18040	##
226	$2A60 10848	$4A60 19040	++++++++++########++++++++	$2A78 10872	$4A78 19064	##
227	$2E60 11872	$4E60 20064	++++++++++########++++++++	$2E78 11896	$4E78 20088	##
228	$3260 12896	$5260 21088	++++++++++####++++++++++++	$3278 12920	$5278 21112	++
229	$3660 13920	$5660 22112	++++++++++####++++++++++++	$3678 13944	$5678 22136	++
230	$3A60 14944	$5A60 23136	++++++++++####++++++++++++	$3A78 14968	$5A78 23160	++
231	$3E60 15968	$5E60 24160	++++++++++####++++++++++++	$3E78 15992	$5E78 24184	++
232	$22E0 8928	$42E0 17120	++++++++++####++++++++++++	$22F8 8952	$42F8 17144	++
233	$26E0 9952	$46E0 18144	++++++++++####++++++++++++	$26F8 9976	$46F8 18168	++
234	$2AE0 10976	$4AE0 19168	++++++++++####++++++++++++	$2AF8 11000	$4AF8 19192	++
235	$2EE0 12000	$4EE0 20192	++++++++++####++++++++++++	$2EF8 12024	$4EF8 20216	++
236	$32E0 13024	$52E0 21216	++++++++++####++++++++++++	$32F8 13048	$52F8 21240	++
237	$36E0 14048	$56E0 22240	++++++++++####++++++++++++	$36F8 14072	$56F8 22264	++
238	$3AE0 15072	$5AE0 23264	++++++++++####++++++++++++	$3AF8 15096	$5AF8 23288	++
239	$3EE0 16096	$5EE0 24288	++++++++++####++++++++++++	$3EF8 16120	$5EF8 24312	++
240	$2360 9056	$4360 17248	++++++++++####++++++++++++	$2378 9080	$4378 17272	++
241	$2760 10080	$4760 18272	++++++++++####++++++++++++	$2778 10104	$4778 18296	++
242	$2B60 11104	$4B60 19296	++++++++++####++++++++++++	$2B78 11128	$4B78 19320	++
243	$2F60 12128	$4F60 20320	++++++++++####++++++++++++	$2F78 12152	$4F78 20344	++
244	$3360 13152	$5360 21344	++++++++++####++++++++++++	$3378 13176	$5378 21368	++
245	$3760 14176	$5760 22368	++++++++++####++++++++++++	$3778 14200	$5778 22392	++
246	$3B60 15200	$5B60 23392	++++++++++####++++++++++++	$3B78 15224	$5B78 23416	++
247	$3F60 16224	$5F60 24416	++++++++++####++++++++++++	$3F78 16248	$5F78 24440	++
248	$23E0 9184	$43E0 17376	++++++++++####++++++++++++	$23F8 9208	$43F8 17400	++
249	$27E0 10208	$47E0 18400	++++++++++####++++++++++++	$27F8 10232	$47F8 18424	++
250	$2BE0 11232	$4BE0 19424	++++++++++####++++++++++++	$2BF8 11256	$4BF8 19448	++
251	$2FE0 12256	$4FE0 20448	++++++++++####++++++++++++	$2FF8 12280	$4FF8 20472	++
252	$33E0 13280	$53E0 21472	++++++++++####++++++++++++	$33F8 13304	$53F8 21496	++
253	$37E0 14304	$57E0 22496	++++++++++####++++++++++++	$37F8 14328	$57F8 22520	++
254	$3BE0 15328	$5BE0 23520	++++++++++####++++++++++++	$3BF8 15352	$5BF8 23544	++
255	$3FE0 16352	$5FE0 24544	++++++++++####++++++++++++	$3FF8 16376	$5FF8 24568	++
256	$2260 11232	$4BE0 19424	++++++++++####++++++++++++	$2BF8 11256	$4BF8 19448	++
257	$2FE0 12256	$4FE0 20448	++++++++++####++++++++++++	$2FF8 12280	$4FF8 20472	++
258	$33E0 13280	$53E0 21472	++++++++++####++++++++++++	$33F8 13304	$53F8 21496	++
259	$37E0 14304	$57E0 22496	++++++++++####++++++++++++	$37F8 14328	$57F8 22520	++
260	$3BE0 15328	$5BE0 23520	++++++++++####++++++++++++	$3BF8 15352	$5BF8 23544	++
261	$3FE0 16352	$5FE0 24544	++++++++++####++++++++++++	$3FF8 16376	$5FF8 24568	++

Figure 5.9 HIRES Video Scanning Map. (4 of 4)

Table 5.2 Screen Memory Usage Summary.

	TEXT/LORES		HIRES	
LOCATION	HBL	HBL'	HBL	HBL'
SCREEN TOP	(Last 16 of THIRD 40 and UNUSED 8) PLUS $1000	FIRST 40	Last 16 of THIRD 40 and UNUSED 8	FIRST 40
SCREEN MIDDLE	(Last 24 of FIRST 40) PLUS $1000	SECOND 40	Last 24 of FIRST 40	SECOND 40
SCREEN BOTTOM	(Last 24 of SECOND 40) PLUS $1000	THIRD 40	Last 24 of SECOND 40	THIRD 40
VBL	(Last 24 of THIRD 40) PLUS $1000	UNUSED 8 and First 32 of FIRST 40	Last 24 of THIRD 40	UNUSED 8 and First 32 of FIRST 40

The following is a reference list for scanned memory in the HIRES MIXED mode:

Line 0, HPE' + 1 thru Line 160, HPE' — HIRES, Figure 5.9
Line 160, HPE' + 1 thru Line 192, HPE' — TEXT, Figure 5.6
Line 192, HPE' + 1 thru Line 224, HPE' — HIRES, Figure 5.9
Line 224, HPE' + 1 thru Line 0, HPE' — TEXT, Figure 5.6

HPE' occurs during the first video scanner state of HBL. On the screen, the address switching point for HIRES MIXED mode comes just at the end of the display on the right side.

REFRESHING RAM IN THE APPLE

The refresh requirement of 16K dynamic RAM is that every ROW address be accessed at least once every two milliseconds. To achieve refresh in the process of scanning RAM for video output, the address inputs to RAM had to be assigned very carefully. This assignment was made more complex in the Apple design by the capability of using 4K or 16K RAM chips.

If the Apple had been originally designed to operate with only 16K RAM chips, the RAM refresh requirement could easily have been met by assigning H0, H1, H2, SUM-A3, SUM-A4, SUM-A5, and V0 to the RAM ROW addresses. The 4K/16K capability adds a new requirement, however, which isn't met by these assignments. Pin 13 is RA6 on a 16K RAM chip and chip select on a 4K RAM chip. In the 4K configuration, A12 and A13 select among rows C, D, and E via pin 13. This dictates that A12 and A13 are also assigned to pin 13 in the 16K configuration. In other words, A12 and A13 must be assigned to the ROW and COLUMN address of RA6. A13 is equivalent to HIRES PAGE 1 during scanner access. This is not acceptable for a ROW address, because it doesn't change with the video scan. The natural equivalent of A12 is HIRES • VC. This is not acceptable for a ROW address, because it doesn't change in TEXT or LORES mode. The Apple designer made A12 work as a ROW assignment by changing its scanner equivalent to HIRES • VC + TEXT/LORES • HBL. HBL changes during TEXT/LORES and meets the refresh requirements. During HBL the "wrong" addresses are selected, but no harm is done because the screen is blanked during HBL.

Table 5.3 The Video Scanner Row Address Assignments.

RAM Address	Scanner Input
RA0	V0
RA1	H2
RA2	H0
RA3	V1
RA4	SUM-A3
RA5	H1
RA6	HIRES • VC + TEXT/LORES • HBL

The other ROW address assignments are dictated by the use of HBL as a refresh bit. H0, H1, H2, and SUM-A3 run through a 16-state sequence at least once every time HBL is high or low. SUM-A4, SUM-A5, and SUM-A6 are unacceptable, because they do not make a complete sequence every time HBL is high. The remaining two ROW address assignments are V0 and V1.

In HIRES mode, VC is a refresh input but VA and VB aren't. For every state of V1-V0-VC, all the lower refresh bits go through their counts four straight times (once for each state of VB-VA). This means that the maximum time between refresh states is no more than 29 horizontal scans (32-3) or 1.85 milliseconds. In TEXT/LORES mode, VA, VB, and VC are not refresh bits, so the maximum time between refresh states is 25 horizontal scans (32-7) or 1.59 milliseconds. In both modes, the 2 millisecond maximum is met.

RAM ADDRESS MULTIPLEXOR HARDWARE

Figure 5.10 is a schematic diagram of the RAM address multiplexor. The multiplexor is shown jumpered for 16K, but Revision 7 and later Apples are wired this way without jumpers. Figure 5.10 does not show the circuits required for 4K operation.

In older Apples configured for 16K RAM chips, the 74LS139 at E2 serves no purpose and may be used as a spare. There is no chip at E2 on newer Apples. The following circuits serve no purpose in older 16K configured Apples and are not wired in newer Apples:

LOCATION	TYPE
E2	LS139
J1 sections a & b	LS257
H1 pins 1,2,3,11,12, & 13	LS08
C14 pins 1,2, & 3	LS32

Owners may dream up their own modifications to utilize these circuits.

In the address multiplexor, 4 to 1 multiplexing is accomplished in LS153s, while 2 to 1 multiplexing is accomplished in LS257s. The horizontal address offset is generated by an LS283 adder. CAS' and RAM SELECT' are decoded in an LS139.

Propagation delays in the CAS' and RAM SELECT' lines are important. CAS' falling is delayed to RAM by the high to low propagation of the LS139 at F2. This is listed as 21 nanoseconds typical and 32 nanoseconds maximum by the Texas Instruments *TTL Data Book*. This delay is 25 nanoseconds in the author's Apple. The significance here is that RAS' rises only 178 nanoseconds after CAS' falls at the RAM chips (210 minus 32 equals 178). The TCAC specification (maximum time from CAS' to read data valid) should be less than 178 nanoseconds in RAM chips used in the Apple II.

A second important delay period is the time from PHASE 0 falling until RAM data becomes valid on the data bus after an MPU access to an address above $C00F. This propagation delay represents the time after PHASE 0 falls, by which any device other than RAM must stop controlling the data bus. The chain is listed below:

LOCATION	TYPE	SIGNAL PATH	TYPICAL	MAXIMUM
J1	LS257	pin 1 to pins 9,12	14 nsec	21 nsec
F2	LS139	pins 14,13 to pins 10,11,12	26 nsec	39 nsec
D2	LS20	pins 9,10,12 to pin 8	9 nsec	15 nsec
A2	LS00	pin 5 to pin 6	10 nsec	15 nsec
B7,B6	LS257	pin 15 to pins 4,7,9,12	20 nsec	30 nsec
		TOTAL	79 nsec	120 nsec

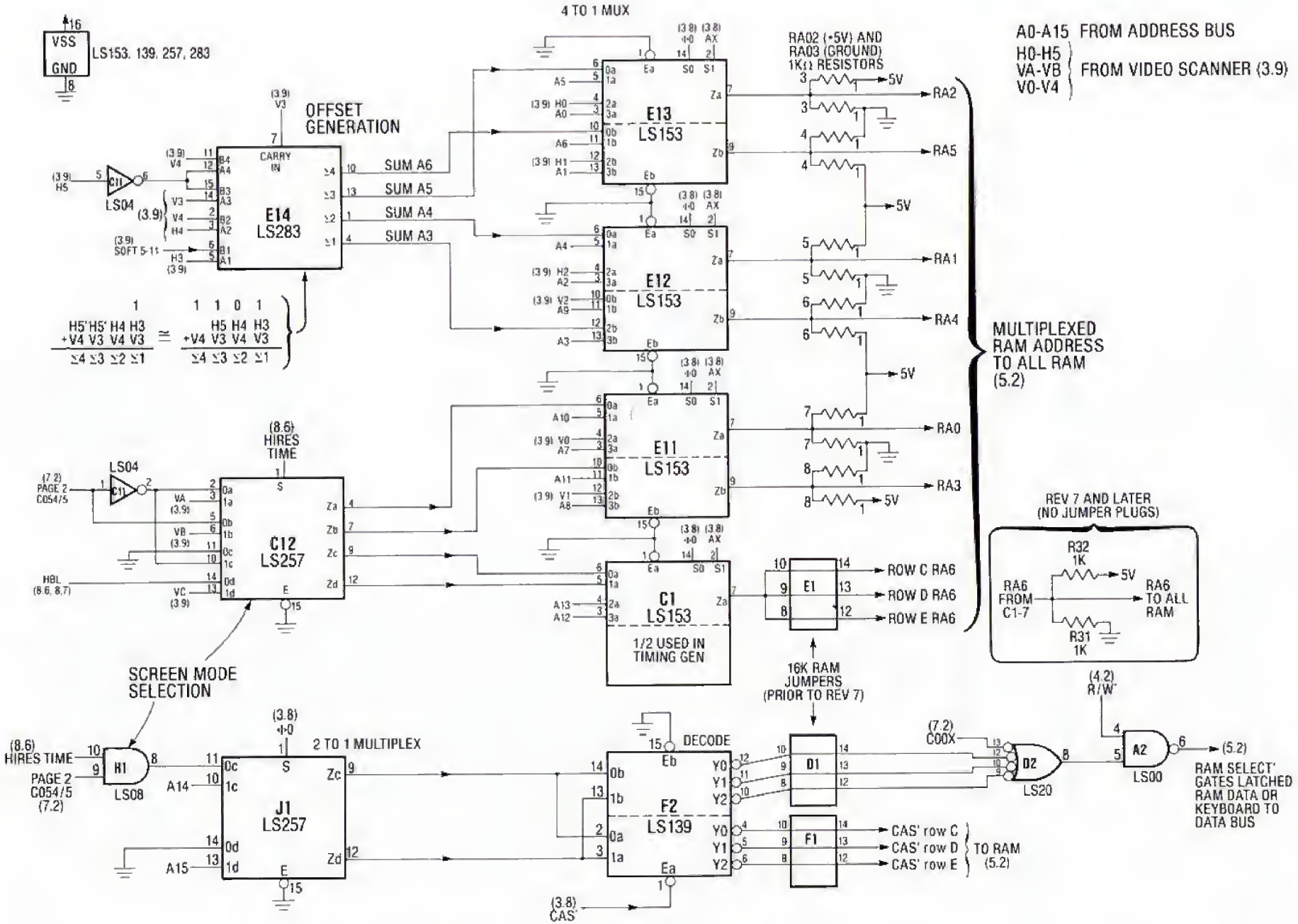


Figure 5.10 Schematic: The Address Multiplexor.

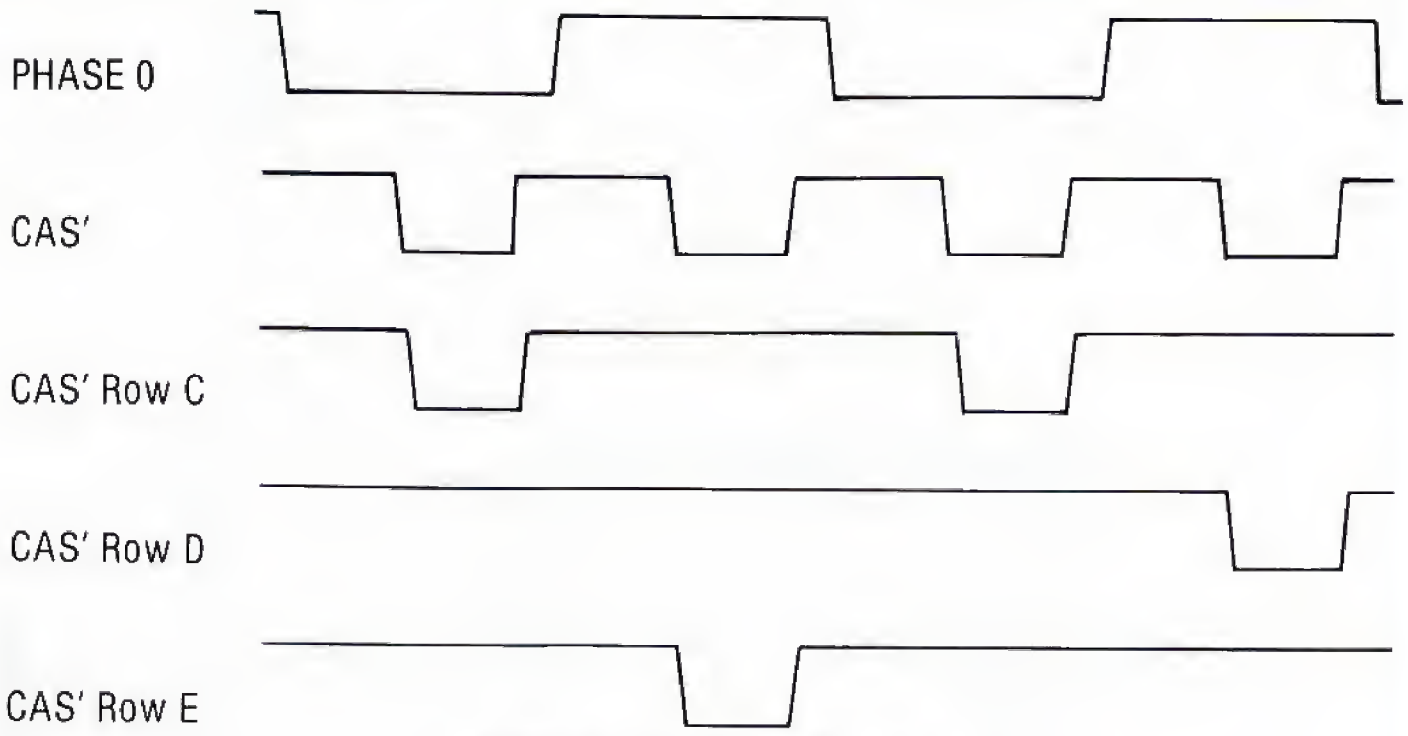


Figure 5.11 CAS' Signal Example.

This time delay is 60 nanoseconds in the author's Apple.

A very interesting feature of Apple RAM connections is the termination resistors on the multiplexed RAM address lines, RA0-RA6 on the right side of Figure 5.10. No RAM data sheet recommends pull-up and pull-down resistors on the address inputs, so why are they there? They are there because Steve Wozniak found that stringing large numbers of RAM chips together can cause reflections on the RAM address lines. Presumably, this is because the inductance of the conductors and the capacitance at the RAM chip inputs give the RAM address lines a characteristic impedance, just like television coaxial cable has a characteristic impedance of 75 ohms. A coax cable or other distributed reactance transmission line needs to be terminated by an impedance equal to its characteristic impedance, or reflections (voltage pulses traveling backwards) will occur. In the Apple, these reflections were causing RAM access to be unreliable. Wozniak is not a transmission line expert, but he determined experimentally that 1000 ohm pull-up and pull-down resistors would make RAM access reliable. He says that this was later verified to be the theoretically correct termination. The RA6 distribution path in older Apples was such that reflections were not a problem. In Revision 7, RA6 distribution was made similar to the other RAM address lines, and a pair of termination resistors for RA6 was added.

RAM TIMING IN THE APPLE II

Most aspects of RAM timing have already been covered in various other related discussions. The intention here is to reinforce the timing details of basic RAM access.

One area of importance is the gating of CAS' to the three rows of RAM. Figure 5.11 shows the last two cycles of the following instruction:

```
9000:   LDA $5000 ;APPLE IS
                  ;CURRENTLY IN
                  ;TEXT MODE.
```

The instruction is stored at $9000 through $9002, so instructions are being fetched from RAM row E. The Apple is in TEXT mode, so the video scanner is accessing row C. The instruction is loading from $5000, so on the last cycle, the MPU accesses row D. The operation is really very simple. CAS' at any RAM row goes low when that row is addressed and when CAS' from the timing generator goes low.

Figure 5.12 shows some basic details of a 6502/RAM **read** cycle. There are several important points illustrated here. For one thing, even though read data from the RAM chips is only valid for a limited period, it is held valid on the data bus for nearly half a microsecond by the RAM latch. When a program stored in RAM is being executed, RAM output data is continuously controlling the data bus

except during and after write cycles or access to addresses above $BFFF. Another interesting point is that while the MPU addresses RAM during PHASE 0, data from the MPU access is present on the data bus during PHASE 1, roughly speaking. Similarly, the video data is present roughly during PHASE 0. As Figure 5.12 shows, PHASE 0 rising is a good clockpulse for clocking video data to peripheral cards during read cycles.

It goes without saying that data from the MPU read access is present on the data bus when 6502 PHASE 2 falls. This is required of all read responses to the 6502, whether the responding device is RAM, ROM, or an input port.

The order of important events in Figure 5.12 is:

1. PHASE 2 falls, clocking the data transfer of the previous 6502 machine cycle.
2. PHASE 1 • AX enables the video ROW address to RAM, and the ROW address is clocked to RAM by RAS' falling.
3. PHASE 1 • AX' enables the video COLUMN address to RAM and the COLUMN address is clocked to one row of Apple RAM by CAS' falling.
4. Video data becomes valid at the output of RAM. The time at which this occurs will depend upon the speed of the RAM chips being used in a given Apple. TCAC (Time - CAS' ACcess) should be shorter than 178 nanoseconds in RAM chips used in the Apple. In all Apples, the RAM data must become valid before RAS' rises and latches the RAM data.
5. The data is latched by RAS' rising.
6. About 35 nanoseconds after RAS' rises, the

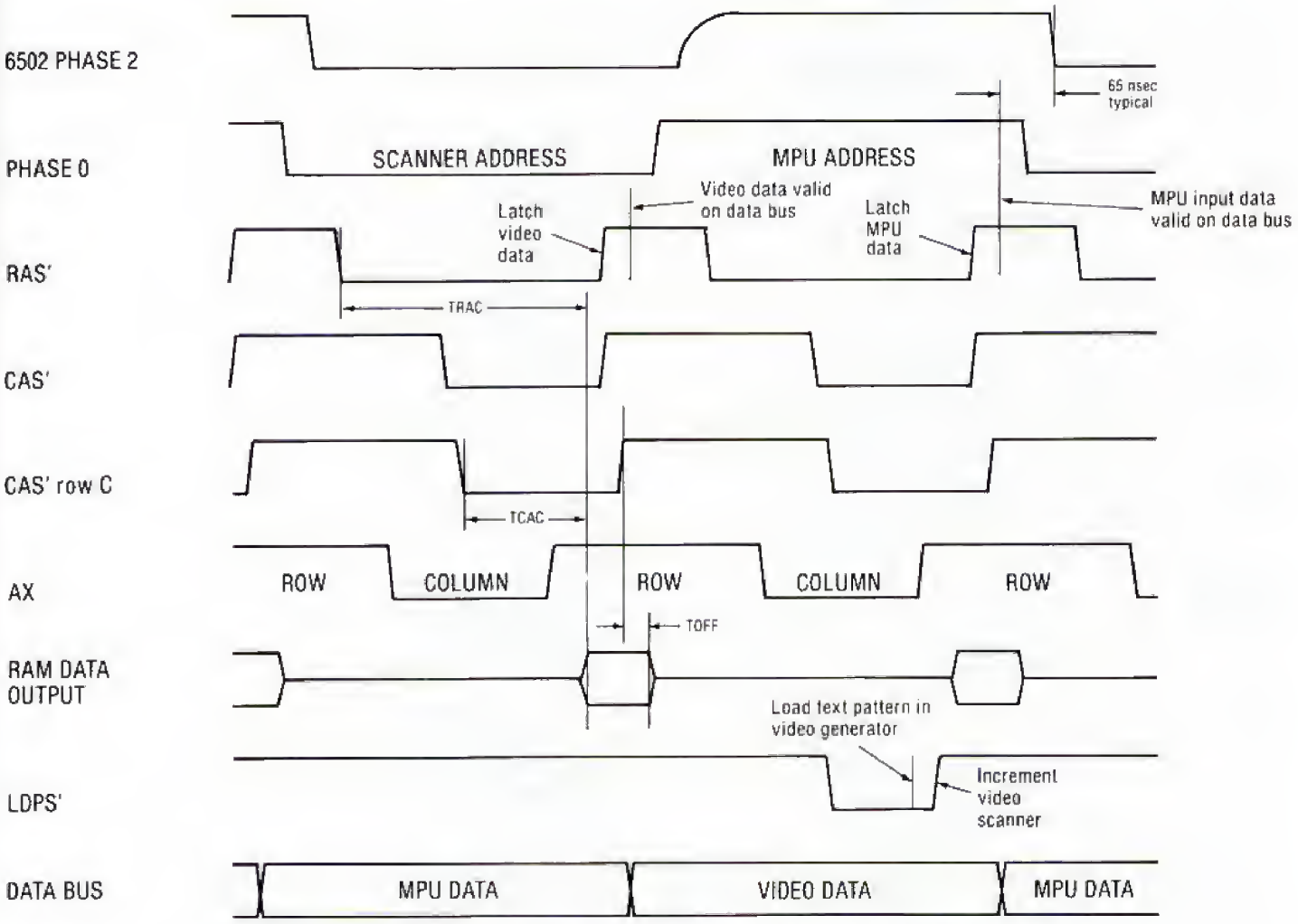


Figure 5.12 Timing Example: A 6502 Read Cycle to Address $1000.

video data becomes valid on the data bus. The delay is due to propagation delay in the RAM latch (LS174) and RAM/keyboard multiplexor (LS257). The latched RAM data is held on the data bus throughout Figure 5.12, because both the video scanner and the MPU are reading from RAM.

7. During PHASE 0, the MPU addresses RAM in the same way the video scanner did during PHASE 1, with AX selecting ROW or COLUMN and RAS' and CAS' clocking the address to RAM.
8. LDPS' drops low during PHASE 0. This is the signal which loads TEXT patterns in the video generator. The TEXT patterns are determined by the current state of the RAM latch and by the current state of VA, VB, and VC.
9. Just as in the scanner access, RAM data from the MPU access becomes valid, is latched, and is propagated to the data bus.
10. The RAM read data is clocked to the MPU by PHASE 2 falling.

Figure 5.13 shows some basic details of a 6502/RAM **write** cycle. The RAS', AX, CAS' address control is identical to a read cycle. The primary difference between the read cycle and write cycle is in data bus management. Also, the features of data bus management during write cycles are the same whether the 6502 is writing to RAM addresses or to any other addresses.

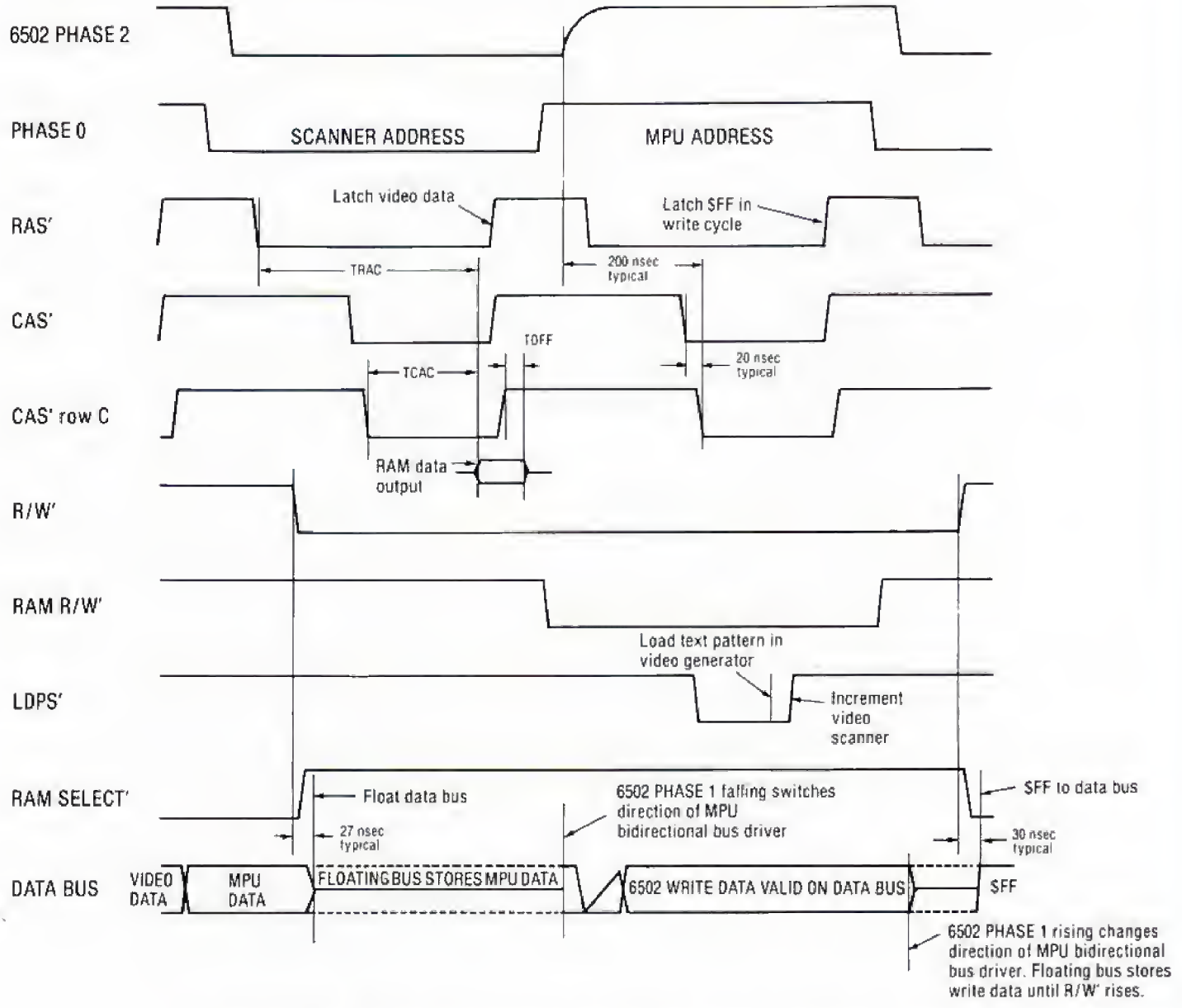


Figure 5.13 Timing Example: A 6502 Write Cycle to Address $1000.

In the write cycle, the video scanner makes its read access to RAM during PHASE 1 just as in the read cycle. Nothing interrupts the scanner access to RAM in an unmodified Apple. The latched video data never gets to the data bus in a write cycle, however. While R/W' is low, the MPU (or DMA peripheral) owns the data bus.

Excluding the scanner access which is the same as in a read cycle, the order of important events in a write cycle is as follows:

1. PHASE 2 falls, clocking the data transfer of the previous 6502 cycle.
2. The R/W' line drops low at the same time the 6502 address becomes valid (about 100 nanoseconds after PHASE 2 falls). This forces RAM SELECT' high which disables the output of the RAM/keyboard multiplexor. No other device takes control of the data bus at this time, so the latched data from the previous MPU cycle remains valid on the data bus (because of the long bleed off time of data when the Apple's data bus is floated).
3. PHASE 0 rises, enabling MPU addressing at the RAM address multiplexor and enabling R/W' to drop low at the RAM chips. The RAS', AX, CAS' addressing sequence is identical to the read cycle sequence.
4. 6502 PHASE 1 falls, causing the MPU bidirectional bus driver to take control of the data bus. The 6502 write data is not yet valid, and until it becomes valid, the data bus in the author's Apple stays low if it was low and oscillates down then up if it was high. The state of the data bus during this time period is not significant.
5. The 6502 write data becomes valid before CAS' falls at the RAM chips. The Apple write cycle is what is referred to in RAM literature as an early write cycle (as opposed to a read/write cycle). R/W' drops low before CAS', and write data must be valid at the RAM input before CAS' drops.
6. RAS' rises, latching $FF in the RAM data latch. During the write cycle, the output of RAM floats. The response of the data latch to the floating input is to bring its output lines high when RAS' rises.
7. 6502 PHASE 1 rises, reversing the direction of the bidirectional bus driver and floating the data bus. The 6502 write data remains valid on the floating data bus.
8. R/W' goes high, causing the RAM SELECT' signal to go low. This places the latched data ($FF) on the data bus. $FF will remain on the data bus until data from the current video access is latched. A write cycle is always followed by a period of time with $FF on the data bus.

The RAM SELECT' term is the primary data bus management signal in the Apple. When the MPU is not accessing RAM, RAM SELECT' defines when the data bus is available for other devices to respond to RAM. This timing is illustrated in Figure 5.14, which shows a read to $C010, the keyboard strobe reset address.

$C010 is one of the Apple addresses which causes the data bus to float. The floating data bus stores the video data from the scanner access so that it can be read by the MPU. A read to a data responding address like $F800 (ROM) will have very similar timing, except that at some time during the floating bus period, the responding device will take control of the data bus. The main order of events in Figure 5.14 is as follows:

1. The scanner accesses RAM during PHASE 1.
2. About 35 nanoseconds after RAS' rises, the latched data from the video access is on the data bus.
3. PHASE 0 rises, routing the MPU address through the address multiplexor. Since the MPU is not addressing RAM or the keyboard, the RAM SELECT' goes high, floating the data bus. It takes about 60 nanoseconds for RAM SELECT' to go high after PHASE 0 rising, because the logic must be propagated through four LSTTL devices.
4. After RAM SELECT' goes high, the data bus floats, storing the latched video data. If a data responding device had been addressed, it could take control of the data bus anytime during this floating period.
5. RAS' rises and latches $FF at the RAM data latch. The RAM data output is floating when RAS' rises, because CAS' does not fall at any of the three rows of Apple RAM. As in a write cycle, the floating RAM output is interpreted as $FF by the latch.
6. PHASE 0 falls, causing the scanner to address RAM.
7. The 6502 PHASE 2 clock falls, clocking video data to the MPU.
8. RAM SELECT' drops low about 60 nanoseconds after PHASE 0. After a further short delay, the latched RAM data ($FF) is gated to the data bus. As in a write cycle, any MPU access to an address above $BFFF is followed by a period of time with $FF on the data bus.

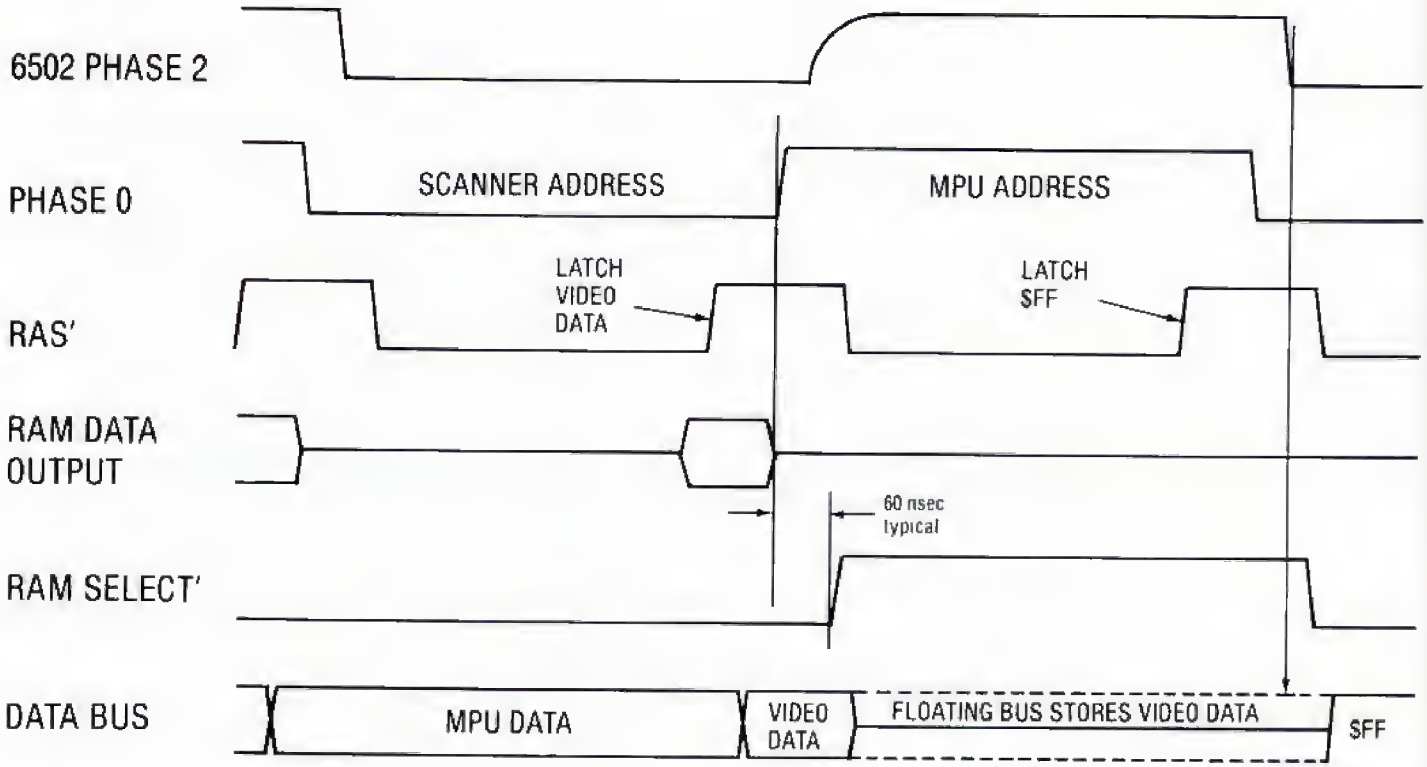


Figure 5.14 Timing Example: A 6502 Read Cycle to Address $C010.

THE 16K RAM CARD

There are two common operational philosophies encountered in today's microcomputers. One is to have an extensive operating system, including a BASIC interpreter in ROM. These computers come up in BASIC and may use disk or cassette or both for program and data storage. The other philosophy is to have a strictly disk based system with little firmware beyond a bootstrap program which loads operating systems from disk to RAM. This philosophy yields far more versatility than the other for the obvious reason that any available operating system may be loaded, giving the computer a completely different personality. The Apple computer is of the former philosophy with a string attached. The string is that installation of a **16K RAM card** converts it to a computer of the latter philosophy.

A card in any peripheral slot can inhibit motherboard ROM and respond, itself, to addresses $D000-$FFFF. 16K RAM cards use this feature to steal the 12K addressing range from ROM and use it for addressing expansion RAM. Use of 16K chips creates an excess of 4K of RAM which is utilized by bank switching the $D000-$DFFF range. Installation of the 16K RAM card gives the Apple the full versatility of a 60K disk based computer with the added features of 4K of bank switched auxiliary RAM and 12K of instantaneously available firmware.

The 16K RAM card was originally released as part of the "Language System", which included the RAM card and Pascal, Integer BASIC, and Applesoft BASIC on disk. 16K RAM cards eventually became available as stand alone products from a number of sources, and the card now supports several languages as well as CP/M when used in conjunction with a Z80 MPU card.

This section will cover the operational features of the 16K RAM card produced by Apple; however, probably all 16K RAM cards made for the Apple will operate the same way. The basis for these discussions will be the schematic diagram of Figure 5.15. This drawing was prepared by the author after study of a "Language System" RAM card and of an imitation purchased in Tokyo. The imitation turned out to be identical in all but a few details.

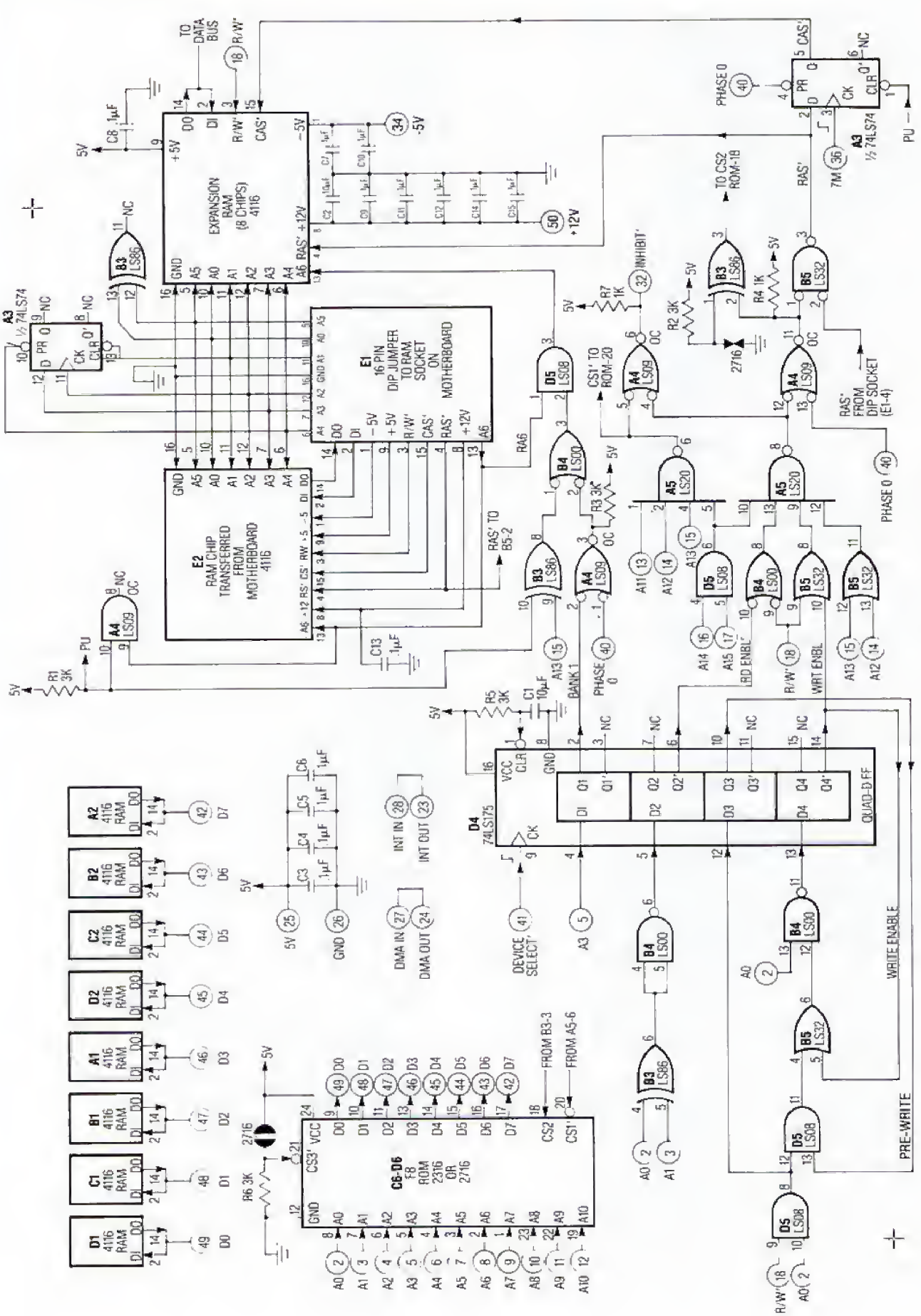


Figure 5.15 Schematic: The 16K RAM Card.

Apple's 16K RAM card requires the owner to move a RAM chip from the motherboard to the RAM card and connect a 16-pin DIP jumper between the card and the vacated motherboard socket. This was Apple's way of getting around the necessity of developing RAS' and ROW/COLUMN multiplexed addresses on the card. RAM timing signals are not available at the peripheral slots, and developing equivalent signals is made difficult by the absence of 14M at the slots. Rather than tackle the problem of developing timing signals and multiplexing the address bus from 14 to 7 lines, Apple bypassed the problem by connecting the jumper to the motherboard RAM socket. This probably lowered the cost of producing the hardware, although you wouldn't have guessed it based on the retail price of the "Language System."

The RAM chip that is transferred from the motherboard has all of its pins connected to pins of the DIP jumper socket. This means that the transferred chip performs the same function it did on the motherboard and that any RAM chip may be chosen for transfer as long as the jumper reaches the vacated socket. RA0-RA6 and RAS' are also distributed to other points from the DIP jumper. RA0-RA5 are connected directly to their respective pins on all eight expansion RAM chips. RAS' and RA6 are gated to their respective pins on the eight chips during PHASE 1 with CAS' held high. This results in the same RAS'-only refresh during PHASE 1 that occurs on the motherboard. During PHASE 0, things get a little more involved.

Enabling and disabling of transfer to and from the expansion RAM during PHASE 0 is controlled via RAS'. The RAM may be disabled, enabled for reading only, enabled for writing only, or enabled for both reading and writing under program control. When an address in the $D000-$FFFF range is on the address bus during PHASE 0 and expansion RAM is enabled for the existing state of R/W', RAS' will be gated to the eight expansion RAM chips. Also during PHASE 0, CAS' will follow RAS', delayed by one 7M period. This results in RAS' and CAS' falling at the expansion RAM at nearly the same point as motherboard RAS' and CAS' with the only difference being propagation delay.

The tri-state outputs of the expansion RAM are connected directly to the data bus, not to an 8-bit latch as on the motherboard. This makes timing less critical because RAM read data does not have to be valid when RAS' rises but must be valid when 6502 PHASE 2 falls.

Bank switching of the 4K auxiliary bank is accomplished via RA6. This is natural when you remember that RA6 is A12 during ROW access and A13 during COLUMN access. The four possible multiplexed states of RA6 therefore partition the 16K of expansion memory into its four most significant divisions. The $D000-$DFFF portion of expansion memory addressing is identified by A12 • A13' meaning the ROW-COLUMN states of RA6 are 1-0. If bank 2 is selected, this ROW-COLUMN combination is allowed to occur. If bank 1 is selected, the ROW-COLUMN combination is forced to 0-0, causing access to a different 4K area of expansion memory. Bank 2 is selected at power up and, under some conditions, by FP, INT, or RUN commands or a RESET when DOS is active. Bank 2 is therefore normally accessed at $D000-$DFFF, and bank 1 is the auxiliary bank.

The expansion RAM enabling and bank switching circuitry is centered around the four flip-flops of a 74LS175. These flip-flops save the current configuration of expansion memory which is set up by DEVICE SELECT' commands. The assigning of these command functions was obviously designed for DOS compatibility, so the RAM card would look like the firmware card to DOS when it was checking for availability of Applesoft and Integer BASIC. This means that a write access to $C080 must enable a RAM card in Slot 0 for reading only, and a write access to $C081 must disable the Slot 0 expansion card for reading. In their effort to maintain this compatibility, Apple came up with a rather involved programming method for configuring expansion RAM. The Slot 0 configuration characteristics go like this:

A3 controls the 4K bank selection. $C080-$C087 resets the BANK 1 flip-flop, enabling bank 2. $C088-$C08F sets the BANK 1 flip-flop, enabling bank 1.

A0 and A1 control the READ ENABLE flip-flop. Access to $C080, $C083, $C084, $C087, $C088, $C08B, $C08C, or $C08F sets the READ ENABLE flip-flop, enabling reading from expansion RAM. Access to $C081, $C082, $C085, $C086, $C089, $C08A, $C08D, or $C08E resets the READ ENABLE flip-flop, disabling reading from expansion RAM.

Writing to expansion RAM is enabled when the WRITE ENABLE' flip-flop is reset. The controlling MPU program must set the PRE-WRITE flip-flop before it can reset the WRITE ENABLE' flip-flop. The PRE-WRITE flip-flop is set by an odd read access in the $C08X range.

It is reset by an even access or a write access in the $C08X range. The WRITE ENABLE' flip-flop is reset by an odd read access in the $C08X range when the PRE-WRITE flip-flop is set. It is set by an even access in the $C08X range. Any other type of access causes the WRITE ENABLE' flip-flop to hold its current state.

At power up, the RAM card is disabled for reading and enabled for writing. The PRE-WRITE flip-flop is reset, and bank 2 is selected. The power-up circuit of R5 and C1 forces this configuration by resetting the LS175 flip-flops.

The Apple II system RESET at pin 31 of the RAM card has no effect on the RAM card configuration. The RESET handling program must configure the RAM card.

The write enabling flip-flops can be thought of as a write counter which counts odd read accesses in the $C08X range. The counter is set to zero by even or write access in the $C08X range. If the write counter reaches the count of 2, writing to expansion RAM becomes enabled. From that point, writing will stay enabled until an even access is made in the $C08X range. This means there is a feature of RAM card control not documented by Apple: write access to an odd address in the $C08X range controls the READ ENABLE flip-flop without affecting the state of the WRITE ENABLE' flip-flop.

The Slot 0 control characteristics are summarized in Table 5.4. There are two address commands possible for every function. The programming convention is to use addresses $C080-$C083 and $C088-$C08B.

Based on Table 5.4, here are some Slot 0 programming examples.

```
BIT $C080    WRTCOUNT = 0, WRITE
             DISABLE, READ ENABLE,
             BANK 2. SIMULATES
             FIRMWARE CARD.

BIT $C083    WRTCOUNT + 1, READ
             ENABLE, BANK 2.
BIT $C083    WRTCOUNT + 1, READ
             ENABLE, BANK 2.
             ENABLE 12K OF EXPAN-
             SION RAM FOR READING
             AND WRITING, BANK 2.

BIT $C081    WRTCOUNT + 1, READ
             DISABLE, BANK 2.
BIT $C081    WRTCOUNT + 1, READ
             DISABLE, BANK 2.
             ENABLE WRITE, DISABLE
             READ, BANK 2.

BIT $C082    WRTCOUNT = 0, WRITE
             DISABLE, READ DISABLE,
             BANK 2.  DISABLE
             EXPANSION RAM.

STA $C081    WRTCOUNT = 0, NO
             EFFECT ON WRITE
             ENABLING, DISABLE
             READ, BANK 2.

STA $C08B    WRTCOUNT = 0, NO
             EFFECT ON WRITE
             ENABLING, ENABLE
             READ, BANK 1.

BIT $C08B
BIT $C08B    ENABLE READ/WRITE
             BANK 1.
LDA $DXXX
BIT $C083    ENABLE READ/WRITE
             BANK 2.
STA $DXXX    TRANSFER BYTE FROM
             BANK 1 TO BANK 2.
```

As mentioned before, the unusual nature of the RAM card configuration commands are a result of making the RAM card compatible with DOS. This involves making it compatible with the short routine which starts at $A5B2 of DOS 3.3. This is the routine used by DOS to switch to Integer or Applesoft. It is entered with $20 in the accumulator when looking for Integer and $4C in the accumulator when looking for Applesoft. These are the values found at address $E000 of the two programs. With firmware card comments, the DOS language finding routine looks like this:

```
A5B2:  CMP $E000
A5B5:  BEQ $A5C5    ;FOUND IT.
A5B7:  STA $C080    ;ENABLE FIRM-
                    ;WARE CARD.
A5BA:  CMP $E000
A5BD:  BEQ $A5C5    ;FOUND IT.
A5BF:  STA $C081    ;DISABLE FIRM-
                    ;WARE CARD.
A5C2:  CMP $E000
A5C5:  RTS
```

If the desired language is already selected, no action is taken. Otherwise, the language is looked for with switching via STA $C080 first and STA $C081 second.

If a RAM card is installed in Slot 0, the comments look more like this:

```
A5B2:   CMP $E000
A5B5:   BEQ $A5C5     ;FOUND IT.
A5B7:   STA $C080     ;WRTCOUNT = 0,
                      ;DISABLE WRITE,
                      ;ENABLE READ,
                      ;BANK 2.
A5BA:   CMP $E000
A5BD:   BEQ $A5C5     ;FOUND IT.
A5BF:   STA $C081     ;WRTCOUNT = 0,
                      ;NO EFFECT ON
                      ;WRITE ENABLING,
                      ;DISABLE READ,
                      ;BANK 2.
A5C2:   CMP $E000
A5C5:   RTS
```

If the desired language is already selected, no action is taken. Otherwise the language is looked for via storing commands which reset the write enable count, disable the RAM card for writing, select bank 2, and enable then disable the RAM card for reading. This means that with the RAM card installed and DOS connected, initialization of either Applesoft or Integer BASIC via FP, INT, RUN, or RESET will select bank 2 and disable writing to the RAM card. Reentering a language, however, does not effect the RAM card configuration at all. For example, "RUNing" an Applesoft program when Applesoft is already active causes no change in the status of RAM card read enable, write enable, or bank selection flip-flops.

The action of RESET at the RAM card is of particular interest. RESET performs no hardware function at the RAM card. The power-up configuration of the RAM card is controlled by circuitry on the card which is not associated with the Apple system RESET that occurs at power up. The only effect RESET has on the RAM card is through the RESET handling program. If DOS is not connected or some other controlling program is not cognizant of the 16K RAM card, pressing RESET will do nothing to the RAM card. Apple supports only one RAM card installation, namely: RAM card in Slot 0, Autostart Monitor active, and DOS connected. The RESET is soft, and any variations from the normal installation will require appropriate software support.

Table 5.4 16K Ram Card Address Bus Commands.

BANK 2	BANK 1	ACTION	
C080 C084	**C088** C08C	WRTCOUNT = 0*, WRITE DISABLE	READ ENABLE
RC081 RC085	**RC089** RC08D	WRTCOUNT = WRTCOUNT + 1*	READ DISABLE
WC081 WC085	**WC089** WC08D	WRTCOUNT = 0*	READ DISABLE
C082 C086	**C08A** C08E	WRTCOUNT = 0*, WRITE DISABLE	READ DISABLE
RC083 RC087	**RC08B** RC08F	WRTCOUNT = WRTCOUNT + 1*	READ ENABLE
WC083 WC087	**WC08B** WC08F	WRTCOUNT = 0*	READ ENABLE

*Writing to expansion RAM is enabled when WRTCOUNT reaches 2.

The 16K RAM card also contains a socket for a 2316 ROM or 2716 EPROM. To use 2716 EPROM, one jumper pad must be cut and a second must be soldered. These configuration pads are labeled 2716 on the RAM card. The ROM is an F8 ROM, meaning that it responds to addressing in the $F800-$FFFF range. Operation is such that when the expansion RAM is disabled, the F8 ROM on the RAM card is enabled for response to the $F800-$FFFF range. In other words, when the RAM card is installed, the F8 ROM on the motherboard is never accessed and may as well not be installed. There must, however, be an F8 ROM installed on the RAM card.

It is the author's guess that the reason Apple included the ROM socket was as a vehicle for distributing the Autostart ROM to those persons who were still using the old Monitor ROM. Additionally, they gave owners of the "Language System" an easy means of installing their own F8 EPROM. User beware! Some commercial software won't run correctly if anything other than the Autostart Monitor or old Monitor programs are responding to $F800-$FFFF. This is because some copy protection schemes call for performing a checksum on this addressing range to ensure the user has not installed copy busting firmware. If the checksum is incorrect, the program won't run.

The INHIBIT' output is the means by which the RAM card steals $D000-$FFFF addressing. Bringing the INHIBIT' line low disables motherboard ROM response to this address range. The RAM card brings the INHIBIT' line low:

1. If an address in the $F800-$FFFF range is on the address bus.
2. If an address in the $D000-$FFFF range is on the address bus, R/W' is high, and the expansion RAM is enabled for reading. This also allows RAS' and CAS' to fall during PHASE 0 at the expansion RAM. It also disables the F8 ROM on the RAM card.
3. If an address in the $D000-$FFFF range is on the address bus, R/W' is low, and the expansion RAM is enabled for writing. This also allows RAS' and CAS' to fall at the expansion RAM during PHASE 0. It also disables the F8 ROM on the RAM card.

The 16K RAM card is normally mounted only in Slot 0, in spite of the fact that the card will operate in any slot. Multiple RAM cards or combinations of RAM cards and firmware cards are rare because little or no software exists to support such configurations. Also, unlike the firmware card, the RAM card does not utilize the DMA priority chain to provide a fail-safe method of preventing two or more cards from trying to steal $D000-$FFFF at the same time. As a result, multiple RAM/firmware card configurations would be susceptible to accidental simultaneous enabling. The RESET key might be of little help in the resulting chaotic situation because RESET does not automatically disable RAM cards. RESET does disable firmware cards if the switch in the back is down. The fact remains that use of the RAM card anywhere but in Slot 0 would require very careful software support, probably generated by the owner.*

One last point of detail in the RAM card schematic is that RA0 through RA6 are all connected to an unused LSTTL input. It is the author's assumption that this is to prevent reflections which might otherwise be caused by transmitting RAM address information via a DIP jumper to nine RAM chips on a peripheral card.

*Multiple RAM card configurations are the subject of an Application Note at the end of this chapter.

HARDWARE APPLICATION

UPGRADING APPLES TO 48K RAM

There has been a dramatic decrease in the price of 16K dynamic RAM chips since the Apple was first introduced. It is the author's perception that the prime cause of this price reduction has been competition from Japanese manufacturers. Also, the 16K RAM chip is becoming obsolete as a mass memory device. Early Apple owners paid hundreds of dollars to achieve the Apple's 48K capability. In 1980, the author saved $300 from the price of his Apple by buying the 16K version, taking a train to Tokyo, and paying $60 for the remaining 32K of RAM chips. In 1981, the advertised price in U.S. magazines for sets of eight 16K dynamic RAM chips was $25. The December, 1982 issue of BYTE magazine carried an ad for 200 nanosecond, 16K chips at $10 per set of eight. The point is this: you should have 48K of RAM in your Apple II, and you shouldn't pay very much money to put it there.

The Apple supports 48K of motherboard RAM fully. Plug the chips into the sockets and away it goes. When changing from 4K to 16K chips in older Apples, it will be necessary to buy or build 16K configuration plugs. You can modify old 4K plugs to 16K plugs or build 16K plugs out of IC sockets. The schematic of the 16K jumper plug can be seen in Figure 5.10. In most instances, coming up with 16K jumper plugs won't be necessary because a 16K Apple will come jumpered for 48K. Newer Apples have no RAM jumpers, but are hard-wired for 16K chips.

The only remaining questions are what chips to buy and where to buy them. What to buy is 4116 type 16K RAM chips that are fast enough to operate in the Apple, which is to say, not exactly greased lightning. RAM chips are usually referred to as 250 nanosecond RAM, or 300 nanosecond RAM or something similar. This time is the read response time of the chip. There are two important read response times in dynamic RAM—the maximum time after RAS' falls before data becomes valid (TRAC), and the maximum time after CAS' before the data becomes valid (TCAC). The TRAC period is usually the way the chips are referred to. Read data from 250 nanosecond RAM becomes valid a maximum of 250 nanoseconds after RAS' falls, except not in the Apple. You see, when CAS' falls pretty soon after RAS' falls, the access is **ROW-limited**. CAS' falls 140 nanoseconds after RAS' in the Apple, which is not soon enough, so Apple RAM access is **COLUMN-limited**. In other words, the TCAC specification is what determines the acceptability of a RAM chip for use in the Apple.

In the Apple, RAM data output is latched by the rising edge of RAS', so TCAC on Apple RAM chips should be less than 178 nanoseconds to insure data is valid before RAS' rises. This criteria is met on 250 nanosecond or faster RAM (TCAC = 165 nanoseconds), but is not met by 300 nanosecond RAM. Since TCAC is the critical specification, you can operate the Apple with 250 nanosecond RAM. The author's computer was supplied with NEC 416-2 (200 nsec) chips. The computer was upgraded to 48K with Fujitsu 8116N (250 nsec) chips.

Table 5.5 is a list of manufacturer's part numbers of 4116 compatible 16K dynamic RAM chips. If you do not have a good discount supplier, the best place to get RAM is through the mail. It is recommended that you peruse microcomputer magazines to find a hardware supplier you can live with. Strictly as a convenience to readers of this book, a list is included here of some suppliers advertising 4116 type RAM chips for under $15 per eight in the December, 1982 issue of BYTE Magazine. No endorsement of these suppliers is intended. The reader may wish to contact one of them to acquire ordering information.

DoKay Computer Products Inc.
3250 Keller St. #9
Santa Clara, Ca. 95050
1-800-538-8800
1-800-848-8008 (Ca.)
1-408-988-0697

Jameco Electronics
1355 Shoreway Road
Belmont, Ca. 94002
1-415-592-8097

JDR Microdevices Inc.
1224 S. Bascom Avenue
San Jose, Ca. 95128
1-800-538-5000
1-800-662-6279 (Ca.)
1-408-995-5430

Table 5.5 16K Dynamic RAM Chips.

MANUFACTURER	120 nsec	150 nsec	200 nsec	250 nsec
AMD		9016F	9016E	9016D
Fairchild		F4116-2	F4116-3	f4116-4
Fujitsu		MB8116H	MB8116E	MB8116N
Hitachi	HM4716A-1	HM4716A-2	HM4716-3	HM4716A-4
Intel		P2117-2	P2117-3	P2117-4
Intersil		IM4116-2	IM4116-3	IM4116-4
ITT		ITT4116-2	ITT4116-3	ITT4116-4
Mitsubishi		M5K4116-2	M5K4116-3	M5K4116-4
Mostek		MK4116-2	MK4116-3	MK4116-4
Motorola		MCM4116B-15	MCM4116B-20	MCM4116B-25
National	MM5290-1	MM5290-2	MM5290-3	MM5290-4
NEC Micro	uPD416-5	uPD416-3	uPD416-2	uPD416-1
Oki		MSM3716-2	MSM3716-3	MSM3716-4
Panasonic			MN4116	
Siemens			HYB4116-3	HYB4116-4
TI		TMS4116-15	TMS4116-20	TMS4116-25
Toshiba		TMM416-2	TMM416-3	TMM416-4
Zilog		Z6166-2	Z6166-3	Z6166-4

Quest Electronics
P.O. Box 4430X
Santa Clara, Ca. 95054
1-800-538-8196 (orders only)
1-408-988-1640

Solid State Sales
P.O. Box 74B
Sommerville, Ma. 02143
1-800-343-5230 (orders only)
1-617-547-7053

HARDWARE APPLICATION

BANK SWITCHING THE MOTHERBOARD RAM

The Apple design includes a very versatile characteristic in that the motherboard ROM can be inhibited from the peripheral slots. This feature enables the design of very important peripherals such as ROM and RAM expansion cards which work by stealing ROM addressing ($D000-$FFFF). With the coming of the 64K RAM chip, we now have 64K, 128K, and bigger Apple RAM expansion cards, divided up into 12K banks which are switched in and out and addressed at $D000-$FFFF with motherboard ROM disabled.

There is a trick, though, which the RAM card designers haven't picked up. The trick is that it is quite possible and easy to disable motherboard RAM in the Apple. By disable, it is meant that you can isolate the motherboard RAM from the data bus and access other devices at addresses $0000-$BFFF. This opens up the possibility of peripheral card designs with RAM or ROM switched in banks of up to 64K.

Disabling of motherboard RAM is easy, because the functions of gating CAS' to the RAM chips and gating RAM output data to the data bus are both controlled from one chip, F2. This chip is the LS139 decoder which decodes address bits to generate RAM SELECT' and CAS' for rows C, D, and E (see Figure 5.10). Consider what would happen if all outputs of F2 were forced high during PHASE 0, regardless of the address bus. First, CAS' would not be generated for any RAM row, so no data transfer would take place to or from motherboard RAM. Second, RAM SELECT' would stay high unless $C00X was being addressed, so latched data would not be gated to the data bus, even if the MPU was calling out a RAM address. Thus the data bus is floating and available for control from a peripheral card. Third, the scanner access is not interrupted, because the F2 outputs are forced only during PHASE 0. So motherboard RAM is still refreshed as it is scanned for video output.

The way to implement this is to pull the LS139 from the F2 socket and connect the socket to the peripheral card via a short 16-line DIP jumper (short because F2 is close to the peripheral slots). The LS139 is plugged into a socket on the peripheral card where it performs the same functions it did on the motherboard except that the outputs are enabled by bank switching logic.

Figure 5.16 shows the basic idea of how to bank switch motherboard RAM. Only one half of the LS139 is used and CAS' is gated by three separate low level AND gates. This is very important, because the scheme causes no further delay in CAS' propagation than is normally tolerated in motherboard RAM operation. This will ensure correct operation, even with 250 nanosecond RAM. The AND gate propagation delay will actually be about 10 nanoseconds less than in normal operation, making the 6502 motherboard write data setup slightly more critical. However, you would probably have to put a 6502 in a freezer to make it take 200 nanoseconds to set up write data after its PHASE 2 clock rises. CAS' could be routed through an extra logic device to completely equalize CAS' propagation time to that of normal operation.

The 64K bank switched RAM could be controlled by any number of schemes. The most naturally implemented scheme would be a 192K RAM card, giving the expanded Apple a 256K byte capability in four banks. The screen display is always stored in motherboard RAM and special care would be required for bank switching PAGE 0 and PAGE 1.

Expansion RAM is not the only possible application of the capability to inhibit motherboard RAM. For example, a fairly inexpensive design could be built with the Apple DOS in EPROM, operating at its normal address. For about $50 in parts you could leave behind forever the process of booting the DOS and the necessity of wasting disk space on the Apple DOS. That would be a pretty nice improvement to the disk controller.

The design suggestions in this Application Note are put forth in hopes of encouraging hardware manufacturers to develop this previously overlooked Apple capability.

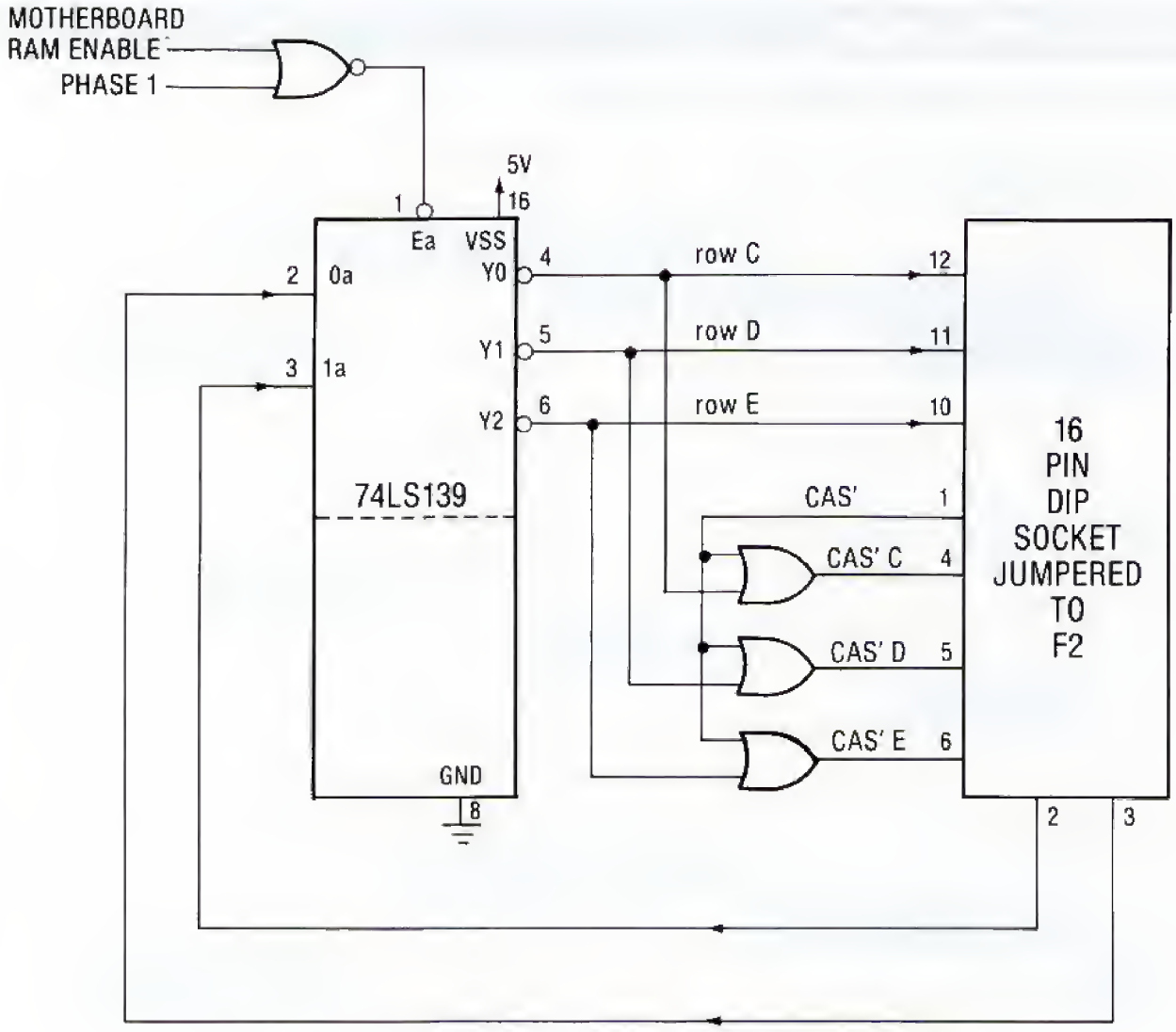


Figure 5.16 Schematic: Bank Switching Motherboard RAM.

SOFTWARE APPLICATION

READING VIDEO DATA FROM A PROGRAM

In the October, 1982 issue of *SOFTALK* magazine, Bob Bishop reported an exciting discovery. Any time a 6502 program reads an address from which there is no data response, the video data from the previous scanner access is read by the MPU. Mr. Bishop made a big mistake publishing this information. He could have made a living making bets with Apple experts that you can sync an unmodified Apple to the video scan under program control. He'd have made about a thousand dollars per sucker. Who would have thought that the data on the floating data bus would remain valid for over half a microsecond. The article "Have an Apple Split" contains programming examples and is highly recommended reading. It is partly because of "Have an Apple Split" that this book has endeavored to present extensive memory scanning maps for programmer reference.

One reason for syncing a program to the video scan is to create displays that are a mixture of the normal Apple screen modes. For example, if you switch to HIRES before line 5 of every vertical scan and switch to LORES before line 10 of every vertical scan, you will have a stable combination of HIRES and LORES graphics displayed on the screen.

The method of reading video sync from a program is to set up flags in scanned memory at the point of the television scan where the program needs to take an action, such as switching from LORES to HIRES. Then the program polls a nonresponding address such as the cassette output port until it detects the flag. The choice of flags and their location in memory will be dictated by the application.

Programming a combination display requires the normal programmer's imagination to conceive of the display. Additionally, a thorough grasp of memory scanning details is a necessity. The purpose of this application note is to provide some discussion of techniques that might be used.

For starters, here is a list of addresses from which video data can be read:

$C01X	KEYBOARD STROBE FF Reset
$C02X	Cassette Output
$C03X	Speaker Output
$C04X	C040 STROBE'
$C05X	Screen Switches and Annunciator Outputs
$C06X	Serial Inputs (D0-D6 only)
$C07X	Timer Trigger
$C080-$C7FF	I/O Control (use if slot is empty)
$CFFF	Expansion ROM Disable

Of all these choices, the screen switches stand out as having no chance of interfering with the operation of some device. The screen switches will normally be used, because the task of polling and switching can then be combined. Also, the polling loop is self documenting to investigators of the program. In the rare instance that one of the screen switches will not do for video polling, the annunciators, the keyboard strobe reset, the $C040 strobe, and the cassette output addresses are likely choices.

Figure 5.17 is a diagram designed to aid the programmer in selecting locations for syncing flags. It should be used in conjunction with the memory maps from earlier sections of this chapter. From this diagram, one can quickly see the prospects for successful syncing at a given point in a given mode. The most obvious feature discerned from Figure 5.17 is that HBL scanned memory in HIRES overlaps HBL' scanned memory, but the HBL and HBL' scanned memory areas of TEXT/LORES are distinct. This means that different problems will arise when choosing flags for these two different memory scanning modes.

A problem with HIRES is that most memory gets scanned more than once every vertical scan. Only the first 16 bytes of the SECOND 40 and the first 16 bytes of the THIRD 40 are scanned just once per vertical frame. This makes it more difficult to uniquely flag a scan position. In TEXT/LORES, memory is more uniquely scanned with only a partial overlap between the area scanned during VBL and during the top of the screen. However, one must deal with the fact that critical non-displayed memory areas are scanned. HBL scanned memory is in an area normally taken up by Applesoft programs or Integer BASIC variables, and the undisplayed eight bytes at the end of each 128 byte segment are used by the DOS for critical disk information (like active slot number).

The following is a typical video polling loop:

```
POLLIT  CMP $C050   ;FLAG VALUE
                    ;IN ACCUMU-
                    ;LATOR.
        BNE POLLIT
```

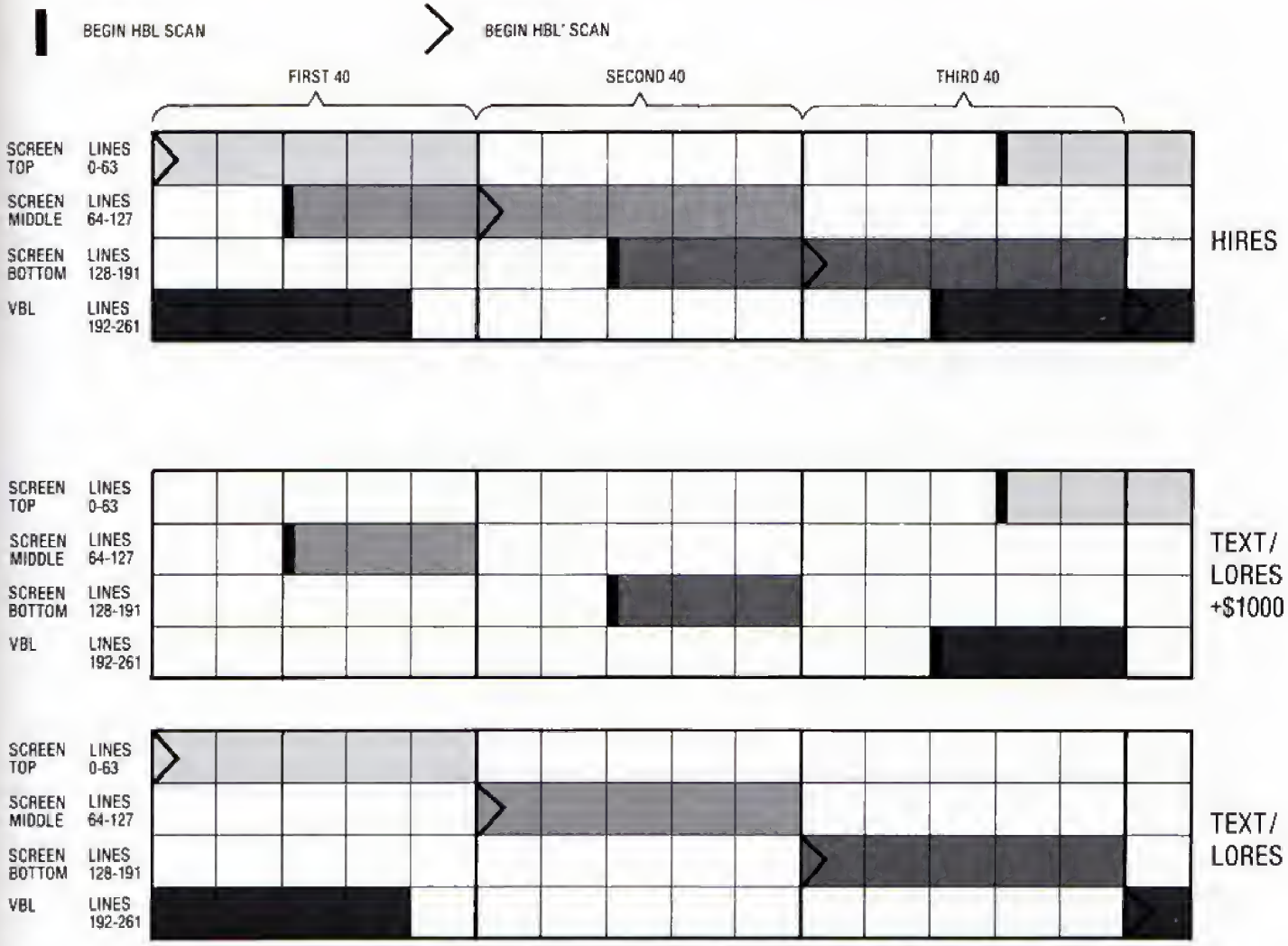


Figure 5.17 Screen Memory Scanning.

The loop takes seven clockpulses to execute and thus establishes one criteria for a screen flag. It must occupy a minimum of seven adjacent scanned bytes of memory or six bytes if one of the bytes is the first byte of a HBL scan. The first byte of a HBL scan is scanned twice so six bytes here is the same as seven elsewhere.

Here is one way to flag PAGE 1 of TEXT memory at the beginning of line 12 (the thirteenth line from the top if the first line is line 0). Store the value $5B at locations $1610-$1627. Figure 5.6 shows this is HBL scanned memory before line 12, and Figure 5.17 shows that this memory will only be scanned once per vertical scan. $5B is the ASCII value of a flashing left bracket which is very unlikely to be found in text memory. It may be found in the UNUSED 8 or in other HBL scanned memory, which creates a problem. One way to solve it is to blank the other HBL scanned memory. Then poll for the flag as follows:

```
POLLIT    CMP $C051   ;ACCUMULATOR
                      ;IS $5B
          BNE POLLIT
          NOP
          CMP $C051
          BNE POLLIT
```

This loop will not be exited without finding two $5Bs separated by eight bytes. This excludes accidentally syncing on anything in the UNUSED 8 and solves the problem without messing up the memory locations used in disk I/O. Any Applesoft program beginning at $800 and extending beyond $140F is clobbered.

It is not necessary to uniquely flag a scan position. It is only necessary that there is no interference in detecting a scan flag between the present scan position and the flagged scan position. For example, assume we switched from HIRES to LORES at the middle of the display screen and we wish to detect the beginning of VBL. A flag at $1460 through $1477 will serve this purpose nicely. Even though parts of this memory are also scanned at the top of the screen, the next time they are scanned is during HBL, before VBL. There is no interference between the present location and the detection point.

Mr. Bishop showed an interesting flagging technique in his *SOFTALK* article. He flags the middle of the TEXT/LORES display area with a string of $E0s. He uses this to switch from TEXT to LORES. These $E0s are printed as spaces on the text screen and as a row of black blocks and a row of aqua blocks in LORES. By detecting this row of $E0s from TEXT mode then switching to LORES mode for the lower part of the screen, the program effortlessly transits from an empty text line to a LORES display with an upper aqua border. The program is short and simple and illustrates that programming mixed screen displays is a bit of an art form.

The further possible flagging techniques have not all been dreamed up yet. Here are some related ideas and facts:

1. Switching modes during HBL or VBL eliminates the unsightly display of switch points. This can be accomplished by flagging VBL or HBL or by flagging a display area and waiting for HBL or VBL with timed execution loops.
2. The UNUSED 8 is the only undisplayed area of HIRES available for flagging. It is scanned during HBL before the top third of the display and just after HBL during VBL. The undisplayed eight is of minimal use in flagging PAGE 1 of TEXT/LORES because of interference with the DOS.
3. Bit 7 of HIRES may be used as a flag and checked with the BIT instruction. This is one way of flagging the displayed HIRES area if bit 7 isn't critical for color or positioning.
4. Bit 7 is a good TEXT flag if there is no inverse video on the screen. Other likely TEXT flags are flashing ASCII, control ASCII, lowercase ASCII, and ASCII of characters not supported by the Apple keyboard.
5. When considering LORES flags in displayed areas, bits 0-3 control the upper block, and bits 4-7 control the lower block.
6. The video polling method works very well in conjunction with timed execution loops. There are 65 cycles in a horizontal scan—25 cycles of HBL and 40 cycles of HBL'. There are 262 horizontal scans in a vertical scan—64 in each third of the display screen and 70 during VBL. When a group of flagged bytes is located, it is then possible to find a precise byte in the group by slewing backwards in 17029 cycle loops until the first flagged byte is found. 17031 cycle loops can be used to slew forwards.
7. The first byte scanned during HBL is scanned twice in a row. In TEXT/LORES, every memory line is scanned eight times in a row, except the last line of VBL is scanned 14 times. In HIRES, every memory line is scanned once, but lines 250-255 are rescanned after line 255 (250 is same as 256, 251 is same as 257, etc.).
8. Switching rapidly between GRAPHICS and TEXT mode will cause many televisions to lose color sync. This is a factor of alignment and response of the 3.58 MHz oscillator inside the television. Because of this unpredictability from TV to TV, it is not possible to say what percentage of the time a program can leave the Apple in TEXT mode and still hope to maintain color sync. Commercial programmers could be well advised to keep at least a 50% GRAPHICS mode to TEXT mode ratio in their products if color stability is important.
9. A very heavily loaded data bus may not reliably store video data long enough to be read by the MPU. Apparently, some commercially available peripheral cards cause this condition when plugged in. Beware!

Figures 5.18 and 5.19 are example programs that will create a mixed display in which all the text on the screen except the bottom line is underlined. NORMAL, INVERSE, and FLASHING text are all underlined. The program works by setting up a LORES map of PAGE 1 text in PAGE 2. Then the screen mode is switched to LORES PAGE 2 for the top of every text line except line 0, then back to TEXT PAGE 1 for the next seven lines.

Figure 5.18 is an Integer BASIC program which sets up the PAGE 2 map and flags HBL scanned memory for both PAGE 1 and PAGE 2. The process is pretty slow in BASIC, but this program is included for illustration and BASIC is easy to follow. Just give the program about thirty seconds to work. Integer BASIC is used because Integer is very easy to work around when modifying low memory just above $800. The actual screen splitting is done by the assembly language program in Figure 5.19.

```
 1 REM
 2 REM
 3 REM  CALL UNDERLINE
 4 REM
 5 REM
 6 REM  FIRST MOVE INTEGER LOW POINTERS UP TO $1C40.
 7 REM
 8 REM
10 POKE 74,64: POKE 75,28: POKE 204,64: POKE 205,28
20 PRINT "BLOAD UNDERLINE.OBJ0"
22 POKE -16304,0: POKE -16299,0: POKE -16298,0: REM LOOK AT LORES PAGE 2.
25 REM
26 REM
27 REM  NOW FILL PAGE 2 WITH THE UNDERLINES FOR THE CURRENT TEXT DISPLAY.
28 REM
29 REM
30 FOR P2=2048 TO 2176: POKE P2,0: NEXT P2: REM BLANK TOP LINE OF PAGE 2
40 FOR P1=1024 TO 1919:P2=P1+1152: GOSUB 160
45 NEXT P1
50 FOR P1=1920 TO 1999:P2=P1+168: GOSUB 160
55 NEXT P1
60 REM
61 REM
62 REM  NOW FLAG HBL SCANNED MEMORY WITH "$5B"S.
63 REM
65 REM
70 FOR A=0 TO 1920 STEP 128: FOR B=0 TO 23
75 FLAG=0: IF B<6 THEN FLAG=91: REM  91 IS FLASHING LEFT BRACKEET
80 POKE 5136+A+B,FLAG: REM HBL BEFORE MIDDLE SCREEN = FLAG
90 POKE 5176+A+B,FLAG: REM HBL BEFORE BOTTOM SCREEN = FLAG
100 NEXT B
110 FOR B=0 TO 31
115 FLAG=0: IF B<14 THEN FLAG=91: REM   91 IS FLASHING LEFT BRACKET
120 POKE 5216+A+B,FLAG: REM HBL BEFORE VBL AND TOP SCREEN = FLAG
130 NEXT B: NEXT A
140 CALL 7168: REM  GO SPLIT THE SCREEN
150 END
155 REM
156 REM COLOR = 0 OR 1 OR 15
157 REM
160 T= PEEK (P1):COLR=1: REM  UNDERLINE COLOR = 1
165 IF T=32 THEN COLR=15: REM  INVERSE SPACE?
170 IF T=96 OR T=160 OR T=224 THEN COLR=0: REM  SPACE?
180 POKE P2,COLR: RETURN
```

Figure 5.18 Integer BASIC Listing: Underline Program.

The difficult part about a program of this nature is designing a flagging method. UNDERLINE uses strings of $5Bs as flags stored in the HBL areas. $5B is the code for a flashing left bracket, which can safely be said to rarely be printed on the TEXT screen. A string of 14 $5Bs is stored beginning at HBL before VBL. Note from Figure 5.17 that this also stores six $5Bs in HBL before screen top. Additionally, six $5Bs are stored in HBL before screen middle and HBL before screen bottom. All other HBL scanned bytes are cleared. Since the first byte of HBL scanned memory is driven out twice, this flagging method results in seven $5Bs being driven out every HBL before display and fifteen $5Bs being driven out at HBL before VBL.

Here's the scheme for switching. First, wait for VBL. HBL before VBL is the only area which will respond to two consecutive polls with $5B. Once the

```
SOURCE FILE: UNDERLINE
0000:              1 ********************************************************
0000:              2 *                                                      *
0000:              3 *                                                      *
0000:              4 *                  UNDERLINE TEXT                      *
0000:              5 *                                                      *
0000:              6 *                  BY JIM SATHER                       *
0000:              7 *                                                      *
0000:              8 *                      1/4/83                          *
0000:              9 *                                                      *
0000:             10 *                                                      *
0000:             11 ********************************************************
0000:             12 *
0000:             13 *
C050:             14 GRAPHIX EQU  $C050
C051:             15 TEXT    EQU  $C051
C052:             16 NOMIX   EQU  $C052
C054:             17 PAGE1   EQU  $C054
C055:             18 PAGE2   EQU  $C055
FCA8:             19 WAIT    EQU  $FCA8
0000:             20 *
0000:             21 *
0000:             22 *
----- NEXT OBJECT FILE NAME IS UNDERLINE.OBJ0
1C00:             23         ORG  $1C00
1C00:8D 52 C0     24 DOSCRN  STA  NOMIX
1C03:8D 54 C0     25         STA  PAGE1
1C06:A9 5B        26         LDA  #$5B
1C08:CD 51 C0     27 SCRNLP  CMP  TEXT      LOOK FOR STRING OF 15 "$5B"S
1C0B:D0 FB        28         BNE  SCRNLP
1C0D:EA           29         NOP
1C0E:CD 51 C0     30         CMP  TEXT
1C11:D0 F5        31         BNE  SCRNLP
1C13:A9 28        32         LDA  #$28      GOT VBL
1C15:20 A8 FC     33         JSR  WAIT      WAIT 4553 CYCLES
1C18:A9 5B        34         LDA  #$5B
1C1A:A0 17        35         LDY  #23       DO 23 LINES
1C1C:A2 08        36         LDX  #8
1C1E:D0 02        37         BNE  TEXTLP
1C20:             38 *
1C20:             39 *
1C20:A2 07        40 DOLINE  LDX  #7        7 TEXT LINES FOR 1 LINE OF GRAPHICS
1C22:CD 51 C0     41 TEXTLP  CMP  TEXT
1C25:D0 FB        42         BNE  TEXTLP
1C27:CA           43         DEX            GOT HBL
1C28:D0 F8        44         BNE  TEXTLP
1C2A:8D 55 C0     45         STA  PAGE2     SWITCH TO LORES
1C2D:CD 50 C0     46 LORESLP CMP  GRAPHIX
1C30:D0 FB        47         BNE  LORESLP
1C32:8D 54 C0     48         STA  PAGE1     GOT HBL
1C35:8D 51 C0     49         STA  TEXT
1C38:88           50         DEY
1C39:D0 E5        51         BNE  DOLINE
1C3B:F0 CB        52         BEQ  SCRNLP

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 5.19 Assembler Listing: Underline Program.

first line of VBL is located, wait 4553 cycles. This is a ball park figure which waits until after HBL of the first displayed line. Now that the scan is past VBL, Figure 5.17 shows that there is no further possibility of reading $5B in polling loops before the next VBL, except that $5B will be read once within the first seven bytes of every HBL. This creates a flagged situation in which HBL can be detected before any displayed line.

The given task is to underline the current PAGE 1 TEXT display. The ground work for doing this is laid in the BASIC program. First understand that the top line of every text character pattern is blank. This creates the space between the text lines. To underline the character at position 1 of line 0, you map a LORES block into the top of position 1 of line 1 in PAGE 2. Then for the first scan of text, line 1 you switch to GRAPHICS PAGE 2. During HBL of the next scan, you switch back to TEXT PAGE 1. This creates an underline appearance for line 0. The BASIC program simply checks the PAGE 1 display memory for space or no space, then maps corresponding LORES blocks into the adjacent line of PAGE 2 memory. The bottom line of text is not underlined because there is no adjacent lower line to switch to LORES.

With screen memory fully mapped, the task of the screen splitting program is reduced to finding the top of the screen precisely, then switching to LORES PAGE 2 for one line every eighth line. It's easy once the flags are properly set.

Since the Apple is in TEXT mode most of the time during UNDERLINE, probably any television will lose color sync and the underline will be white rather than colored. Readers may wish to experiment with colors by changing lines 160 through 180 of the BASIC program. Just don't POKE 91 into LORES displayed memory. 91 ($5B) is our flag, and it is reserved only for undisplayed memory.

HARDWARE APPLICATION

MULTIPLE RAM CARD CONFIGURATIONS

Installing a 16K RAM card in Slot 0 makes a nice improvement to the Apple. The capability of loading programs to high memory is inarguably desirable. But a natural question arises as to whether it is possible to install more than one RAM card in an Apple and use the second card for auxiliary data or program storage. The answer to this question is a qualified "yes." It is possible to use more than one 16K RAM card in the Apple, but certain obstacles must be overcome. This Application Note discusses the nature of the obstacles and offers some ways to overcome them.

The first obstacle is the nature of the ROM enabling circuitry on the RAM card. If the expansion RAM is not enabled on a RAM card, the ROM on the card will respond to $F800-$FFFF addressing—period. There is no way to inhibit this response without modifying the RAM card. The result is that when a RAM card is installed, neither the motherboard nor any peripheral card can respond to F8 addressing. This "I own F8" philosophy of the 16K RAM card represents a degradation of the versatility of the Apple which, fortunately, is easily overcome. Just perform the following simple modification to each RAM card you want to install with other F8 stealing cards.*

1. Remove the F8 ROM from the RAM card.
2. Remove the 74LS20 IC from the A5 socket of the RAM card.
3. Bend pin 6 of the 74LS20 IC so it will not go into the socket and reinstall the 74LS20 in the A5 socket.
4. Place the RAM card, component side down, on your work table. Solder a bare wire jumper between pins 4 and 5 of the A4 socket (A4 holds a 74LS09).
5. Install your primary F8 ROM in the F8 socket of the motherboard. Leave the ROM socket on the 16K RAM card empty.

The original versatility of your Apple is restored by this modification. The RAM card responds to F8 addressing only when enabled for reading or writing to RAM. Any RAM card, firmware card, or other peripheral card can steal access to F8 addressing by pulling the INHIBIT' line low. If INHIBIT' is high, the motherboard F8 ROM will respond to F8 addressing.

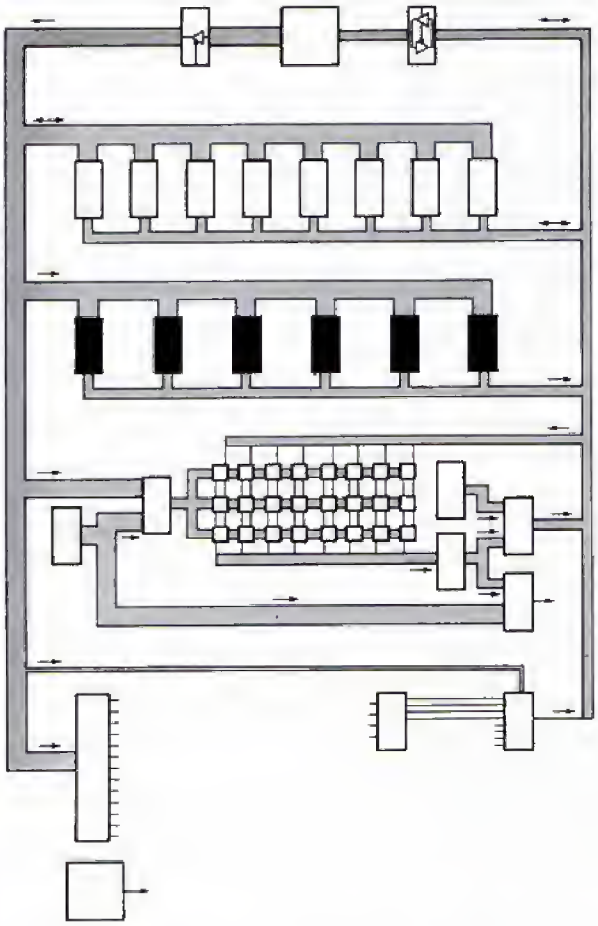
*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

A second obstacle to multiple RAM card configurations is loading on the RA0-RA6 multiplexed RAM address lines. These lines drive 24 RAM chips on the motherboard plus 1000 ohm pull-up and pull-down resistors. Additionally, RA0-RA5 drive eight RAM chips and one LSTTL load on every RAM card installed, and RA06 drives two LSTTL loads on every RAM card installed. Presumably, this loading places a practical limitation on the number of RAM cards which can be installed in an Apple. I have made no attempt to define this limit through experimentation, so I can only suggest that multiple RAM card users should be on the lookout for signs of unreliable transfer of RAM data. Symptoms of such problems would quite possibly vary with operating temperature.

If RA0-RA6 prove to be overloaded in your Apple, one trade-off you can consider is to replace the 74LS153s at C1, E11, E12, and E13 on the motherboard with 74S153s (see Figure 5.10). STTL chips draw more current than LSTTL chips at their inputs, but they can drive more circuits connected to their outputs. The trade-off is that changing to 74S153s will greatly increase the amount of RAM chips that RA0-RA6 can drive, but it will cause some increase of the load on the address bus. The desirability of this trade-off must be evaluated for your total peripheral card configuration.

The final obstacle to multiple RAM card configurations is that all software support must be generated by the user. This will, of course, be the most time consuming obstacle to overcome. Important points to consider in your software are handling of your configuration after a RESET, patching DOS and Pascal to support your configuration, and fail-safe software which avoids enabling two different RAM cards for reading. One programming trick is to enable two RAM cards for writing simultaneously when you want to fill their memories with identical data. Both RAM cards will accept the data from a single storing instruction to the $D000-$FFFF range.

When installing more than one RAM card, a separate DIP jumper must be connected from a motherboard RAM socket to each RAM card. Just select any convenient RAM socket for each RAM card and move the RAM chip from that socket to the empty socket on the RAM card. The motherboard socket which you choose makes no difference because the signals which the RAM card uses (RAS' and RA0-RA6) are identical on every socket.



chapter 6

ROM in the Apple II

Non-erasable, random access, read only memories have taken many forms in the history of digital computers. From vacuum tubes to diodes to small scale integration to large scale integration, there has always been a need for the general purpose computer to have a resident program ready to tell it what to do at turn on. With the coming of large read only memories on single chips, the scope of programs contained in ROM in general purpose computers has expanded greatly. Placing a BASIC interpreter in ROM was a significant event in the development of personal computers.

The Apple II design supports 12,288 bytes of motherboard ROM, addressed from $D000 through $FFFF. Additionally, several provisions exist for controlling the Apple via programs in ROM on peripheral cards. These include addressing peripheral ROM using a slot's assigned address area (its $C0nX DEVICE SELECT' range and its $CnXX I/O SELECT' range), addressing peripheral ROM using the expansion ROM addresses ($C800-$CFFF), and inhibiting motherboard ROM and stealing addresses $D000-$FFFF. All told, the Apple computer has a very versatile capability for operating under control of programs stored in ROM.

The sophistication of the ROM chips makes their connections in the Apple simple and easy to understand. Nevertheless, the topic of ROM is involved enough to merit its own chapter. Since the hardware connections are simple, let that be the starting point.

ROM HARDWARE

The Apple II firmware is programmed into six **2316B type ROMs**. The 2316B is a 24 pin, 2048 byte, NMOS ROM with three programmable chip selects and a 450 nanosecond access time. NMOS stands for Negative channel, Metal Oxide Silicon construction. These technical terms will help you if you look for the equivalent chip in a manufacturer's data book. Look under NMOS, 2048x8 ROMS.

The 2316B goes by several different manufacturer's part numbers. This book refers to it as the 2316B because that seems to be the most common identifying number. The *Apple II Reference Manual* refers to the ROM as 9316B, which is the part number used by General Instruments for 2316B compatible ROM. Perhaps General Instruments was the ROM supplier when the Apple II was first introduced. All the Apples that the author has inspected have Synertek ROMs and the Synertek part number is 2316B. In

any case, the 2316B, 9316B, and all other 2316B compatible ROMs have the same characteristics.

These ROMs are programmed by the manufacturer to the specifications of Apple Computer Inc. Once the chip is built, only a casualty will cause alteration of the stored data. In addition to specifying the data to be contained in the ROM, Apple also specifies which of the three **chip select** inputs are active high and which are active low. This is why the chip selects are said to be programmable.

The ROM connections in the Apple are shown in Figure 6.1. The ROM chips are wired in the way most microcomputer hardware is wired, with repetitious, identical wiring going to all six chips. It takes eleven lines to address 2048 bytes and the eleven address inputs to the ROM chips are connected directly to A0-A10 of the address bus. The eight tri-state data outputs are connected directly to the data bus. These address and data connections are pleasantly simple when contrasted to RAM connections.

The only remaining connections are power supply connections (+5V and GROUND) and the chip select inputs. The chip selects control the tri-state data outputs, enabling the outputs when all three chip selects are active. In other words, for an Apple ROM chip to take control of the data bus, its **CS1'** and **CS3'** inputs must be low, and its **CS2** input must be high. The CS2 input of all six ROM chips is tied to the wire-OR **INHIBIT'** signal from pin 32 of the peripheral slots. This reveals the power of the INHIBIT' line. If any peripheral card pulls it low, all motherboard ROM is isolated from the data bus. This means that any peripheral can respond to addresses $D000-$FFFF in any way it wants when it has pulled INHIBIT' low. The only function of ROM is to put data on the data bus. When it is isolated from the data bus, ROM may as well not be installed.

The CS1' and CS3' inputs of each ROM chip are tied together and connected to one of six address decoded **ROM ENABLE'** signals. These six signals are decoded directly from the address bus during PHASE 0 and divide ROM addressing up into six equal parts ($D000-$D7FF, $D800-$DFFF, etc.). Decoding of signals from a multiline input is a very common problem in digital design, so general purpose ICs exist which are tailor made to the task. Detecting the address range of each ROM chip and generating the data bus management signals for those six chips is accomplished by a single 74LS138 decoder and an AND gate.

Two other outputs from the LS138 represent the $C000-$C7FF and $C800-$CFFF address ranges during PHASE 0. The $C000-$C7FF signal is used to enable I/O decoding as described in Chapter 7 and illustrated in Figure 7.2. The $C800-CFFF signal is the **I/O STROBE'**, which is tied to pin 20 of all peripheral slots. This signal enables the "**seventh ROM chip**" which can be installed on any peripheral card or several peripheral cards. The $C800-$CFFF addressing is shared by all peripheral cards under a protocol described in the next section. Within this protocol, any card can activate response to addresses $C800-$CFFF, thus supplying its own 2K program in firmware. The I/O STROBE' at pin 20 is the the "C8" enable term for any card on which the "seventh ROM", or "expansion ROM", is active.

An interesting feature of the ROM connections in the Apple II is that ROM is not gated off by R/W'. This means that if a program writes to a ROM address, the enabled ROM will compete with the MPU bidirectional bus driver for control of the data bus. The possibility of "**bus fights**" like this is something microcomputer engineers try very carefully to eliminate from their designs. At least three bad things can happen in a "bus fight": hardware can be damaged; data transfer on the bus can be unreliable; and current spikes created on the power supply and ground lines by the "bus fight" can interfere with the operation of other circuits. It appears that none of these ill effects occurs when a program causes the MPU to write to a ROM address in the Apple.

The ICs involved are the 2316B ROM and the 8T28 (8304 in the RFI Revision) bidirectional driver. When the 2316B and an 8T28 or 8304 fight for control of a line, the 2316B loses. Surprisingly, even when the 2316B is trying to pull a line low, an 8T28 or 8304 can pull the line above 2 Volts. The current drawn during this bus fight is not sufficient to damage the 2316B, 8T28, or 8304, so hardware damage is not a likely result. The second possible ill effect, unreliable data transfer on the bus, is not a factor. Remember we are writing to ROM, an unusual programming action which is normally done only by accident (as when the programmer tries to write to a 16K RAM card but neglects to enable the RAM card). Data transfer does not exist when the MPU writes to ROM, because no device on the data bus is configured to receive data. The third possible ill effect, interference caused by current spikes, appears not to occur. I ran a test program for twelve straight hours on my Revision 3 Apple. The program continuously wrote the complement of the data at $E000 to that same address. No discernable operational malfunctions occurred during this experiment.

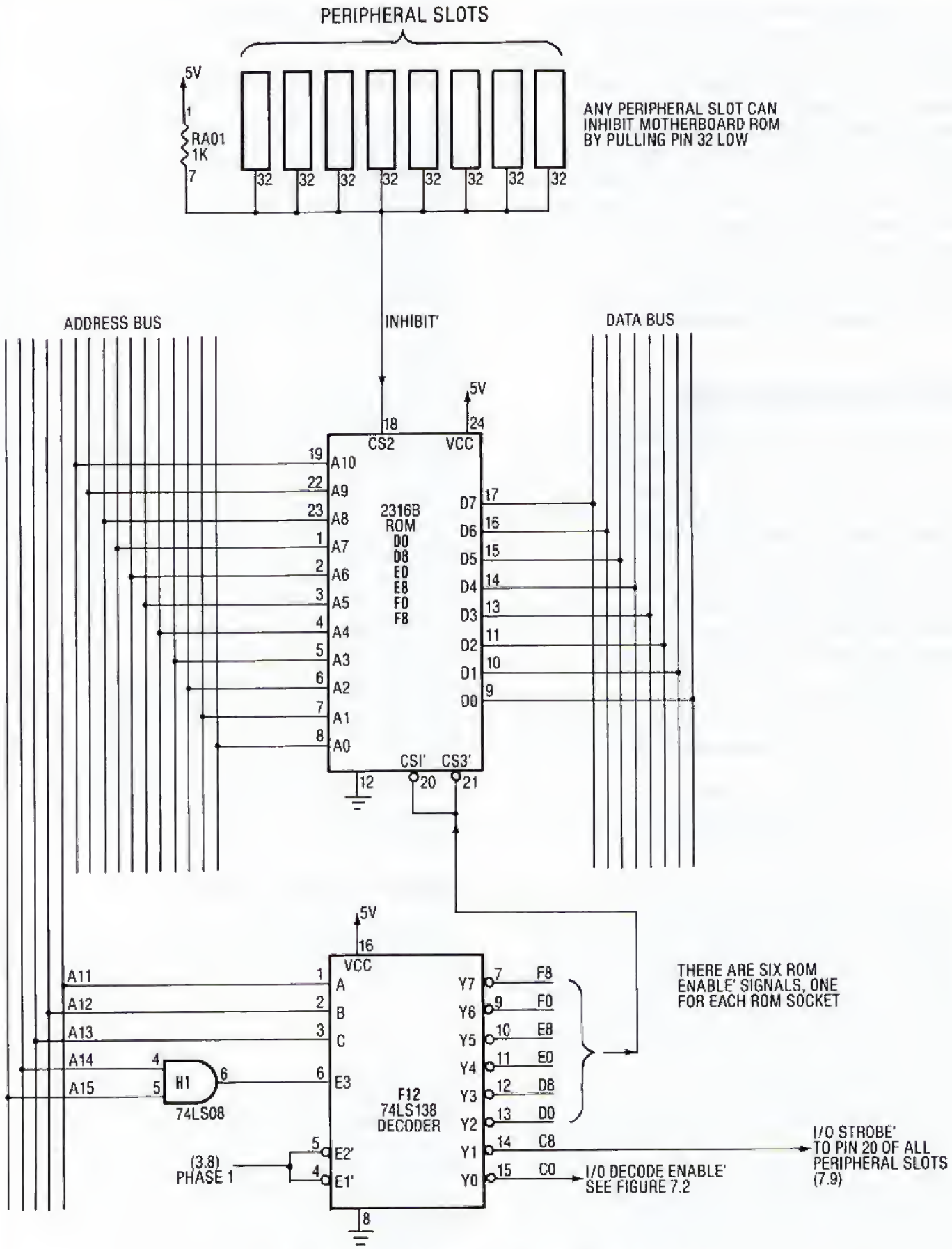


Figure 6.1 Schematic: ROM in the Apple II.

Even though there seems to be no real harm in it, purposefully writing to ROM addresses while the INHIBIT' line is high is not recommended. The outcome of "bus fights" may not always be absolutely predictable, and the Apple designer would have done well to prevent the possibility by connecting R/W' to one of the chip selects of the motherboard ROM. A parallel situation to that of ROM exists with the motherboard serial inputs. If a program stores data to address $C06X, both the MPU driver (8T28 or 8304) and the serial input multiplexor (74LS251) attempt to control D7 of the data bus at the same time. As with ROM, purposely writing to the serial input addresses is not recommended.

THE SEVENTH ROM CHIP

As Figure 6.1 shows, the I/O STROBE' signal at pin 20 of the peripheral slots is generated identically to the six ROM ENABLE' signals. In fact, I/O STROBE' is the ROM enabling signal for any peripheral card on which $C800-$CFFF is currently active. There are no motherboard control signals to tell the various slots when they may or may not respond to the I/O STROBE', so Apple Computer Inc. decided on the following protocol which cards responding to I/O STROBE' must follow:

1. When pin 1 (I/O SELECT') or pin 41 (DEVICE SELECT') goes low, a peripheral card may begin to actively respond to the I/O STROBE' at pin 20.
2. When $CFFF is on the address bus during PHASE 0, all peripheral cards must stop responding to the I/O STROBE'.

Actually, the *Apple II Reference Manual* says only to use I/O SELECT' to activate response to the I/O STROBE'. However, Slot 0 shares $C800-$CFFF addressing with the other slots and it has no I/O SELECT' input. The only way Slot 0 can activate response to I/O STROBE' is through its DEVICE SELECT' input at pin 41. It should be noted that Slot 0 designs which respond to the I/O STROBE' are either rare or nonexistent.

An example should clarify how the I/O STROBE' protocol works. Assume that Slot 1 and Slot 2 have peripheral cards installed and that both have a "seventh ROM" on board. A PR#1 is executed from BASIC which results in 6502 program flow vectoring to address $C100 for all character output. The card at Slot 1 responds to $C1XX addresses by placing a program on the data bus, and by activating response to the I/O STROBE'. Suppose that the program driven out at address $C100 begins with:

```
C100:  BIT $CFFF
C103:  JMP $C800
```

On the last cycle of execution of the first instruction, response to the I/O STROBE' is deactivated on all peripheral cards including Slot 1. However, on the first cycle of execution of the second instruction, $C1XX is back on the address bus and I/O STROBE' response is again activated at Slot 1. It is now safe to begin execution of programs stored in the "seventh ROM" at Slot 1. The ROM at Slot 2 will not interfere with Slot 1 access to $C800-$CFFF, because it is designed to ignore the I/O STROBE' after an access to $CFFF.

It is possible for a peripheral card to store its $CnXX program and its $C800-$CFFF program on a single 2K ROM. This is accomplished by enabling the output of the ROM to the data bus when I/O SELECT' (pin 1) goes low or when I/O STROBE' response is activated and I/O STROBE' (pin 20) goes low. If the card is located at Slot 1, the 256 bytes at $C1XX will be identical to the 256 bytes at $C9XX. The 256 bytes at $C1XX can be accessed at any time. The 2048 bytes at $C800-$CFFF can be accessed only when I/O STROBE' response is activated (after a $C1XX access).

The "seventh ROM" capability should not be confused with the capability to steal addresses $D000-$FFFF via the INHIBIT' line. The former gives a peripheral card unlimited access to 2048 bytes of addressing not used by motherboard devices. The latter gives a peripheral card access to 12,288 bytes of addressing at the expense of disabling motherboard firmware.

FIRMWARE IN THE APPLE

The hard features of the Apple ultimately determine its capabilities and limitations. However, the computer is only as powerful as the program controlling it at any given moment. Possibly the most important programs ever written for the Apple are the ones stored in the motherboard ROM. These give life to the machine and are used so often that we forget that they are just programs and that operational features created by any program can be changed.

The contents of Apple II firmware has been the subject of many writings which this book cannot hope to match in a limited space. The goal here is just to give an overview of the Apple firmware and to provide some insight into the importance of the programs contained there. The approach is historical.

First, there was Integer BASIC, the Monitor ROM, and some 6502 utilities. These programs were written primarily by Steve Wozniak, the designer of the original Apple computer. This was in the bad old days before the proliferation of inexpensive disk drives and before Microsoft Inc. started supplying all the computer companies with more sophisticated BASIC interpreters. Integer BASIC takes up about 5K of memory and has some very important limitations: no floating point arithmetic, no HIRES graphics commands, and no subscripted string variables, to name three. This is not said to belittle the considerable design accomplishments of Mr. Wozniak. It's just that within a few years, Integer BASIC became less than state-of-the-art.

The 6502 utilities included with Integer BASIC were some floating point arithmetic routines, the "SWEET 16" double word length command interpreter, and most importantly, the Mini-Assembler. **SWEET 16** is a small but sophisticated computer language which lets the programmer manipulate data in 16 bit word lengths. It utilizes RAM addresses $0 through $1F as 16 registers of 16 bits each and has normal machine language commands such as ADD, SUBTRACT, COMPARE, BRANCH, etc. Writing some programs in SWEET 16 will make them take less space than the equivalent 6502 program, but the 6502 program will run faster.

The **Mini-Assembler** is the utility on which uncounted numbers of Apple owners have first learned to program 6502 assembly language. Its use is fully described in the *Reference Manual*, and there is also some description in the 6502 programming section of Chapter 4 of this book.

SWEET 16 and the floating point routines are not described in any published Apple literature. In the old red *Reference Manual*, there are source/object listings of SWEET 16, the floating point routines, and the Mini-Assembler. SWEET 16 is fully described in the November, 1977 edition of *Byte* magazine ("SWEET 16: the 6502 Dream Machine" by Steve Wozniak).

The Monitor ROM

In microcomputer terminology, a system monitor is a program containing the most basic utilities of the system. Before the innovation of BASIC in ROM, a **monitor** in ROM was the primary user interface to the microcomputer. Some basic routines of a system monitor are keyboard input routines, video output routines, memory display and modification routines, and storage media input/output routines.

The Apple was one of the microcomputers which led the transition from a monitor in ROM to BASIC in ROM as the primary human to machine interface for home computers. However, the Apple II does have an extensive monitor in ROM, and the older Apples came up in the system monitor, not in BASIC. The system monitor is contained in the F8 ROM, and the F8 ROM was naturally called the **Monitor ROM**.

The Monitor ROM contained such important Apple utilities as entry to BASIC, keyboard input, video text and LORES graphics output, cassette I/O, assignment of different peripheral slots as primary input or output, memory display and modification, a 6502 disassembler, machine language single step, trace, or normal subroutine execution, handlers for RESET', NMI', IRQ', and BREAK, and some 16 bit multiply and divide routines. Control of the Monitor is via a highly usable command interpreter, the use of which is well described in the *Reference Manual*.

This, then, was the Apple: a cassette based system with a poor man's BASIC and a rich man's monitor in ROM and two empty ROM sockets for user firmware. Oh yes, one more important thing—Apple published a source/object listing of the Monitor ROM. This was a risky move which paid off immeasurably. By publishing their listing, Apple opened the way for investigators to learn thoroughly the nuts and bolts operation of the Apple. They thus aided competitors in developing numerous software and hardware applications for the Apple II. With this combination of extensive available applications and freedom of information, the Apple II pulls off the delicate trick of appealing not only to the masses who don't care how it works but also to more serious users who develop even more applications.

The contents of the Monitor ROM dictate many of the operational characteristics of the Apple: what happens when the computer is turned on or RESET is pressed, the format of the screen text, the nature of cursor moves, and the assignment of primary input and output devices. The monitor zero page assignments must be taken into account by software designers. Page 2 of RAM is thought of as the Apple keyboard input buffer because that is the way the monitor uses Page 2. Cards capable of being primary input or output devices must have programs at their I/O SELECT' addresses, because the firmware assigns a slot as a primary input or output by jumping to the slot's first I/O SELECT' address ($C100 for Slot 1, $C200 for Slot 2, etc.). These features are not inviolate in the Apple. They can be changed by

loading a new operating system into RAM or by replacing a single ROM chip, which is exactly what Apple did when they decided to change a few operational features of their computer.

The Apple II Plus

The Apple II evolved rapidly in the late 1970's into a more sophisticated machine than was originally introduced. The dynamic nature of the hardware and software support provided to the Apple II by Apple in 1978 and 1979 is remarkable. In approximate order,

1. Applesoft BASIC became available on cassette,
2. The Disk II was introduced with its powerful DOS,
3. Applesoft became available on diskette,
4. Applesoft became available on a firmware card,
5. The Apple II Plus was released with Applesoft and Autostart Monitor in ROM,
6. The Language System was released with 16K RAM card, Pascal language, and 16 sector disk capability, and
7. DOS 3.3 was released with its 16 sector capability (August 1980).

A popular Apple configuration became both Integer BASIC and Applesoft in ROM with automatic selection between the two by the DOS when a program was RUN. One BASIC resided in motherboard ROM, and the other resided in ROM on a Slot 0 **firmware card**. As this configuration demonstrates, bank switching of firmware operating systems is a very powerful concept.

The availability of **Applesoft BASIC** greatly improved the versatility of the Apple by making the manipulation of the HIRES screen, large disk text files, and floating point numbers practical. Unfortunate Applesoft weaknesses were incompatibility in command and memory usage with Integer BASIC, no AUTO numbering, no DSP (DiSPlay) command, and the absence of the 6502 Mini-Assembler. Also, the SWEET 16 interpreter and old floating point routines which are associated with Integer BASIC are not available with Applesoft.

The new **Autostart ROM** reflected the changing nature of the personal computer owner. It caused the Apple to come up in BASIC instead of the system monitor, gave the Apple the capability to boot a disk at power up, and greatly improved the ESCape mode cursor moves. In the process, the SINGLE STEP and TRACE investigative utilities for machine language programs and the 16-bit multiply and divide routines were removed. Programmers' utilities were thus sacrificed for improved operational features. The small businessmen gained a system that automatically loads and starts at power up, and the computer hacks lost the convenience of STEP and TRACE in ROM.

The Impact of the RAM Card

A more recent development in the evolution of the Apple II has been the popularity of the **16K RAM card**. With a disk based system, as the Apple is now, it is no longer as necessary to have extensive operating systems in ROM as it was with a cassette based system. The entire Integer BASIC program and associated utilities can be loaded into the $E000-$F7FF area of the 16K card in a tolerably short period of time, giving the user the equivalent of the firmware card but more versatile. The system monitor becomes alterable by the user and the number of operating systems that may possibly reside in high memory becomes unlimited. The disadvantages of the 16K RAM card are the possibility of overwriting high memory, occasional extra waiting for loading the program into the 16K RAM card from disk, the impossibility of protecting 6502 vectors from program encrypting artists, and the lost capability of other peripheral cards to respond to $F800-$FFFF addressing. For most users the advantages of the 16K RAM card outweigh the disadvantages.

So now we have the Apple II as it was before the introduction of the IIe in January, 1983: Applesoft and system monitor in ROM with alternate operating systems in firmware or RAM peripherals, disk based, and who knows what other peripherals plugged in. A clean machine.

ROM TIMING

ROM read timing is very simple and similar to a read to any address above $C00F. The two main timing signals involved are RAM SELECT' and the ROM ENABLE' signal for the addressed ROM chip. The important specifications of the 2316B ROM are:

1. The output data will become valid a maximum of 450 nanoseconds after the address input becomes valid.
2. The output data will become valid a maximum of 120 nanoseconds after the three chip selects become active.
3. The data output will go to high impedance a maximum of 100 nanoseconds after one of the chip selects becomes inactive.

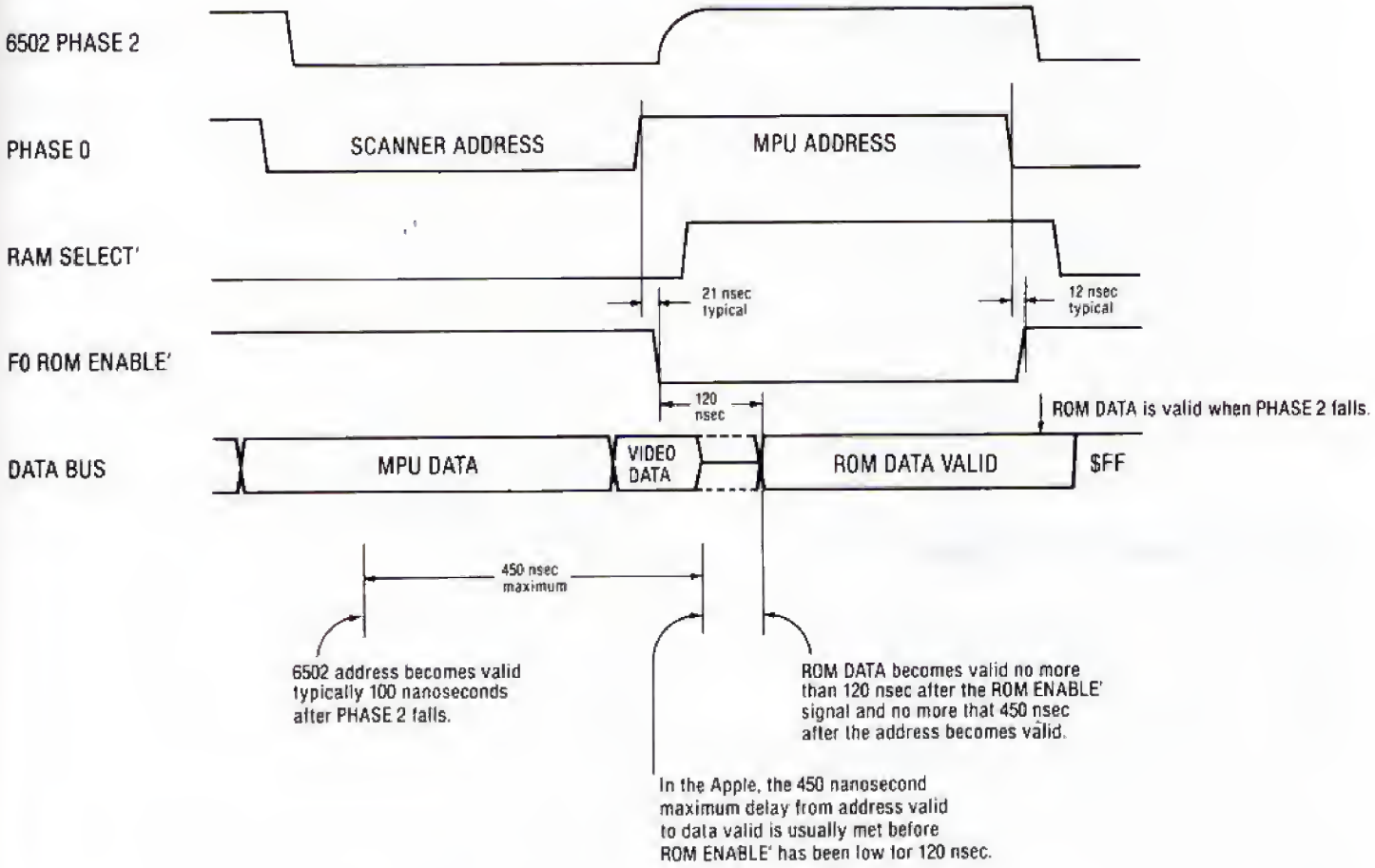


Figure 6.2 Timing Example: ROM Read, Address $F000.

ROM read timing in the Apple is illustrated in Figure 6.2. The main order of events is:

1. During PHASE 1, the video scanner performs its normal access to RAM.
2. At approximately 100 nanoseconds after 6502 PHASE 2 falls, the MPU address becomes valid. ROM data will be valid 450 nanoseconds from this point if the ROM ENABLE' input has been low sufficiently long.
3. PHASE 0 rises, causing the ROM ENABLE' signal to drop and the RAM SELECT' to rise. It takes longer for RAM SELECT' to rise because PHASE 0 is propagated through several logic devices to RAM SELECT'. The data bus floats, storing the video data until the ROM data becomes valid.
4. 120 nanoseconds maximum after the ROM ENABLE' signal falls, the ROM data becomes valid, assuming the address bus became valid early enough. With a typical 6502, the address becomes valid early enough that the chip select to data valid delay of the ROM chip will determine when data becomes valid. In any case, ROM data becomes valid well before 6502 PHASE 2 falls.
5. PHASE 0 falls, causing the ROM ENABLE' signal to rise, followed by 6502 PHASE 2 falling, followed by RAM SELECT' falling. The time differences in the three events caused by PHASE 0 falling are the result of different propagation delays. After ROM ENABLE' goes high, the ROM chip holds data valid for a maximum of 100 nanoseconds. If the ROM chip

stops holding data valid before PHASE 2 falls, the floating data bus will hold the ROM data valid anyway until after RAM SELECT' falls. This ensures that PHASE 2 falling correctly clocks the ROM data to the MPU. In the unlikely event that the ROM chip actually tries to control the data bus for 100 nanoseconds after ROM ENABLE' rises, it will fight with the RAM/keyboard data multiplexor for control of the data bus for about 30 nanoseconds. When the RAM/keyboard data multiplexor actually does take control of the data bus, it will bring the data bus to $FF as it always does following a read access to an address above $BFFF. As was noted in the chapter on RAM, this is because the RAM data latch interprets the floating output of the RAM chips as $FF (all ones).

THE FIRMWARE PERIPHERAL CARD

Not long after Applesoft was introduced on cassette tape, it became available on the Applesoft firmware card. Then, when Apple started putting Applesoft in the motherboard ROM, Integer BASIC became available on a firmware card. These cards are supported by DOS 3.2 and DOS 3.3 in an amazingly usable manner. One simply types "RUN program name" and the DOS automatically switches to the language that "program name" is written in and loads and runs the program. Unlike possibly any computer system prior to the Apple, this switching of high order languages is accomplished almost instantly by disabling one bank of ROM and enabling another.

This switching is made to seem even more amazing than it is, because Apple doesn't supply a hint of how the firmware card works in any of their documentation. Here then, possibly for the first time in print, is a description of the capabilities and limitations of the firmware card.

Figure 6.3 is a schematic diagram of the firmware card. I prepared Figure 6.3 after analyzing an Applesoft card with my eyes and an ohmmeter. Please refer to this figure during the following discussions.

Enabling the Firmware Card

The first thing to notice about the firmware card schematic is that there are six ROM chips with enabling signals generated by a 74LS138, just like on the motherboard. As on the motherboard, the LS138 is enabled when A14 and A15 are high during PHASE 0 to decode ROM ENABLE' signals for the six ROM chips. Unlike the LS138 on the motherboard, the LS138 on the firmware card is also gated by R/W', the DMA priority chain input, and the output of a 74LS74 flip-flop. Any of the six ROM ENABLE' signals at the output of the LS138 can drop low only if all of the enabling criteria are met. Again, these criteria are:

DMA priority chain input	-	high
R/W'	-	high
A14	-	high
A15	-	high
PHASE 1	-	high
flip-flop B2-B	-	reset

The reasons for some of these criteria are easy to explain. A14 and A15 being high defines the address range $C000-$FFFF. LS138 outputs corresponding to the $C000-$CFFF range are not connected. Enabling firmware card ROM only when R/W' is high prevents a fight for control of the data bus between the firmware card and the MPU bus driver in case a write to a ROM address is accidentally performed. PHASE 1 low or PHASE 0 high defines the time when all MPU access to Apple devices occurs.

The two other enabling criteria require more detailed explanation. B2-B is the enable/disable flip-flop for the firmware card. When B2-B is set, motherboard ROM is enabled. B2-B gets set when a RESET occurs and the switch in the back of the card is down, or when the MPU accesses any odd address between $C081 and $C08F (assuming the firmware card is in Slot 0). When B2-B is reset, firmware card ROM is enabled. B2-B gets reset when a RESET occurs and the switch in the back of the card is up, or when the MPU accesses any even address between $C080 and $C08E. Thus, the firmware card enable function can be controlled from software or by the position of the switch in the back when a RESET occurs. When controlled from software, the switching is clocked by the trailing edge of DEVICE SELECT' a few nanoseconds after PHASE 0 falls. This insures that switching always occurs when all the ROM ENABLE' signals are high. Switching in the middle of a ROM access could cause a program to crash.

The combination of DOS and the Autostart ROM completely defeats the functions of the switch in the back of the firmware card. When a power-up RESET occurs, the firmware card is first enabled or disabled as a function of the switch in the back, but the DOS soon enables or disables the card based on the language of the "HELLO" program. If RESET

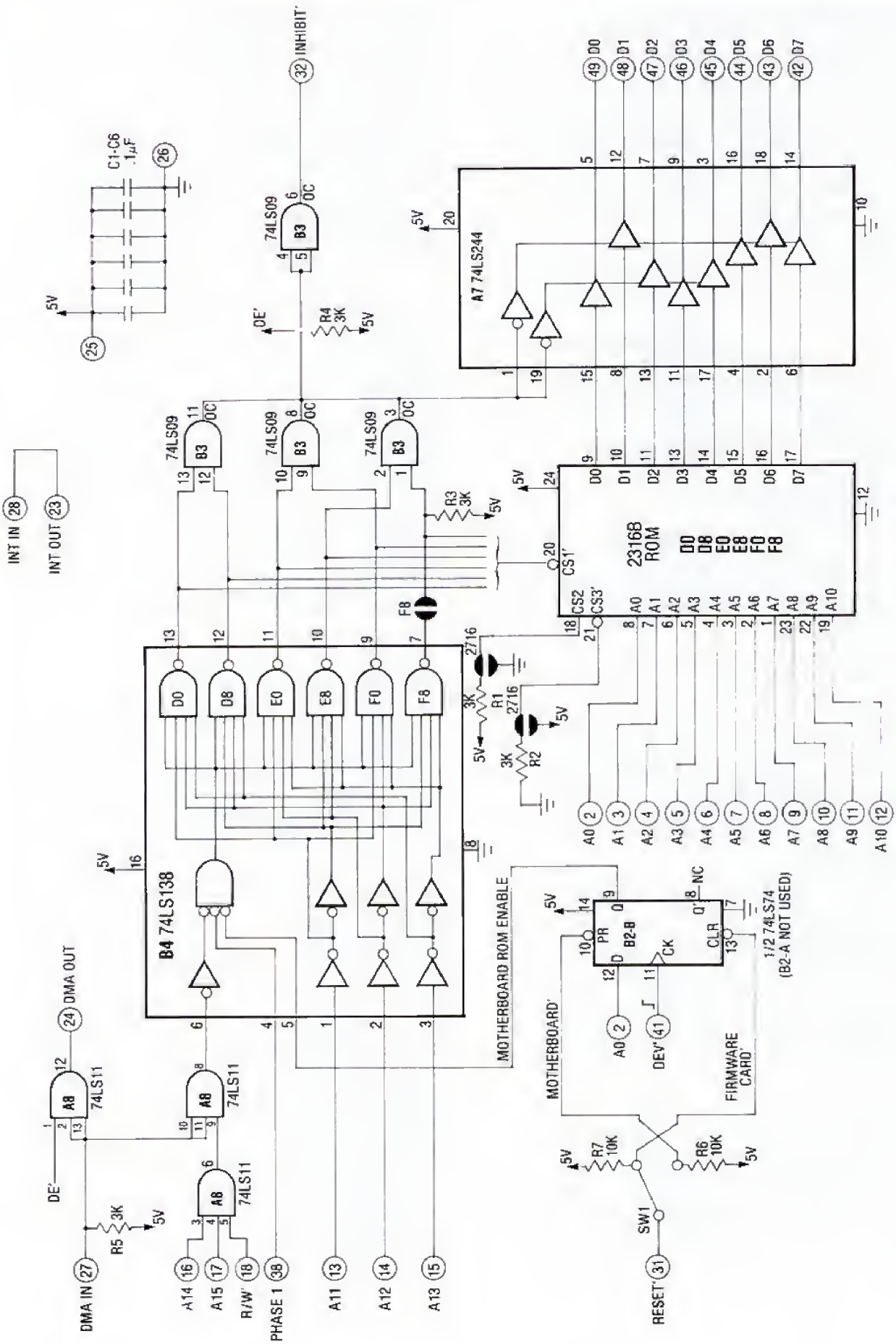


Figure 6.3 Schematic: The Firmware Card.

is pressed and the DOS is linked up, the firmware card again is enabled or disabled as a function of the switch in the back, but the DOS shortly reenters the last BASIC that was active. The result of all this is that the switch in the back usually serves no function in a disk based Apple with an Autostart ROM. In systems with no disk drive, it determines the power-up BASIC and provides an inconvenient method of switching back and forth between Integer and Applesoft.

Even though any addresses between $C080 and $C08F may be used to control the Slot 0 firmware card, the DOS always uses $C080 and $C081 to switch back and forth. The DOS doesn't care what ROM is in the motherboard and what ROM is in the firmware card. It checks both ROMs to see if the BASIC it is looking for is available. The firmware card will operate in any slot, but the DOS only supports it in Slot 0.

The DMA Priority Chain

The final enabling criteria for the LS138 is that the DMA priority chain input be high. This is because the firmware card was designed so that you can have several of them in adjacent peripheral slots. It uses the DMA priority chain to make sure that two or more firmware cards do not accidentally become enabled at the same time. This is accomplished as follows:

1. When the DMA' priority input is pulled low, all ROM ENABLE' signals on a firmware card go high and the firmware card is isolated from the data bus. Also, the DMA priority output is pulled low, so all lower priority firmware cards are also inhibited.
2. If the DMA priority input is high and a firmware card is enabled (B2-B reset), an MPU read to a ROM address results in one of six firmware card ROM ENABLE' signals going low. This causes the DMA priority output to drop low, disabling lower priority firmware cards. Also, motherboard ROM is inhibited, and the data from the addressed ROM chip is gated to the data bus.

The firmware card does not perform DMA. It implements bank switching of addresses between $D000 and $FFFF. However, because of its use of the DMA priority chain, the firmware card will interfere with lower priority DMA cards, and it will be interfered with by higher priority DMA cards. This can be prevented by keeping an empty slot between a firmware card chain and a DMA card chain or by modifying the leading or trailing card in one of the chains so the lower priority chain ignores its priority input.

By using the DMA' priority chain to prioritize the firmware card, Apple implicitly stated that there is a priority by which the peripheral slots or the motherboard may respond to addresses $D000-$FFFF. Slot 0 has the highest priority, and the motherboard has the lowest priority. Then, in violation of their own implied guidelines, they produced the 16K RAM card which steals ROM addressing without supporting the DMA priority chain. This means that if one elected to put a RAM Card in Slot 0 and a firmware card in Slot 1, there would be no fail-safe method of preventing accidental simultaneous enabling of the two cards. In any case, the DOS does not support firmware cards anywhere but in Slot 0, so one would have to patch the DOS to achieve automatic selection of firmware other than in Slot 0. Hardware capabilities not supported by software are of little use. Also, the concept of a priority chain of 12K firmware cards is a bit obsolete. With today's ROM, you could easily put 64 kilobytes of firmware on one card.

The ROM ENABLE' Signals

When one of the six ROM ENABLE' signals goes low, it means that the firmware card is enabled and the MPU is addressing ROM. These ROM ENABLE' signals are each tied to the CS1' input of one of the six ROM chips. The ROM data from that chip then becomes valid within 120 nanoseconds. The data is routed to the data bus through a 74LS244 line driver, which is gated on by any ROM ENABLE' signal dropping low.

The fact that one of the ROM ENABLE' signals is low is detected by a six input, low level OR gate consisting of three open collector AND gates on B3. This detected signal is labeled DE' (Data Enable') in Figure 6.3. Besides gating the LS244 outputs to the data bus, DE' also causes INHIBIT' and the DMA priority output to drop low. Dropping INHIBIT' isolates the motherboard ROM from the data bus, so the firmware card data is read by the MPU. Note that DE' and INHIBIT' are not identical logic terms. DE' is the local "data enable" to the data bus, but INHIBIT' is the low level wire-OR gate of the pin 32 output of all eight peripheral slots.

The LS244 is not necessary for operation of the firmware card. You could remove the LS244 and short all the outputs to their respective inputs on the empty socket, and the firmware card would still work. Apple probably included the LS244 in the design to reduce data bus loading. A gated OFF LS244 loads the data bus less than six gated OFF 2316B ROMS. Notice that the address bus is not treated with such tender respect. A0-A10 are connected directly to the six ROM chips and A0 is additionally connected to the D-input of B2-B.

Firmware Card Jumpers

There are three solder pad jumpers on a firmware card. Two of these jumpers configure the firmware card for Apple ROM or 2716 EPROM. Apple ROM chip selects are not programmed to be compatible with 2716 EPROM. By placing a touch of solder on the two jumper pads labeled 2716, the owner can remove the ROM and replace it with EPROM containing programs of his choice. This is not a bad idea for people who have replaced their firmware card with a RAM Card. Put your utilities in EPROM in an empty peripheral slot.* Note that the jumper pads allow you to configure the firmware card as all EPROM or all ROM. It is very easy, however, to modify the card's wiring so some of the sockets are EPROM configured and some are ROM configured.

The third solder pad jumper on the firmware card is labeled F8 on the card. If this jumper is soldered, the F8 ROM on the firmware card will be enabled or disabled with the other ROMs on the firmware card. If this jumper is not soldered, the F8 ROM on the firmware card will never be enabled. Instead, the motherboard F8 ROM will remain active whether the firmware card is enabled or not. Apple has delivered firmware cards in at least two configurations: with F8 jumpered and the Autostart ROM installed, and with F8 not jumpered and no F8 ROM installed.

The F8 jumper gives the user some options for configuring his Apple. If the owner has only one F8 ROM, he should plug it into the motherboard F8 socket and leave the F8 pad on the firmware card unsoldered. Suppose he has an Autostart ROM and an old Monitor ROM. If he wants, he can plug one into the firmware card and plug the other into the motherboard. One such configuration would be that any time the computer was in Applesoft, the Autostart ROM would be active; and any time the computer was in Integer BASIC, the old Monitor ROM would be active. This would be desirable in that the best features of both ROMs would be available. It would be undesirable in that the owner would have to put up with the old cursor moves when in Integer BASIC. An Application Note at the end of this chapter shows how to modify the firmware card so that the F8 ROM is selectable independently of the other five ROMs. By an easy modification, the owner truly gets full access to the best features of both ROMs.

*When a RAM card and firmware card are installed in an Apple II at the same time, either the F8 jumper of the firmware card should be unsoldered, or the RAM card should be modified as described in the Application Note at the end of Chapter 5. See "Hardware Application: Multiple RAM Card Configurations."

Firmware Card Timing

Timing for reading from the firmware card is very similar to reading from motherboard ROM. The major difference is that the data bus acts differently when the firmware card is read because of the LS244 buffering ROM from the data bus. The important point is that the ROM data must still be valid on the data bus when 6502 PHASE 2 falls if it is to be correctly read by the MPU.

The timing of a read from ROM on the firmware card is shown in Figure 6.4. The important events that are different from a motherboard ROM read are:

1. PHASE 0 rises, followed by a ROM ENABLE' signal falling, LS244 DATA ENABLE' falling, DMA priority output falling, INHIBIT' falling, and RAM SELECT' rising. All of these events are gated by PHASE 0, but they differ in time of occurrence because of different propagation delays. It is interesting that the LS244 ENABLE' signal drops low about 20 nanoseconds before RAM SELECT' rises, causing the LS244 to compete with the RAM/keyboard data multiplexor for control of the data bus for a short period.
2. Within 120 nanoseconds of ROM ENABLE' falling, the ROM data becomes valid and is propagated through the LS244 to the data bus. This assumes that the 6502 address became valid about 100 nanoseconds after PHASE 2 fell.
3. PHASE 0 falls, followed by ROM ENABLE' rising, DATA ENABLE' rising, 6502 PHASE 2 falling, the DMA priority output rising, INHIBIT' rising, and RAM SELECT' falling. Again, all these events are triggered by PHASE 0 falling. The data bus is floated by DATA ENABLE' rising approximately at the same time PHASE 2 falls. As normal, the floating data bus keeps the ROM data valid until after RAM SELECT' rises.

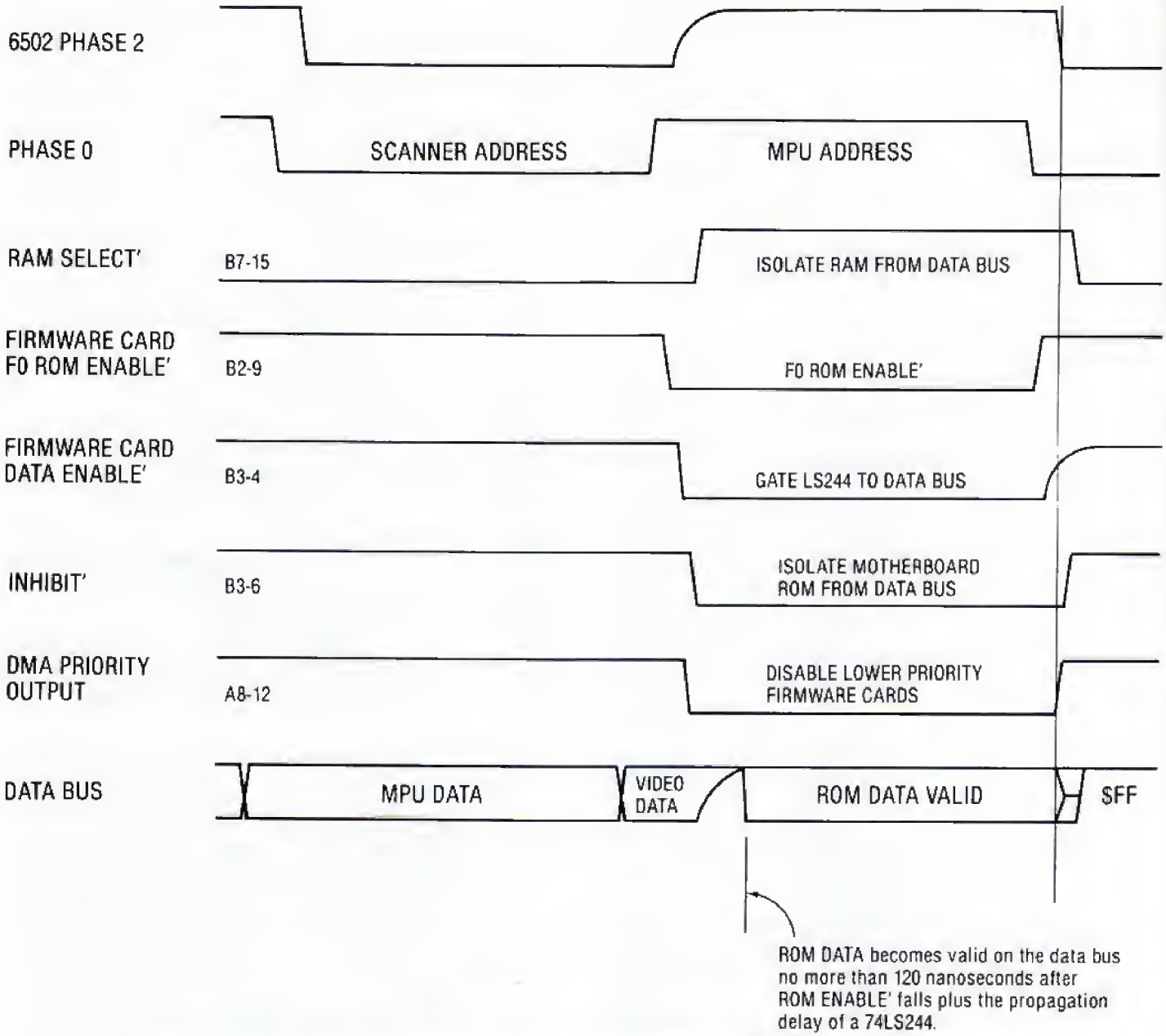


Figure 6.4 Timing Example: A Read From Address $F000, Firmware Card Enabled.

HARDWARE APPLICATION

EPROM IN THE APPLE

EPROM is **Erasable Programmable Read Only Memory**. It is like ROM except that it is user programmable and user erasable. The program in ROM is placed there in the manufacturing process, never to be altered. The program in EPROM is placed there by an EPROM programming device, never to be altered unless you shine ultraviolet light through a little window in the top. Like ROM, it is nonvolatile memory, and the programs stored there are always ready to run the instant you turn your computer on.

The 2316B ROMs used in the Apple II are physically and electrically compatible to 2716 EPROMs. This means that the owner can burn his own firmware and plug it into the ROM sockets of the Apple—almost. It turns out that the Apple's ROM chip selects are incompatible with the 2716 chip selects. Remember that the chip selects of the 2316B are specified as active high or active low by the buyer (in this case, Apple Computer, Inc.). The chip selects of the 2716 are not programmable. Pin 18 is active low, pin 20 is active low, and pin 21 is active high. Thus Apple's ROM sockets are incompatible with 2716 EPROM. Unfortunately the 2716 had just come on the market when the Apple II was first designed, and the desirability of pin compatibility with the 2716 was not forseen by the Apple designer.

You can still use the 2716 EPROM in the Apple motherboard sockets if you plug them in through an adapter. The adapter is made by taking two 24-pin DIP sockets and modifying them as shown in Figure 6.5. This adapter is very cheap but will take about an hour of your time to construct if you are as slow as the author when working carefully. A ready made adapter is available for about $10 from a company named Microproducts, whose address is given at the end of this Application Note. The adapter can be used on the firmware card as well as on the motherboard.

Texas Instruments and Motorola make a 2716 EPROM which is not compatible with Apple ROM or Intel type 2716 EPROM. The TMS2716 requires +12 volts on pin 19. The correct 2048 x 8 EPROMs to order from these companies are the Texas Instruments TMS2516 and the Motorola MCM2716.

There is a new type of EPROM called the EEPROM (Electrically Erasable and Programmable Read Only Memory). The EEPROM requires no ultraviolet light source for erasure. Instead, it is both programmed and erased with combinations of +25 and +5 volts. These are the same voltages required for programming an ultraviolet EPROM, and they are required on the same pins. For this reason, some programmable 2716 PROM burners used in the Apple should be able to program and erase 2716 compatible EEPROM like the Hitachi HN48016P. This appears to be the case with the author's PROM burner (made by John Bell Engineering), judging from its schematic diagram. Of course, the controlling software for burning and erasing EEPROM must be different. With the coming of EEPROM, we may soon see computers capable of programming and erasing resident firmware.

2716 EPROM burners for the Apple are available from several companies in the $100-$200 price range. These EPROM burners dump a 2K block of memory to EPROM in about a minute and a half. Also available are inexpensive cards which have multiple sockets for 2716 EPROMs and make use of the $D000-$FFFF bank switching or $C800-$CFFF "seventh ROM" capability. These cards allow EPROM users to have the convenience of permanently accessible user firmware. The Applesoft or Integer Card may also be configured to accept 2716 EPROM instead of ROM. Another place to use EPROM is in the text character generator (socket A5 on the motherboard). The ROM at this socket contains the dot patterns for Apple screen text. In Revision 7 or later Apple IIs the A5 socket is compatible with 2716 EPROM and the owner can install his own upper case/lower case screen character set.

Jeffrey Mazur wrote a nice article about EPROM and the Apple for his "Hardtalk" column in *Softalk* magazine. The article, which appears in the July and August 1982 *Softalk* issues, contains product reviews and a thorough discussion of EPROM. This article is recommended for Apple owners wishing to learn more about EPROM.

Listed below are some companies which manufacture products related to using EPROM in the Apple:

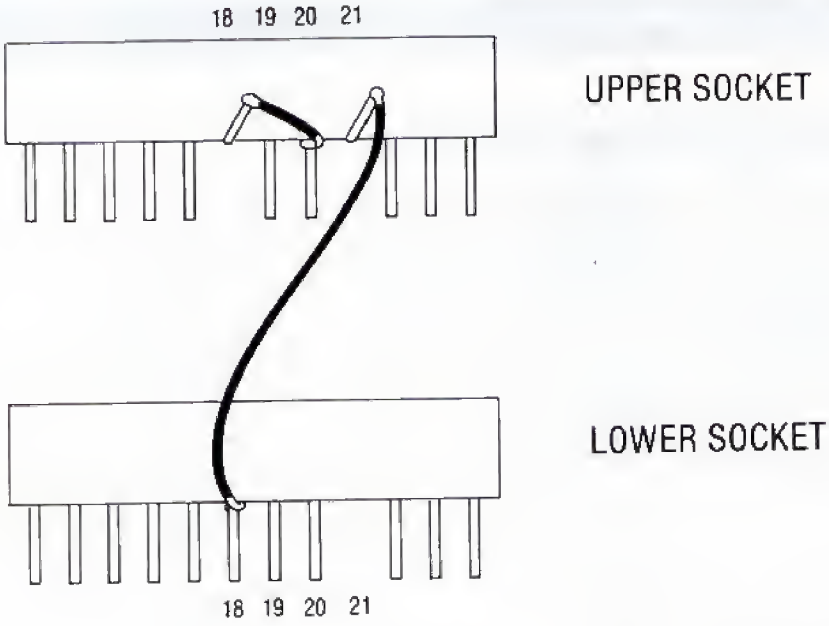
John Bell Engineering, Inc.
1014 Center St.
San Carlos, Ca. 94070
415-592-8411

Microproducts
24627 Watt Road
SDCE
Ramona, Ca. 92065
714-789-6510

Mountain Computer
300 El Pueblo
Scotts Valley, Ca. 95066
408-438-6650

Soft CTRL Systems
Box 599
West Milford, N.J. 07480
201-728-8750

Word Power
P.O. Box 736
El Toro, Ca. 92630



1. Use two 24 pin DIP sockets. If pin 1 is not clearly marked on the sockets, choose pin 1 and mark it with fingernail polish or other permanent marking.
2. All pictured wiring is shown in phantom. Pins are bent inward, and wiring is routed on inside of sockets. Use 30 gauge insulated wire and keep wire lengths as short as possible.
3. Remove pin 21 from lower socket. Save this pin as a spare.
4. Bend pins 18 and 21 of the upper socket inward so they will not make contact with the lower socket when the sockets are mated. This must be done carefully so the EPROM will still make contact with the bent pins when it is inserted into the upper socket.
5. Solder a short jumper between bent pin 18 and the base of pin 20 in the upper socket. Carefully dress the jumper at the base of pin 20 so it will not interfere with the mating of the two sockets.
6. Solder a short jumper between bent pin 21 on the upper socket and the base of pin 18 on the lower socket. Carefully dress the jumper at the base of pin 18 so it will not interfere with plugging the completed adapter into a ROM socket.
7. Push the upper socket into the lower socket. Inspect visually or with an ohmmeter to make certain there is no contact between pins 18 on the upper and lower socket. With some sockets it may be necessary to modify or insulate pin 18 on the lower socket so that it does not touch pin 18 on the upper socket. This must be done carefully so pin 18 is still firmly mounted in the lower socket.

Figure 6.5 Construction of a Socket Adaptor, EPROM to ROM.

HARDWARE/SOFTWARE APPLICATION

MODIFYING THE SYSTEM MONITOR

The system monitor determines many of the operational features of the Apple, and modifying the system monitor is one way of enhancing the Apple. The monitor resides in the F8 ROM and in the $F800 to $FFFF addressing space of a 16K RAM card. It is easily modified in RAM by the user and by commercial programs, so modifying the monitor in RAM will give you access to your personal utilities but won't necessarily help you investigate programs written by others. If you want an unalterable modification, you must make it in EPROM and have your Apple configured so that you can override software control to enable your EPROM.

The Autostart ROM contains a fine system monitor with some good utilities that you probably don't want to delete. However, here are two parts of the Autostart ROM which may not be too critical to you:

1. Addresses $FCC9-$FD0B, $FECD-$FEF5, and $FEFD-$FF2C are cassette read/write routines. There is no reason to have these routines in firmware if you own a disk drive and don't usually use cassette I/O. Modify the cassette routines to run at a RAM address and save them on diskette. This generates 156 bytes for your personal firmware. This can be increased to 164 by moving the control-Y jump statement to $FEC2. This is accomplished by storing $4C, $F8, and $03 at $FEC2-$FEC4 and by storing $C1 at $FFE4.
2. There are 14 consecutive unused bytes at $FBB3-$FBC0. This can be increased to 18 by storing $B0, $EC at $FBAD and $FBAE. This makes $FBAF-$FBC0 unused and causes only a minor operational change. The change is that pressing L will not cause the Apple to leave the ESCAPE mode. Even more space could be generated by rewriting the ESCAPE handler so ESCAPE-A,B,C, and D were not recognized. These old cursor moves are not needed because far better cursor moves are included with Autostart ROM.

The idea in making a minor modification to the Autostart ROM is to create an EPROM that is identical to the Autostart ROM except for small areas which contain your data. One idea for modification is to increase the command repertoire of the monitor. The deletion of STEP and TRACE routines from the system monitor opened space for two commands in the command tables (CHRTBL and SUBTBL) of the monitor. If you delete the cassette routines, that will also open up two more command spaces since the READ and WRITE commands will no longer function.

What sort of commands can be installed? Here are some possibilities:

1. Breakpoint insert and breakpoint remove commands which facilitate the use of the 6502 BREAK instruction as a debugging breakpoint.
2. Hex to decimal and decimal to hex commands.
3. Commands that link to large routines in the spare ROM space of Integer BASIC, even if Integer BASIC is not currently enabled.
4. "Click on" and "click off" control of a keypress click simulator.
5. Commands that enter Applesoft, Integer, Pascal, or CP/M.
6. Commands that connect and disconnect the DOS.
7. A command to transfer the DOS from EPROM on a firmware card to RAM.
8. A dump screen to printer command that links to your printer driver on the "seventh ROM."

The possibilities are endless.

You can modify the system monitor without changing the command table. One idea is to change the ESCAPE handler to recognize special functions. For example, you can assign ESCAPE-G to Graphics, ESCAPE-T to Text, and so on to give yourself control of the screen modes from BASIC, the monitor, and many other keyboard polling programs.

Here is an example of a change to the Autostart ROM that does not delete any routines or capabilities but would still benefit the Apple user. It modifies the RESET routine to begin by checking to see if the last keyboard entry (not including the RESET key) was "CONTROL-SHIFT-M." If the last keypress **was not** "CONTROL-SHIFT-M," then the normal Autostart RESET is performed. If the last keypress **was** "CONTROL-SHIFT-M," then the old monitor RESET is performed. "CONTROL-SHIFT-M" stands for **Monitor** and is not likely to be pressed accidentally. With this modification in firmware, the

user may cause a RESET entry to the monitor anytime he wishes, but the normal RESET is still the Autostart RESET. This modification works because pressing RESET does not change the state of the keyboard latch (read at $C00X) and because a program cannot modify the contents of the keyboard latch.

To create an EPROM with this modification, perform the following steps:

1. Using the monitor MOVE command, transfer the Autostart ROM contents to the area of RAM used as the 2K output buffer by your PROM burner.
2. Store $AF and $FB at the addresses corresponding to $FFFC and $FFFD. This changes the 6502 RESET vector to $FBAF.
3. Beginning at the address corresponding to $FBAD, store this program:

As a final word of advice, you should tag your modified monitor so you can recognize when it is active. It can be tagged audibly by changing location $FBE5 from $0C to $16. This noticeably lowers the pitch of the Apple's BELL so that you can verify your EPROM is active by pressing CONTROL-G or RESET. You can tag your EPROM visually at power-up by changing the contents of $FB09-$FB10 to an ASCII message of your choice. For example, instead of "Apple II" you might have your screen display "KAZOO II" or some other pertinent title.

```
FBAD:  B0 EC       BCS $FB9B   ;Save four bytes by not
                               ;looking for ESCAPE-L.
FBAF:  8D 10 C0    STA $C010   ;Look for CONTROL-SHIFT-M
FBB2:  AD 00 C0    LDA $C000
FBB5:  C9 1D       CMP #$1D    ;CONTROL-SHIFT-M?
FBB7:  D0 03       BNE $FBBC
FBB9:  4C 59 FF    JMP $FF59   ;Do old monitor RESET
FBBC:  4C 62 FA    JMP $FA62   ;Do Autostart RESET
```

HARDWARE APPLICATION

MODIFYING THE FIRMWARE CARD FOR INDEPENDENT SELECTION OF THE F8 ROM

The importance of the F8 Rom to the Apple should be well understood by the reader. The RESET interrupt and Non-Maskable Interrupt vectors reside there and determine who has ultimate control of the Apple, the owner or the author of the currently operating program. Moreover, the system monitor determines much of the personality of the Apple. This becomes very apparent to people who owned the Apple when the Autostart ROM came out. The Autostart Monitor and the old monitor are 90% identical, but with the latter the Apple is more of a hacker's computer, and with the former it is more of an appliance.

There are a good number of the old Monitor ROMs lying around gathering dust, because standard Apple hardware does not facilitate independent switching of the F8 ROM. Certainly this switching **is** a capability of the Apple—they just haven't designed the peripherals to do it. The purpose of this Application Note is to show some schemes for modifying your firmware card to add some versatility to F8 ROM switching. Figure 6.3 should be referred to while reading this Note.

Remember from the firmware card discussion that the F8 ROM on the firmware card can be disabled or enabled via the F8 solder jumper. Suppose you wired an on/off toggle switch across the F8 solder jumper as shown in Figure 6.6a. This is the simplest way to achieve selectability between the Autostart ROM on the motherboard and the old Monitor ROM on a firmware card in Slot 0. The old Monitor ROM is enabled when the firmware card is enabled and the F8 switch is on. Otherwise the Autostart ROM is enabled. Normal operation with this setup is with the F8 switch off, enabling the Autostart ROM. To enter the old monitor, turn the F8 switch on, lift the firmware card enable switch and press RESET. For sure, this will get you an asterisk and a blinking cursor on the screen, just like the good old Apples. You then may do as you like with the Apple, including initiation of STEP and TRACE commands from the old monitor. You can also force the Apple to power up in the old monitor by having the F8 switch on and the firmware card enable switch up.

There is one problem with this modification. Suppose that the firmware card is enabled and you flip the F8 enable switch just before PHASE 0 falls during an MPU access to an F8 address. It is possible for 6502 PHASE 2 to fall while the data bus is in an invalid state, causing a program crash. This would not normally happen, but it will happen once in a while. The crash occurrence rate will be inversely proportional to the quality of your switch since there is no debouncing circuitry. Because of switch bounce, throwing the F8 enable switch while the firmware card is selected can result in multiple toggles between the motherboard and firmware card F8 ROM. This does no harm in itself, because the programs that will be running when the switch is thrown are identical in the Autostart Monitor and old monitor. However, multiple toggles increase the probability that F8 control will be switched during the critical period just before PHASE 0 falls.

If, in spite of the occasional crash, you elect to install this easy modification, here are some installation suggestions:

1. Mount the switch in the bottom of a cable slot in the back of the Apple case. This gives reasonable access to a switch you won't throw all that often. It would be difficult to mount a second switch on the back of the card.
2. Connect the switch to the board using a two pin jack/plug combination. This way you can easily disconnect the switch and remove the firmware card from its slot. One way to do this is to take short speaker extension cable and cut it in half. Solder the cut end of one cable half to the switch. Solder the cut end of the other cable half to the firmware card.

With a few extra wires, it is possible to synchronize the switching of the F8 ENABLE' signal. This is accomplished by wiring up the unused half of the 74LS74 dual flip-flop at B2. The wiring is shown in Figure 6.6b. This modification is functionally identical to the one shown in Figure 6.6a, but there is no possibility of invalid ROM data being read by the MPU. The flip-flop synchronizes the enabling and

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

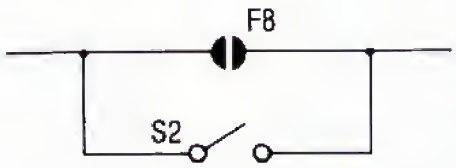


Figure 6.6a An Added Switch to Control the F8 ROM on the Firmware Card.

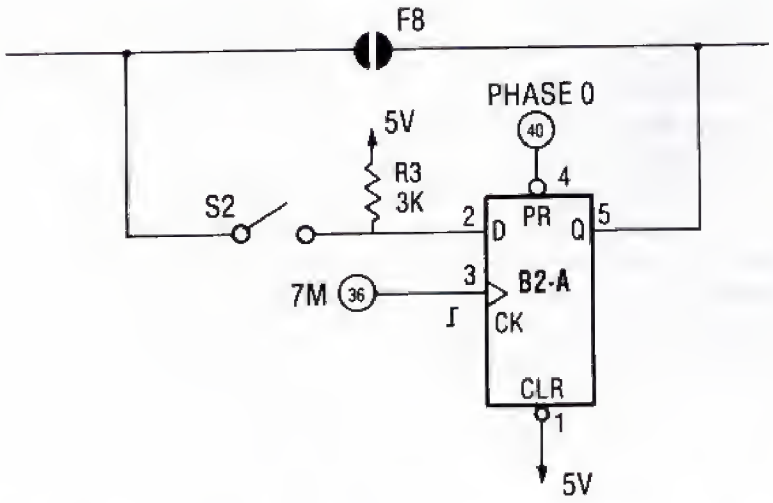


Figure 6.6b This Circuit Syncronizes Switching to Prevent Possible Program Crashing.

disabling of F8 on the firmware card to the rising edge of the 7MHz clockpulse. This prevents switching of the F8 ENABLE' signal during the critical period before PHASE 0 falls. Switch bounce still occurs but any F8 ENABLE' bounce is synchronized to 7M rising.

Several hardware facts of life in the Apple are involved in the operation of this circuit. ROM enable/disable time is 120 nanoseconds maximum. The 7MHz clockpulse is the only available rising edge during PHASE 0 which can synchronize the F8 ENABLE' signal. During the critical period, 7M rises 140 nanoseconds before PHASE 0 falls and at the same time PHASE 0 falls. If you flip the F8 enable switch 140 nanoseconds before PHASE 0, there is plenty of time to disable one ROM, enable the other, and get the new data to the data input of the 6502 before PHASE 2 falls. If you flip the F8 enable switch right before PHASE 0 falls, there is no time, including propagation delays, for the newly enabled ROM to take control of the data bus before PHASE 2 falls.

The following is an installation procedure for modification shown in Figure 6.6b:

The following is an installation procedure for modification shown in Figure 6.6b:*

1. Use 30 gauge, solid, insulated wire to make connections on the firmware card. Solder the wires to the protruding ends of the pins of the IC sockets. Glue long wires to the board using a hot glue gun or other effective glue.
2. Desolder the F8 solder jumper.

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

3. Desolder and disconnect the end of R3 nearest the F8 solder jumper. R3 will be used to pull up the D-input to the synchronizing flip-flop. Bend the open lead of R3 away from the board so it will not accidentally touch any conductors. Remove all solder from the hole that was vacated when the R3 lead was removed. An insulated wire will be strung through this hole.
4. Connect the following points by soldering wires between them: [insert 6-3]

LS74-1 to LS74-14
LS74-5 to LS09-1
LS74-4 to pin 40 of the edge connector
LS74-3 to pin 36 of the edge connector
LS74-2 to the disconnected lead of R3 (no solder)

The wire between LS74-2 and the disconnected lead of R3 should be strung through the hole vacated by R3. Do not solder this connection yet because another wire will be connected here. When soldering a wire to an edge connector pin, carefully solder the wire to the upper part of the pin that does not actually make contact with the motherboard socket pins. These connections can be made more mechanically sound by drilling two small holes near the edge connector and leading the wires in one hole and out the other near the pin to which the wire will be soldered.

5. Connect a wire from one terminal of an on/off switch to the disconnected lead of R3 and the lead from LS74, pin 2. Solder this connection now. The connections to the switch should be via twin lead through an easily disconnected plug/jack connection.
6. Connect a wire between the other terminal of the on/off switch and pin 7 of the 74LS138. This connection can be made mechanically stronger by looping the switch wires through the small hole just above the D0 ROM socket.

Figure 6.6 is far from the last word in modifying the firmware card for versatile F8 selection. It's just the simplest. Figure 6.7 shows another more versatile scheme that does not require a second switch but does require a new chip. Figure 6.7 uses the second half of the LS74 as a completely independent F8 ROM selector flip-flop, with F8 selectable under program control. A 74LS30 is added to the firmware card to detect F8 ROM addressing, even when the main select flip-flop is in the motherboard enabled state. The Slot 0 software control of the

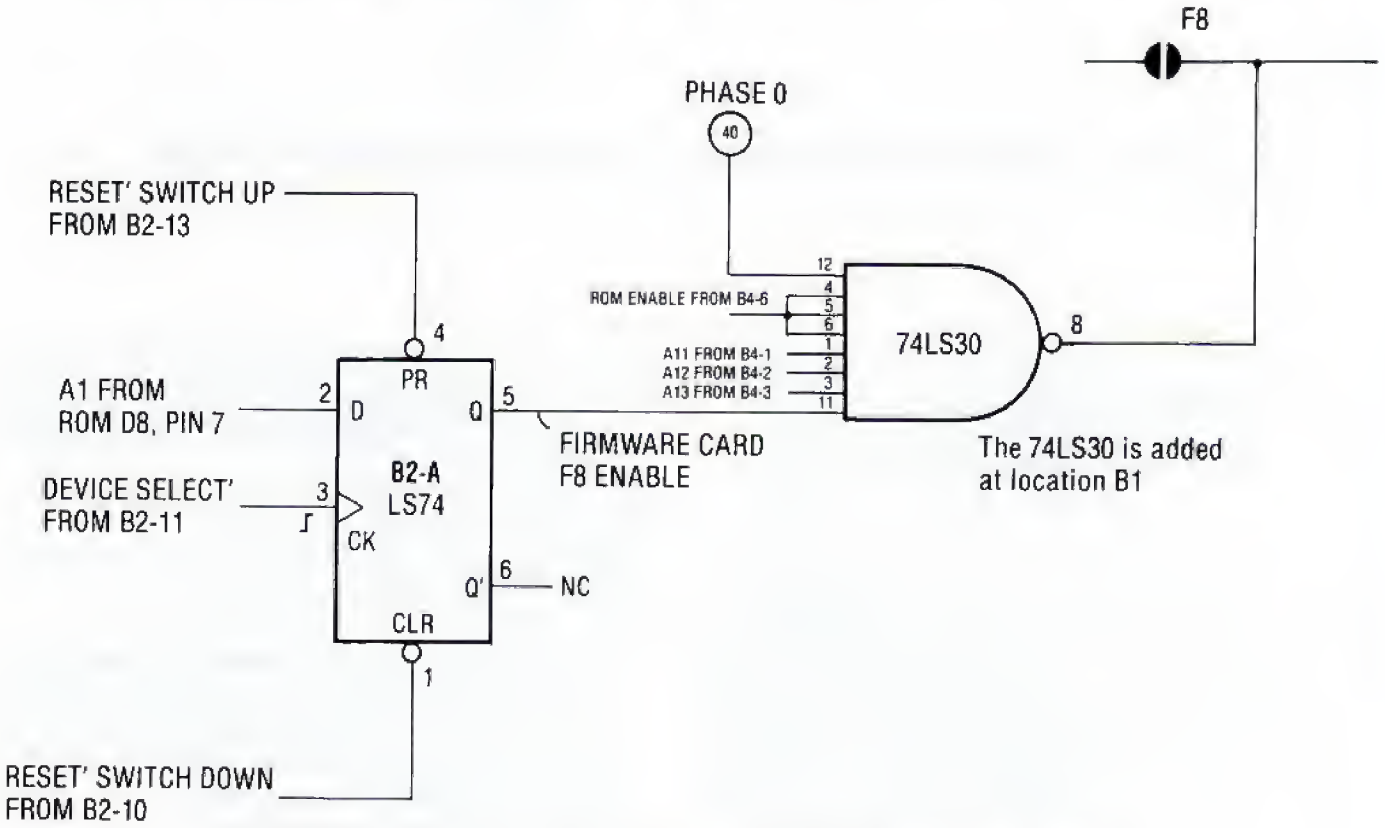


Figure 6.7 Circuit to Allow Independent Selection of the F8 ROM.

firmware card modified to this configuration is as follows:

	$D000-$F7FF	$F800-$FFFF
COMMAND	LOCATION	LOCATION
$C080	Firmware Card	Motherboard
$C081	Motherboard	Motherboard
$C082	Firmware Card	Firmware Card
$C083	Motherboard	Firmware Card

Since DOS only uses $C080 and $C081 for firmware card manipulation, it will always disable the firmware card F8 ROM. You can reverse this situation by connecting pin 11 of the LS30 to pin 6 of the LS74 instead of pin 5. Another feature of operation is that the switch in the back controls the F8 ROM flip-flop in the event of RESET, but it is not necessarily defeated by DOS in this function as it is in an unmodified firmware card. DOS normally controls the firmware card after an **Autostart** RESET. If the RESET selects an F8 ROM containing a firm RESET (like the Monitor ROM), program flow never vectors to DOS.

There is actually room for two new chips below the D0 ROM socket of the firmware card. The way to install a new chip is to mark and drill holes for the pins of a DIP socket.* Then attach the socket to the board using epoxy cement, carefully avoiding getting any epoxy on the contact areas of the pins. Wire can then be soldered to the protruding pins of the socket. Choose the location of the new socket so that you do not drill through conductive traces on either side of the board when drilling pin holes.

Here is one last modification suggestion for the firmware card which many Apple owners could use. This modification is to configure the card for EPROM in the F8 socket and ROM in the other five sockets. To make this modification, pull the F8 ROM from its socket and look at the board through the top of the socket. You can see conductive traces running across to the other sockets. Counting down from the top, the third and sixth traces from the top are the CS3 and CS2 chip select lines (see Figure 6.8). Cut these two traces with a razor knife and curl the traces back slightly to insure the conductive path is broken.* This isolates the chip selects on the F8 ROM from the chip selects on the other five ROMs. Now solder the 2716 jumper pads. This configures F8 for 2716 EPROMs. Finally, on any ROM socket except F8, solder wire jumpers between pins 24 and 18 and between pins 12 and 21, configuring the remaining five sockets for ROM. The F8 socket is now ready to accept your personal firmware in 2716 EPROM, but you can reconfigure it for ROM by removing the solder from the 2716 jumper pads.

The modifications presented here may give you other ideas on personalizing your firmware card. Remember that control of the F8 ROM in a Slot 0 card is the secret to maintaining control of your Apple.

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

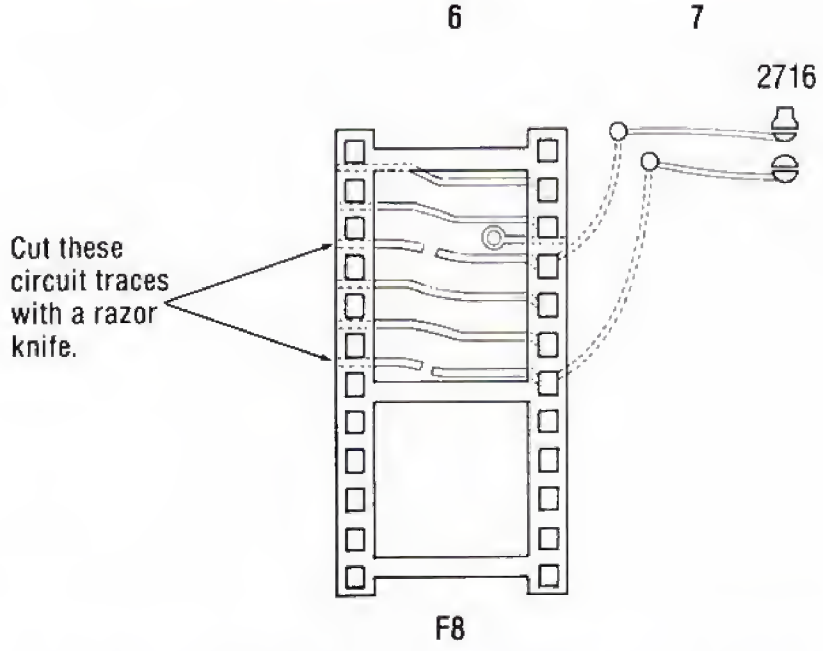
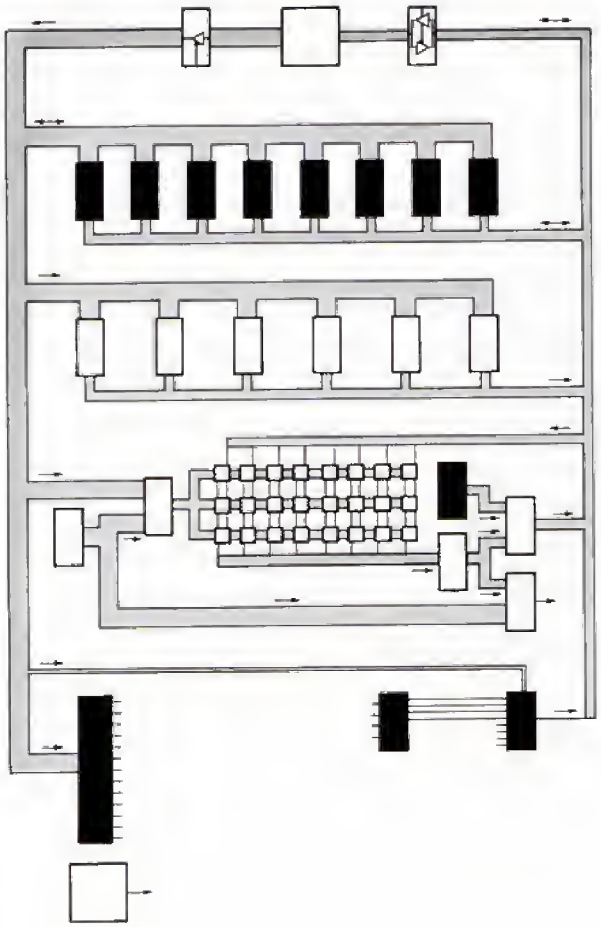


Figure 6.8 Method for Isolating the F8 Chip Selects from Other ROM Chip Selects.



chapter 7

Address Decoding and Input/Output

It bears repeating. MPU command of all devices in the Apple computer is via signals decoded from the address bus. All persons who program the Apple become aware of this sooner or later, and all users of the Apple can save themselves problems if they understand command by addressing. The concept of data transfer between the MPU and a memory location is very easily grasped, but it must be understood that the MPU also controls parts of the computer via the address bus with no related transfer of data on the data bus.

Now there is no "Control Address Bus" command in the 6502's repertoire. The 6502 reads from or writes to the data bus on every cycle. So what does the programmer do when he wants to toggle the speaker? He does a "LDA $C030" or a "CPX $C030" or a "WHO GIVES A DARN $C030" and ignores the meaningless data bus. This is why you can program the speaker with a statement like "SOUND = PEEK(-16336)." The object is not to "PEEK" into memory. The object is to get $C030 on to the address bus, commanding the speaker to toggle. Beneath the lid of the Apple, on every cycle, whether memory or a control function is being addressed, the state of the address bus is decoded to tell the rest of the Apple what the MPU is doing.

Address decoding in the Apple is the process of selecting one of 65,536 addressed locations from a 16-bit address. It is one thing to say you can represent 65,536 different states with a 16-bit number, and quite another thing to discern between all those states in the space of half a microsecond. As was mentioned in Chapter 2 (Bus Structure), much address decoding goes on inside RAM and ROM which we take for granted. In this chapter, we will look at how Apple addressing is divided up into major categories, such as ROM, I/O, and RAM. We will also look at the subdivisions of I/O control, an address area which is not conveniently carved up into thousands of addresses by single chips as RAM and ROM are.

In studying the details of I/O address decoding, it is natural to discuss all topics related to I/O. For this reason, address decoding and Apple Input/Output are covered in this single chapter. Figure 7.1 is a general block diagram showing the areas of discussion. It can be readily seen that the majority of address decoded signals are directly related to I/O functions. Video output is not a topic of this chapter but the sole subject of the next chapter.

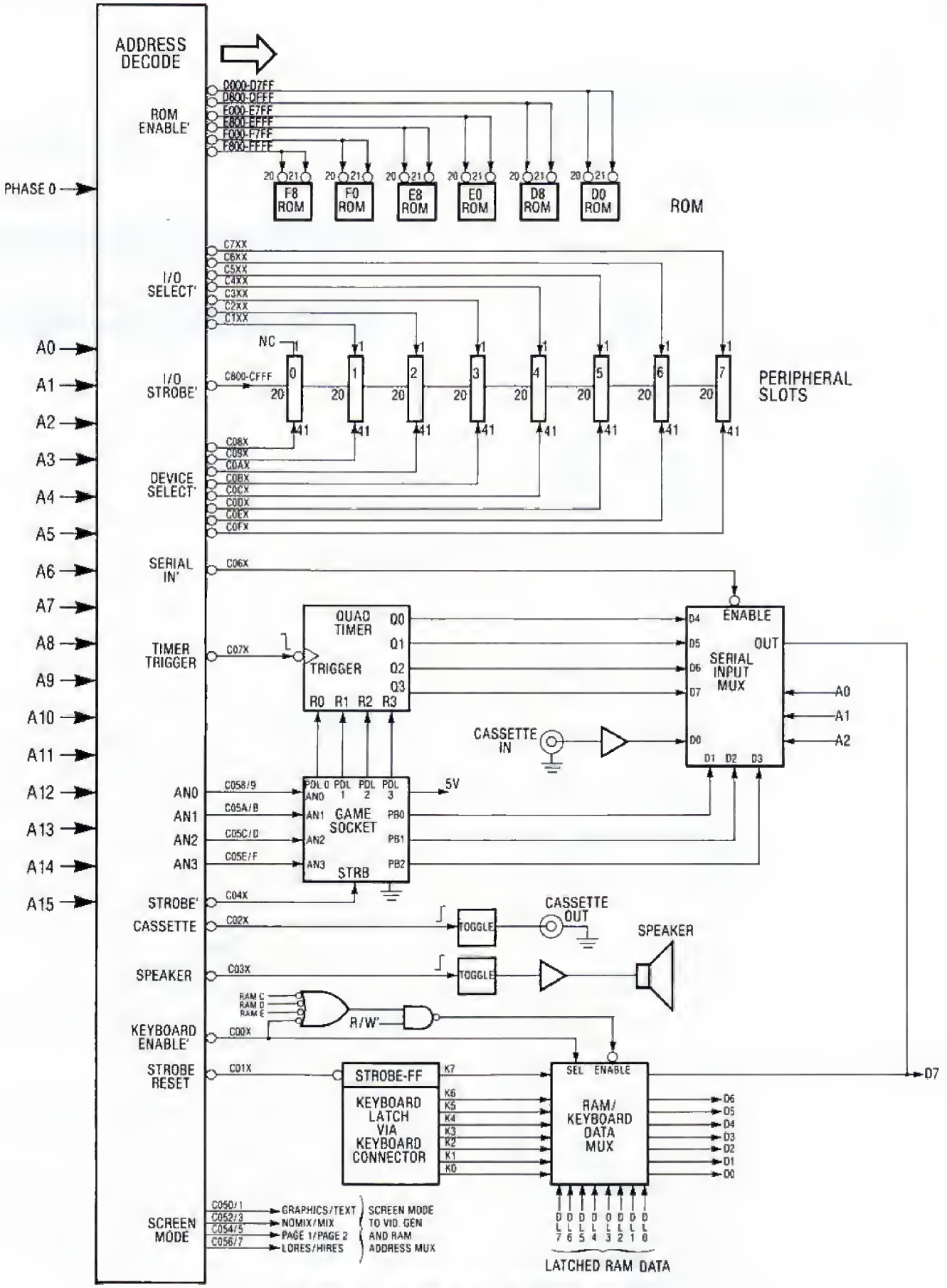


Figure 7.1 Schematic: Address Decode.

All of the decoded signals in Figure 7.1 are in the address range $C000 through $FFFF, in other words, everything except RAM. The decoding of RAM addresses is covered in the chapter on RAM. One way of looking at Figure 7.1 is as an illustration of everything that happens in the Apple when RAM is not being addressed. Here are two points about Figure 7.1. First, PHASE 0 and the address bus are shown as the only inputs. PHASE 1 is actually used at several active-low enabling inputs. The low state of PHASE 1 is PHASE 0. Second, the little circles on an output or input mean the signal is active low. As an example, when $C060 is on the address bus during PHASE 0, the C06X signal goes low. When C06X goes low at the serial input multiplexor, the multiplexor's output is enabled to D7 of the data bus.

THE ADDRESS DECODED SIGNALS

The address decoded signals occur during PHASE 0, or more accurately, the address decoded signals slightly lag PHASE 0. There is always some lag of a logic term behind the signal that controls it. In the Apple, PHASE 0 is a safe time to read the address bus, because the 6502 address is always valid. This way, the address decoded signals respond to 6502 generated addresses.

Some of the address decoded signals are **data bus management** gates. The function of these signals is to gate data to the data bus for reading by the MPU. These signals and their functions are:

$C00X	Gate keyboard data to data bus.
$C06X	Gate serial inputs to D7 of the data bus.
$D000-$FFFF	Gate ROM data to data bus.

Other signals control serial **toggle** outputs:

$C02X	Toggle cassette output.
$C03X	Toggle speaker.

The **screen modes** are controlled by off/on **soft switches**.

$C050/$C051	GRAPHICS/TEXT
$C052/$C053	NOMIX/MIX
$C054/$C055	PAGE1/PAGE2
$CO56/$CO57	LORES/HIRES

The **annunciator** outputs are off/on switches like the screen mode switches, but the annunciator signals are tied directly to output pins on the **game I/O connector**:

$C058/$C059	ANNUNCIATOR 0 OFF/ON
$C05A/$C05B	ANNUNCIATOR 1 OFF/ON
$C05C/$C05D	ANNUNCIATOR 2 OFF/ON
$C05E/$C05F	ANNUNCIATOR 3 OFF/ON

The C040 STROBE' is a signal that goes low during PHASE 0 while $C04X is on the address bus. It is tied directly to an output pin on the game I/O connector.

$C01X and $C07X perform special I/O functions as follows:

$C01X	Reset keyboard strobe flip-flop.
$C07X	Trigger paddle timers.

The last category of signals controls the **peripheral slots**:

$C080-$C0FF	DEVICE SELECTS'
$C100-$C7FF	I/O SELECTS'
$C800-$CFFF	I/O STROBE'

The functions of these address decoded signals were described at some length in previous chapters, and some of the functions are pretty obvious from Figure 7.1. However, a little elaboration concerning the signals won't hurt.

The DEVICE SELECTS', I/O SELECTS', AND I/O STROBE' are connected to the peripheral slots where they can control peripheral cards. Decoding these signals on the motherboard, rather than on the cards themselves, eliminates redundant hardware on the peripheral cards. It also makes it easy to design cards so they will operate in any slot.

Have a look at the game socket in Figure 7.1. The pins on this socket are available to external devices such as joysticks with pushbuttons. Additionally, the four annunciator signals and the C040 STROBE' are tied directly to pins of the game socket. The three **pushbutton** inputs (PB0, PB1, and PB2) are actually TTL inputs to the serial input multiplexor.

They could be used for all sorts of serial input but are normally used for pushbuttons. The four **paddle** inputs (PDL0, PDL1, PDL2, and PDL3) are tied to a **quad timer**. These inputs are not high/low binary voltage inputs like the pushbuttons. For that matter, the timers are not digital devices, although they do have TTL compatible outputs to the serial input multiplexor.

The way each timer works is this: first, the timers are triggered by a $C07X access. The outputs of the four timers then go high and each one stays high for a period of time determined by the position of the paddle connected to its input pin. The paddles are really variable resistors which vary the time constant of the input circuits to the timers. The timer outputs are connected to the serial input multiplexor so their high/low states can be read by a program. The programming method is to trigger the four timers, then poll the output of the pertinent timer in a loop while counting the program loops before the timer resets.

The eighth input to the serial input multiplexor is the **cassette input**. The cassette input jack of the Apple is connected to a high gain amplifier/shaper. The amplifier takes the small signal from the earphone jack of a cassette player and converts it to high and low voltages which can be read correctly by the serial input multiplexor. Of course, you can still adjust the cassette player volume too high or too low, but the amplifier gives you a very reasonable chance of selecting a volume which works.

The serial input multiplexor selects one of its eight inputs for output to D7 based on A0, A1, and A2 from the address bus. This is a good example of how things come in groups of eight or other powers of two in digital computers. Three address lines can be in eight different states, so 8 to 1 multiplexors exist for computer designers to use. This naturally leads to a computer with eight serial inputs. There is an operational oddity in the Apple that there are four paddle inputs but only three pushbutton inputs. It becomes logical in the light of hardware convenience. Eight serial inputs minus four timer inputs minus one cassette input leaves room for three pushbutton inputs.

Moving along, we come to some fairly obvious signals at the bottom of Figure 7.1. The **cassette output** simply toggles every time $C02X is accessed. The toggling device is just a flip-flop whose output is reduced to the level of a signal from a microphone then routed to the cassette output jack. The **speaker** output is also a toggled output, but the output of its toggling flip-flop is routed to the speaker through an amplifier. The amplifier is necessary because an LSTTL flip-flop is not capable of driving a 2 1/4" speaker. The **screen mode control** signals are not output signals, but they control the Apple's primary output, video. They do this by selecting screen memory in the RAM address multiplexor and by selecting the processing mode in the video generator.

The last thing to discuss in Figure 7.1 is the **keyboard** input. The 7-bit ASCII representation of whatever key was last pressed is stored in a big IC on the keyboard itself. This 7-bit latched keyboard data is connected through a short DIP jumper to the **RAM/keyboard data multiplexor**. Another output from the big keyboard chip is the keyboard STROBE, which goes high for about 20 microseconds when any key is pressed. The leading edge of the STROBE sets a flip-flop on the motherboard whose output is connected to the most significant bit input of the RAM/keyboard data multiplexor. When a read to $C00X is performed, the keyboard word is placed on the data bus for reading by the MPU. The MPU reads the 7-bit latched keyboard ASCII and the state of the keyboard strobe flip-flop.

The strobe flip-flop is reset on power up and by any access to $C01X. This enables the programmer to check for a keypress by resetting the strobe flip-flop, then polling $C000 until he finds the most significant bit set.

The I/O assignments of various addresses are important information for Apple programmers. Table 7.1 can serve as a reference for address decoded software commands. It can also serve as a memory map of the $C000-$FFFF address range in the Apple. Figure 7.1 and Table 7.1 go a long way in describing the particular programmable features of the Apple II computer.

ADDRESS DECODE HARDWARE

Let's look at the hardware which produces all these signals by monitoring the address bus. The circuitry is pictured in Figure 7.2. Would you believe it? It's done with just five LSTTL integrated circuits, four LS138s and an LS259. The LS138 is designed to do exactly this sort of task and make it look easy. What it does is take a 3-bit address input and bring one of eight lines low depending on the state of the address inputs. Eight lines, eight slots, two to the power of three: what a nice number of slots to have in a digital computer.

Here's the scheme for dividing up the $C000-$FFFF address range. First, divide the range into eight 2048 byte sections using the LS138 at F12 (see Figures 7.2 and 7.3). These eight sections are addressing for the six ROM chips, the "seventh ROM"

($C800-$CFFF), and the I/O Section ($C000-$C7FF). Then divide the I/O Section into eight 256 byte sections using the LS138 at H12. These eight sections are the seven I/O SELECT' ranges and the rest of I/O control ($C000-$C0FF). The $C000-$C0FF range is divided in two groups by A7 and A7' gating. The A7 group is divided into eight 16 byte sections by the LS138 at H2. These are the eight DEVICE SELECT' ranges. The A7' group is divided into eight 16 byte sections by the LS138 at F13. These are the motherboard control signals.

Most of these control signals are not broken down any further. For example, the cassette output line can be toggled by a reference to any of the 16 addresses in the $C02X range. The programming convention is to address these 16-bit ranges by their lowest address, $C030 for example. The $C05X range is broken down into eight off/on soft switches by the LS259 at F14. These are the annunciators and screen mode control signals. The $C06X signal performs the function of gating the serial input multiplexor (See Figure 7.5) to D7 of the data bus. One of

Table 7.1 Address Decoded Signals.

FUNCTION	HEX RANGE	DECIMAL RANGE	DECIMAL COMPLEMENT
RAM*	$0000 to $BFFF	00000 to 49151	-65536 to -16385
READ KEYBOARD	$C00X	49152 to 49167	-16384 to -16369
RESET KEYBOARD STROBE	$C01X	49168 to 49183	-16368 to -16353
TOGGLE CASSETTE OUTPUT	$C02X	49184 to 49199	-16352 to -16337
TOGGLE SPEAKER	$CO3X	49200 to 49215	-16336 to -16321
C040 STROBE	$C04X	49216 to 49231	-16320 to -16305
GRAPHICS	$C050	49232	-16304
TEXT	$CO51	49233	-16303
NOMIX	$CO52	49234	-16302
MIX	$CO53	49235	-16301
PAGE 1	$C054	49236	-16300
PAGE 2	$CO55	49237	-16299
LORES	$CO56	49238	-16298
HIRES	$C057	49239	-16297
AN0 LOW	$C058	49240	-16296
AN0 HIGH	$C059	49241	-16295
AN1 LOW	$CO5A	49242	-16294
AN1 HIGH	$C05B	49243	-16293
AN2 LOW	$CO5C	49244	-16292
AN2 HIGH	$C05D	49245	-16291
AN3 LOW	$CO5E	49246	-16290
AN3 HIGH	$CO5F	49247	-16289
READ CASSETTE INPUT	$C060/$C068	49248/49256	-16288/-16280
READ PUSHBUTTON 0	$CO61/$CO69	49249/49257	-16287/-16279
READ PUSHBUTTON 1	$CO62/$CO6A	49250/49258	-16286/-16278
READ PUSHBUTTON 2	$C063/$CO6B	49251/49259	-16285/-16277
READ TIMER 0	$C064/$CO6C	49252/49260	-16284/-16276
READ TIMER 1	$CO65/$C06D	49253/49261	-16283/-16275
READ TIMER 2	$C066/$C06E	49254/49262	-16282/-16274
READ TIMER 3	$C067/$CO6F	49255/49263	-16281/-16273
TIMER TRIGGER	$C07X	49264 to 49279	-16272 to -16257
DEVICE SELECTS'	$C080 to $C0FF	49280 to 49407	-16256 to -16129
I/O SELECTS'	$C100 to $C7FF	49408 to 51199	-16128 to -14337
I/O STROBE'	$C800 to $CFFF	51200 to 53247	-14336 to -12289
ROM ENABLES'	$D000 to $FFFF	53248 to 65535	-12288 to -00001

*RAMSELECT' is not decoded directly from the address bus. It is decoded from the screen mode during PHASE 1 and from the address bus during PHASE 0.

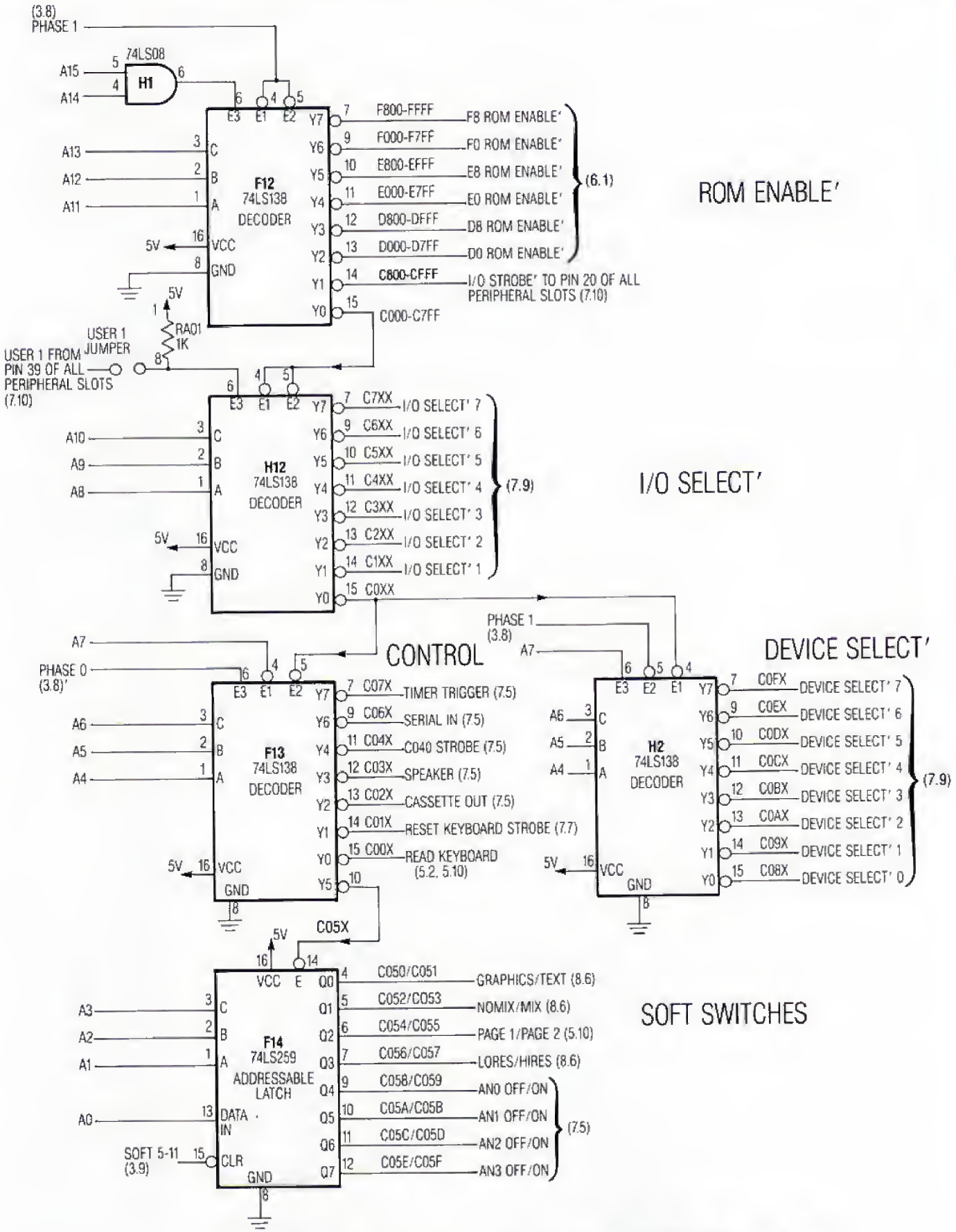
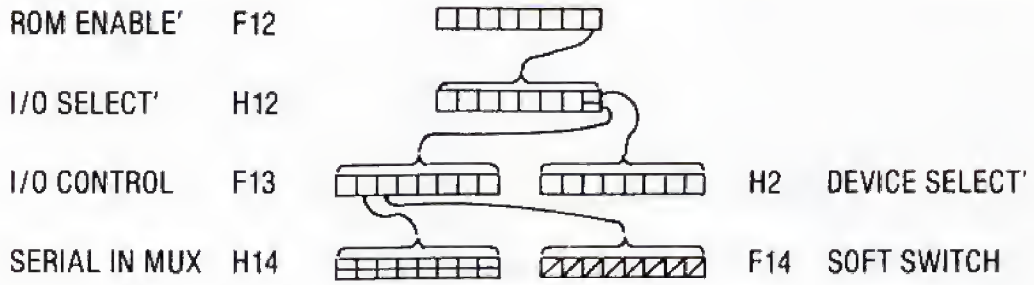


Figure 7.2 Schematic: Generation of Address Decoded Signals.



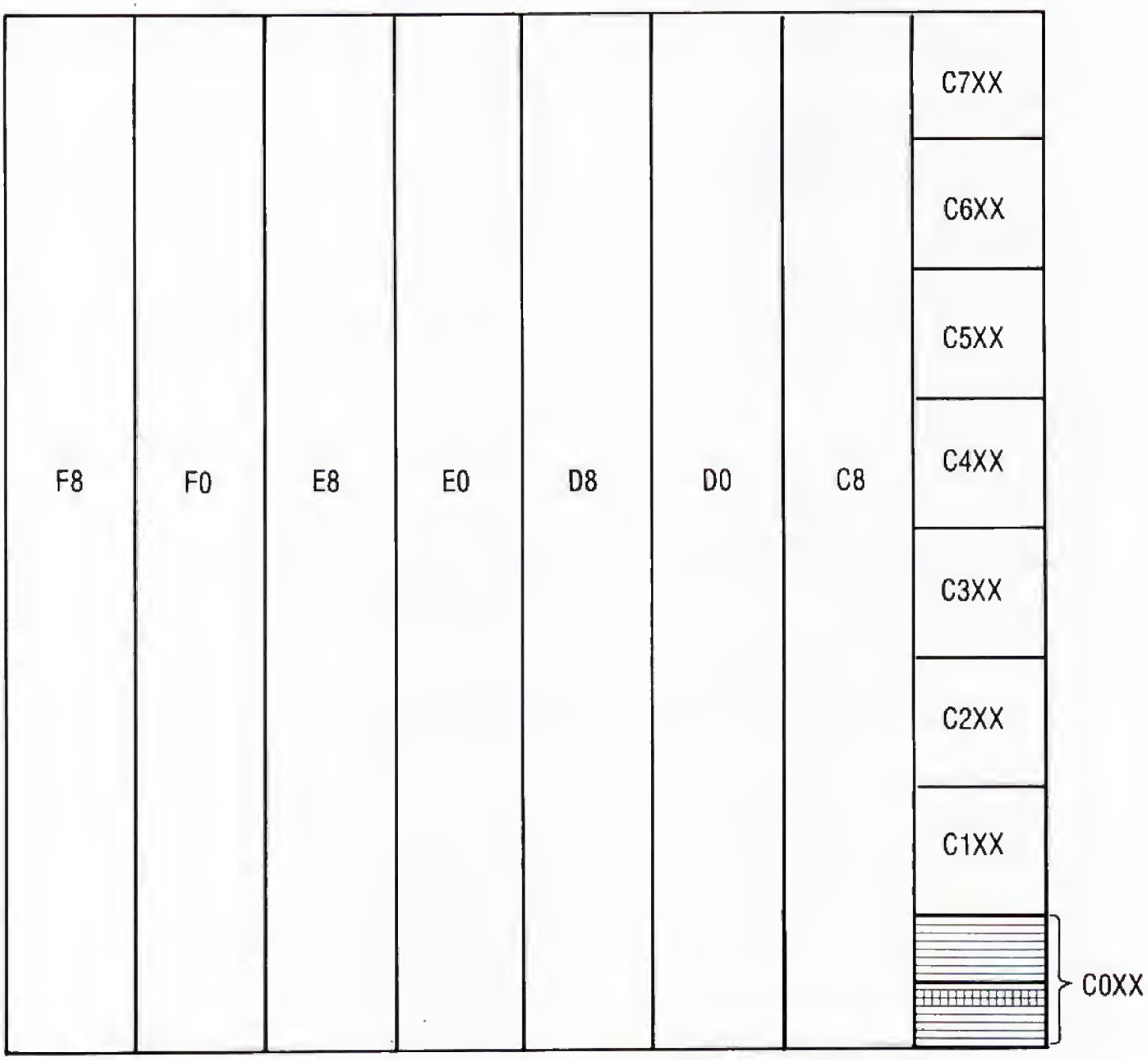


Figure 7.3 Address Decoding in the Apple is an Exercise in Division by Eight.

eight serial inputs are selected by this multiplexor based on A0, A1, and A2 from the address bus. This divides the $C06X range into two identical octets, $C060-$C067 and $C068-$C06F. The programming convention is to address the lower octet when reading a serial input. Therefore $C064 should be read when checking Timer 0, not $C06C.

There are several interesting subtleties of the connections in Figure 7.2:

1. F12, H12, F13/H2, and F14 are connected in a chain. F14 must be enabled before any of the other ICs in the chain can be enabled. Thus, a single AND gate connection to F12 detects the $C000-$FFFF range for all these ICs from A14 and A15.
2. PHASE 1 is connected to an active low enable input to F12, so PHASE 0 high is a prerequisite for any address decoded action above $BFFF. Then why is PHASE 0/PHASE 1 gating also connected to F13 and H2? It takes a long time to bring the F13 and H2 outputs high or low when the enabling term is propagated through three LS138s. Connecting PHASE 1 and PHASE 0 directly to F13 and H2 results in a quicker cutoff of outputs of these chips after PHASE 0 falls. Apparently, the Apple designers did not want the DEVICE SELECTS' and C06X to linger past PHASE 0.*
3. If the **USER1 jumper** is connected and a peripheral card brings pin 39 low, all address decoding between $C000 and $C7FF is inhibited. This 2K of addressing then becomes available for any sort of peripheral card response. The six ROM ENABLE' signals and I/O STROBE' are not affected by USER1.
4. R/W' gating is not connected to the address decode chain. Therefore command of Apple features can be via read or write access. However, if you write to a ROM address or a serial input address, the ROM or serial input multiplexor will compete with the MPU bidirectional data bus driver for control of the data bus. This is an undesirable situation which should be avoided.
5. The division of $C000-$FFFF addressing in the Apple is a very admirable example of design efficiency and creativity. It is a real work of art, considering how usable the Apple turned out to be. The decoding layout results in operational features of the Apple which we take for granted: there are eight peripheral slots; there are only seven I/O SELECT' signals; the I/O addressing range is the same size as a 2K ROM; six I/O control address ranges are 16 bytes wide, resulting in a wastage of 90 bytes; there is only one trigger for the four timers instead of a separate trigger for each timer, and there is no timer reset signal.

*Steve Wozniak gives Alan Baum much of the credit for the design of this area. I feel that they would have been better off to let DEVICE SELECT' linger a bit past PHASE 0, and I suspect that Wozniak and Baum wouldn't argue the point too vigorously.

I/O TIMING

I/O timing is the timing of the address decoded signals. All motherboard I/O and most peripheral slot I/O is controlled by these signals. An access to a DEVICE SELECT' address is sufficient to illustrate timing for all the address decoded signals because of the enabling chain which winds through the LS138s. For example, before any DEVICE SELECT' signal can drop low, the C0XX signal must drop low. The C0XX signal is identical in timing to the I/O SELECT' signals, so I/O SELECT' timing is also illustrated.

Figure 7.4 shows the read and write timing for an access to $C080. The read and write are identical except for data bus management. Data bus management in a write to $C080 is identical to that of RAM in a write cycle, because R/W' dropping always isolates the data bus for control by the MPU. The read cycle data bus management is different from that of a RAM read cycle, because RAM SELECT' rises during PHASE 0 when reading from an address above $C00F.

When accessing $C080, the C000-C7FF, C0XX, and C08X signals fall in succession after PHASE 0 rises. The LS138 chain is ready to be controlled by PHASE 0 since the MPU address has long since been valid on the address bus. The typical high to low propagation time of an active low enable signal through a LS138 is 21 nanoseconds, so the DEVICE SELECT' signal falls about 63 nanoseconds after PHASE 0 rises. This is true of any signal in the $C0XX range, $C06X for example. This means that the DEVICE SELECTS' are identical to the motherboard serial input data bus gate. The implication is that the DEVICE SELECT' is a valid data bus management signal for peripheral cards. In fact, the I/O SELECTS', the DEVICE SELECTS', and the I/O STROBE' all work very nicely for data bus management. This is in spite of the fact that they all rise before 6502 PHASE 2 falls. The reason they work is because of the slow bleed off of data from the floating data bus.

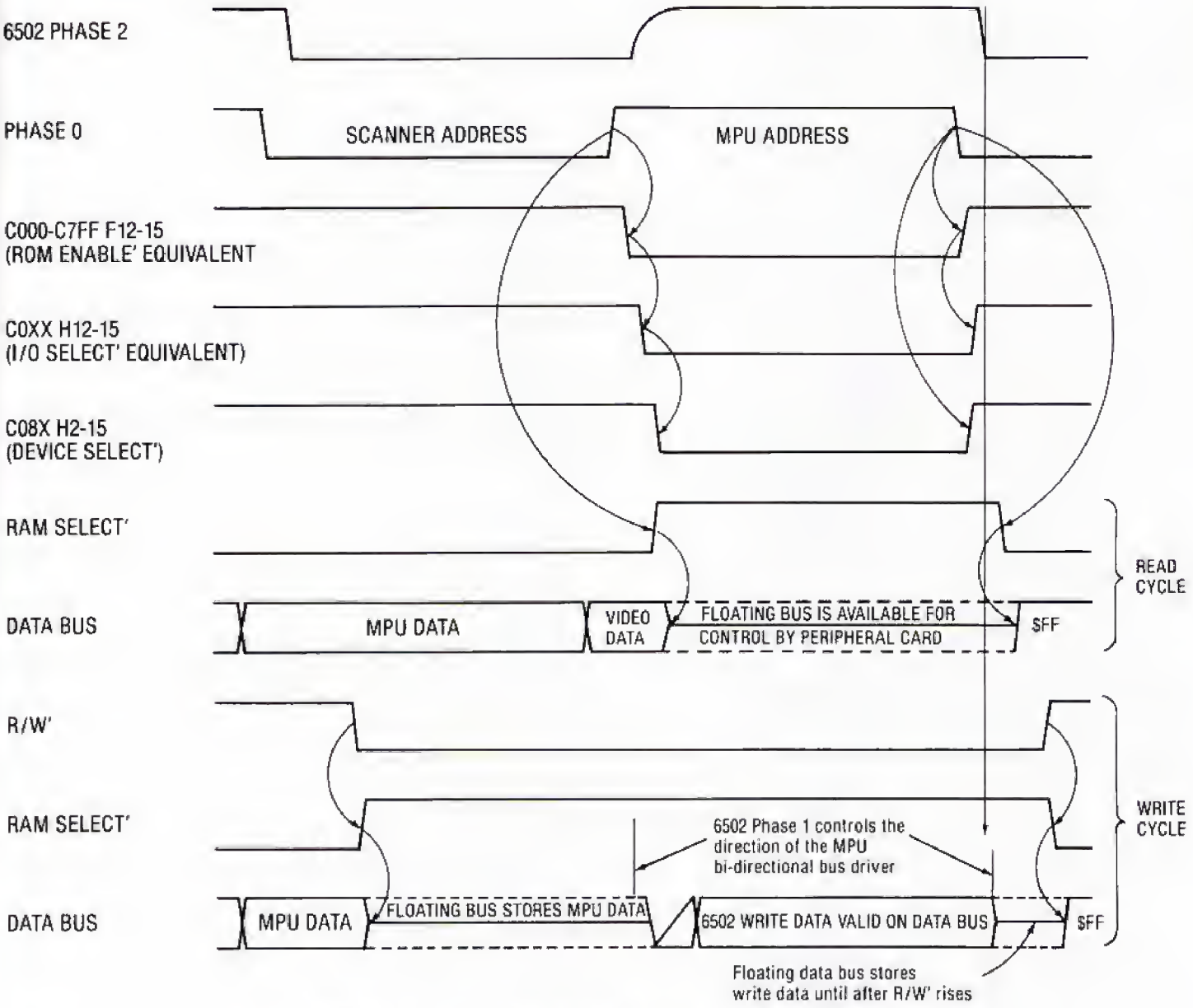


Figure 7.4 Timing Example: 6502 Access to $C080.

SERIAL I/O HARDWARE

Figure 7.5 is a schematic of Apple's serial I/O devices. The only connection to the outside world for most of these I/O devices is through the game I/O socket, and the inputs are connected to D7 of the data bus through the serial input multiplexor. Figure 7.5 shows a standard joystick or paddle set connected to the game I/O socket. This illustrates the fact that plugging a standard paddle set into the game I/O socket makes eight serial input or output ports inaccessible.

The standard paddle set has two pushbuttons, one for each paddle. These pushbuttons are designed to bring the input of an LS251 (the serial input multiplexor) high or low for transfer to D7 of the data bus. When a button is pushed, 5 volts is applied to the LS251 input. When the button is released, the LS251 input is pulled to ground through a 560 ohm resistor. The two pull-down resistors in a paddle set are installed in the plug which is inserted in the game I/O socket. They are well hidden to confuse hardware investigators who are trying to figure out how the Apple works. Just kidding. Apple, do you know

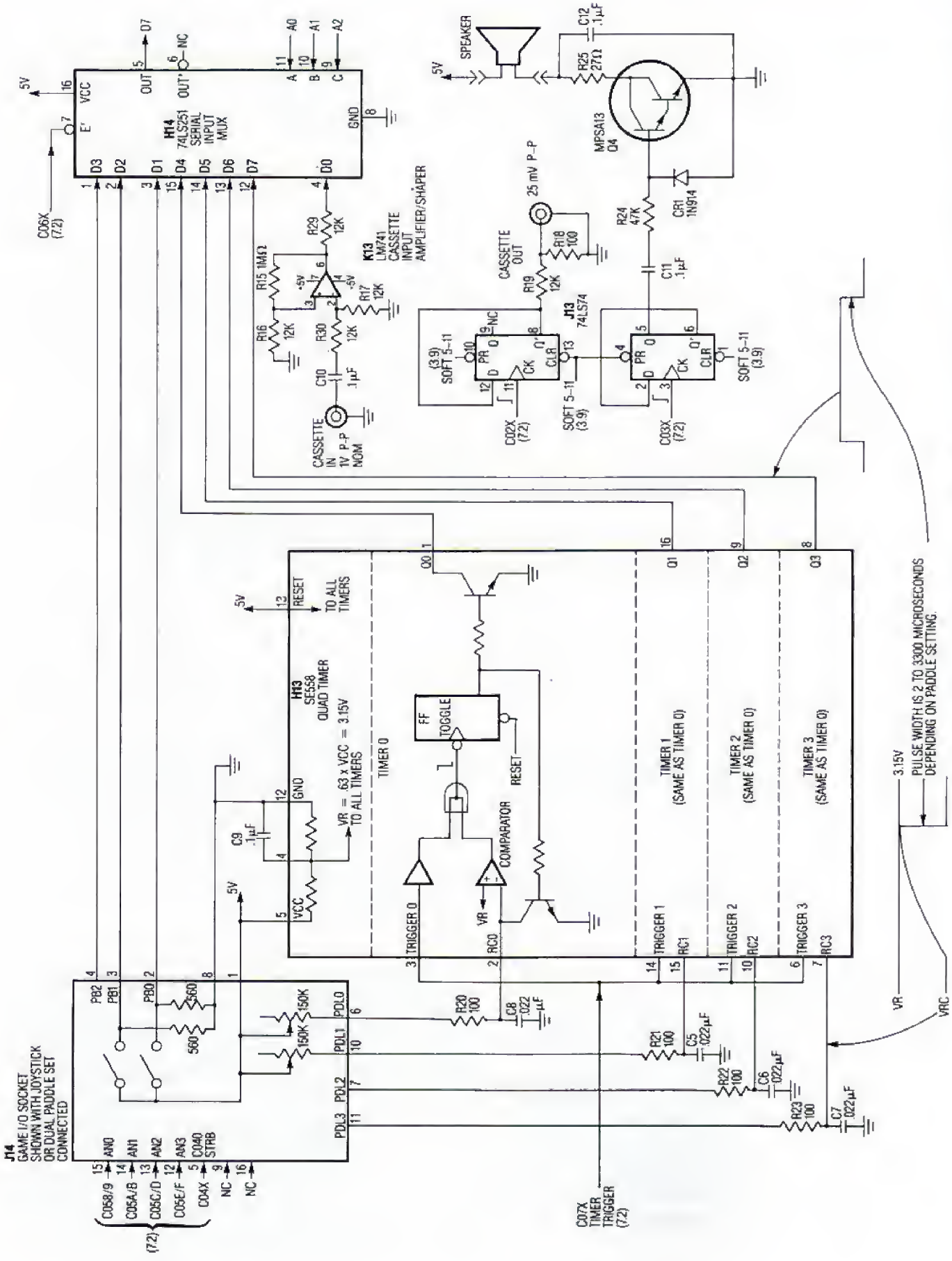


Figure 7.5 Schematic: Serial I/O Devices.

how much production cost you could have saved on 750,000 Apple IIs by mounting pull-down resistors on the motherboard instead of wiring them into that tiny little DIP plug?

The paddle knobs are attached to 150,000 ohm potentiometers (variable resistors). Each of these potentiometers is part of an R/C (Resistance/Capacitance) network which determines the pulse width of one of the outputs of a quad timer. They are designed so that the pulse width in microseconds will be equal to (RP + 100) X .022 where RP is the resistance of a paddle. Since the paddles are 150,000 ohm pots, the timer outputs can be varied from 2 (100 X .022) to 3300 (150100 X .022) microseconds. The four timer outputs are connected to the serial input multiplexor so their duration can be counted in a polling loop by the controlling program. Of course, a standard paddle set uses only two of the four available timers.

The NE558/SE558 quad timer has a RESET input which brings all four outputs low and inhibits triggering. This feature is not utilized in the Apple. Each timer has its own input trigger, but in the Apple, all four triggers are tied to the C07X line. To achieve the reset and individual trigger capabilities would have required extra address decoding circuitry and possibly would have decreased the performance/cost ratio of the Apple. As it is, the common trigger gives the capability of simultaneously reading two or more paddles, and the lack of reset merely requires waiting for timer reset at the beginning of timer polling routines. Unfortunately, neither or these realities is supported by Apple firmware.

The timer set up in the Apple is an astonishingly cheap way to achieve a four channel analog to digital input capability. Variable resistors exist whose resistance is proportional to light, heat, linear motion, rotational motion, chemical composition, and probably a lot of other pertinent things. This means that you can monitor all these qualities with an Apple computer via the quad timer. Of course, it takes a while to read these resistances, 22 microseconds per thousand ohms. On the other hand, how much is the temperature going to change in a thousand microseconds?

The one thing the timers are the least suited to do is the thing they are used quite often for, game controllers for real time arcade type games. A sophisticated HIRES arcade game on the Apple requires all of a programmer's skill if the result is fast paced realistic action. The 6502 in the Apple executes an instruction every three or four microseconds. It takes about 500 times as long to perform a typical timer polling routine. Needless to say, program speed is inversely proportional to the time spent polling the timers. Were the Apple designed today one would suspect that modern multichannel quick response analog to digital converters would be used for paddle input rather than timers.

A final interesting feature of the Apple timer connections are the open-collector outputs. What is interesting is that there are no pull-up resistors connected to them. One would have to walk all the way to Omaha to find another successful commercial design with open collectors driving LSTTL inputs without pull-up resistors. It works without pull-up resistors, but the conventional wisdom says to pull up open collector outputs to achieve satisfactory noise immunity. I wonder why they didn't bother in the case of the Apple timer inputs to the serial input multiplexor.

After the timers and pushbuttons, the remaining serial input is the cassette input. What we have here is an electronic circuit which answers the question, "when is an operational amplifier not an operational amplifier?" The answer is that an LM741 operational amplifier is not an operational amplifier when there is no degenerative feedback. It then becomes a saturated amplifier and, in the case of the Apple cassette input, a threshold detector/signal shaper. Mild apologies for all that electronic language, but this is an electronic circuit. What it does is this:

1. Blocking capacitor C10 removes any DC component of the input voltage and makes the pin 2 input to the LM741 vary above and below 0 volts.
2. The LM741 acts like a threshold detector. If the voltage at pin 2 rises above .15 volt (very approximately), the voltage at pin 6 will go as negative as an LS741 can bring it (about -4.3V) operating from a -5 volt negative supply. If the voltage at pin 2 lowers below -.15 volt (very approximately), the voltage at pin 6 will go as positive as an LM741 can bring it (about +4.3V) operating from a +5 volt positive supply. Thus, as long as the cassette input exceeds the input threshold, the voltage at pin 6 will be an 8.6 volt p-p squared signal that switches high or low as the input crosses the low threshold or high threshold.
3. Pin 6 of the LM741 is connected to the serial input multiplexor through a 12 thousand ohm resistor. While K13-6 is at +4.3 volts, 4.1 volts or so is felt at H14-4. While K13-6 is at -4.3 volts, the negative input clamp of the LS251 holds the

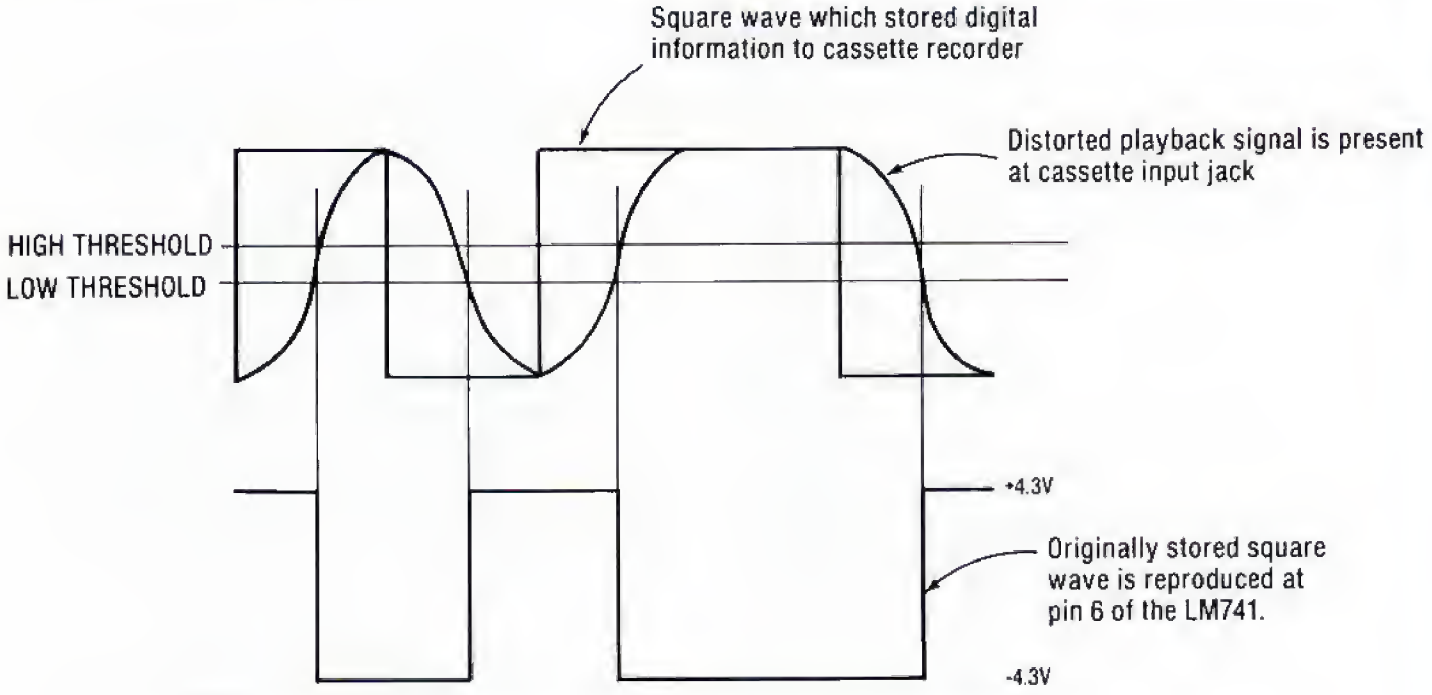


Figure 7.6 Cassette Input Wave Shaping.

voltage at pin 4 very close to zero volts, and the 12 thousand ohm resistor limits input current to about .36 milliamps.

The neat thing about the cassette input circuit is that even when your cheap tape recorder distorts a digital square wave, a square wave of correct pulse width is presented to the input of the serial input multiplexor. Figure 7.6 shows what happens when a $30 tape player makes a sine wave out of your square wave. The LM741 still switches at points separated in time by multiples of the period of the program loop that stored the information to cassette. Of course, there are limits to the distortion which an Apple can work with.

The *Apple II Reference Manual* states that the nominal voltage required at the cassette input is 1 volt peak to peak. This is a voltage one would reasonably expect to find at the earphone output of a cassette recorder. The cassette output of the Apple is a much smaller voltage, comparable in amplitude to the signal out of a microphone. This voltage is the output of a 74LS74 flip-flop reduced by a factor of 121. The flip-flop toggles once every access to $C02X. Its output swings back and forth between 0 and 3 volts while it is being toggled, so the cassette output jack swings between 0 and .025 volts (3/121 = .025).

The audio output is also controlled via a toggling flip-flop. The output of the audio flip-flop controls a simple amplifier which drives the Apple's speaker. Current flow through the speaker is in only one direction so the speaker action is tension/relax rather than push/pull. Alternate references to $C03X tension the speaker diaphragm then relax it, but the program cannot determine whether a $C03X reference causes tension or relaxation. This dampens the possibilities of complex program control of the speaker tension. In any case, an audio cycle consists of a tension half cycle and a relaxation half cycle, so two $C03X references are required per audio cycle. For example, to program a 1000 Hz tone, you reference $C03X 2000 times a second.

THE APPLE II KEYBOARD

Considering the stature of the Apple II computer, its keyboard has rather modest features. In particular, the ability to enter only upper case from the keyboard has always been a serious drawback to the Apple in sophisticated text handling applications. It was a disappointment to many users for years that Apple did not upgrade the overall text handling capability of the Apple II via improvements to the

keyboard, screen display, and firmware. It is a bit of a nuisance to resort to peripheral cards and modifications just to make a computer acceptable for word processing.*

Actually, Apple did significantly upgrade their keyboard (circa 1979). They cured an old headache by requiring the CTRL key to be pressed before the RESET key can function. They also gave the keyboard an upper/lower case capability, but they didn't tell owners how to take advantage of it or even that it was there, This upgraded keyboard came out at approximately the same time as the Apple II Plus, so we will be referring to it as the II Plus keyboard. Apple has never published the II Plus keyboard schematic for the general public. The schematic in the *Apple II Reference Manual* is the schematic of the old keyboard.

Figure 7.7 is a schematic diagram of the old keyboard. It is based on the National Semiconductor MM5740 **keyboard encoder ROM**, which is no longer manufactured. The MM5740 supports up to 90 keys in a 9 x 10 matrix, and it works by scanning through its X-drivers while checking its Y-sensors. The code that is generated from each keypress is specified by the buyer of the encoder ROM (Apple Computer Inc.). The code is latched every time a key in the matrix is pressed and it is gated to nine output pins through tri-state outputs. The fact that the outputs are latched means that you can press a key and its 7-bit ASCII can be checked anytime before another key is pressed. In the Apple, two of the outputs are not connected (B8 and B9), and the tri-state outputs are always enabled by a ground at pin 15. The B1-B7 outputs, which are active low, are inverted and routed through the 16-line keyboard jumper to the tri-state RAM/keyboard data multiplexor.

The MM5740 is capable of producing different code at outputs B5, B6, B7, and B8 for ALONE, CTRL, SHIFT, and CTRL-SHIFT input modes. This means that the Apple could have come with upper and lower case alphabetic character input from the keyboard at no extra manufacturing cost. Rather than doing this, they specified that the encoder output code for all alphabetic keys except M, N, and P would be the same whether SHIFT was pressed or not. The MM5740 also has a SHIFT LOCK input which is not connected in the Apple. No doubt about it, they really didn't want to fool with lower case alphabetics.

I looked at the states of unconnected B8 and B9 with an oscilloscope just to see what was programmed in there. B8 is a parity bit for B1-B7. If an even number of lines B1 through B7 are low, B8 goes low. If an odd number of lines B1 through B7 are low, B8 goes high. This means you could do a parity check on the keyboard input by connecting up a little hardware, but it would serve no useful purpose. I could make no sense of B9. Outputs B1-B4 and B9 are not affected by the SHIFT or CONTROL inputs to an MM5740. In the Apple, B9 is programmed to drop low if RETURN, N, left arrow, period, dash, colon, zero, or space is pressed. It goes high on all other matrix keypresses.

The keys on the Apple keyboard are spring loaded, normally open switches with plastic covers which can be pried off if a switch fails and needs to be replaced. The keys are labeled 1 through 55 with no #27 or #41 key. The power-on indicator is labeled #54 even though it is not a switch. Forty-seven of the keys are in the MM5740 input matrix. All keys on the matrix perform no function except to produce code which must be interpreted by a program. The RESET key is connected directly to the system RESET' line. The left SHIFT, right SHIFT, and CONTROL keys are addressing inputs to the MM5740 which select among four possible sets of output code. The SHIFT line is also active at power up although I am baffled as to why the encoder should be up-SHIFTED at power up. The REPT key enables a 10 Hz oscillator which feeds the MM5740 REPEAT input. When REPT and a matrix key are pressed simultaneously, the 5740 puts out a STROBE 10 times a second to simulate 10 keypresses per second. This REPEAT rate can be increased by reducing the value of R3 or vice versa.

The MM5740 requires an input clock between 10 KHz and 200 KHz. The Apple supplies this from a ring oscillator which runs at 50 KHz in the author's Apple. The frequency of this type of oscillator is very unpredictable, but it is perfectly adequate for this purpose.

Pin 1 on the MM5740 is a keybounce mask input. The MM5740 data sheet shows that the encoder will mask keybounces for 8 milliseconds if pin 17 is connected to **ground** through a .001 microfarad capacitor. In the Apple, pin 17 is connected to **+5 Volts** through a .001 microfarad capacitor. Apparently this works. If you often get two letters on the screen from one keypress, you can experiment with increasing the value of C4, and you might try connecting C4 to ground instead of +5 volts.

*Most readers are probably aware that the text handling capability of the Apple IIe is greatly superior to that of the Apple II. Please remember that these discussions apply to the Apple II only as it was before the introduction of the Apple IIe in early 1983.

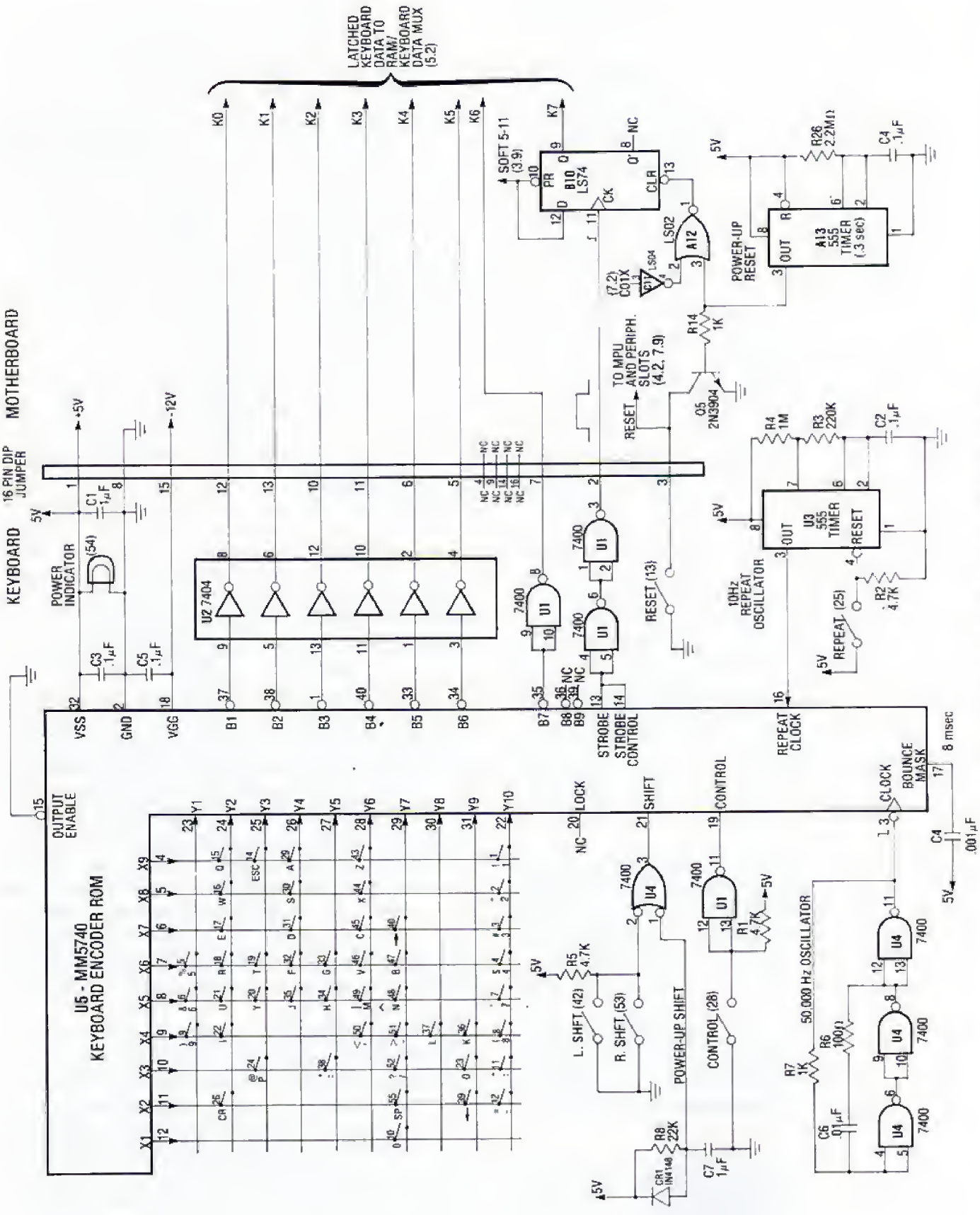


Figure 7.7 Schematic: The Apple II Keyboard.

The encoder outputs a positive STROBE one time when a matrix key is pressed, and 10 times per second if a matrix key is held simultaneously with the REPT key. When pin 13 is tied to pin 14, as it is in the Apple, the STROBE stays high for one period of the input clock. This is 20 microseconds with a 50 KHz clock. The *Apple II Reference Manual* says the STROBE lasts a maximum of 10 microseconds. This is probably a safe number they arrived at, figuring the ring oscillator would never get above 100 KHz. The STROBE output of the MM5740 is routed through two inverting stages to the motherboard where its rising edge sets the keyboard strobe flip-flop. The strobe flip-flop is reset under program control by a $C01X access.

Figure 7.7 also shows the Apple's **power-up reset** generator which is installed on all Revision 1 or later motherboards. This is a 555 timer which outputs a .3 second pulse when the Apple is turned on. This pulse resets the keyboard strobe flip-flop and also forces the RESET' line low. The 6502 reset sequence actually begins when RESET' rises. The .3 second RESET pulse at turn on gives the Apple power supply voltages time to stabilize before the 6502 and the rest of the Apple are off and running. Next time you turn an Apple on, listen carefully. There is a perceptible .3 second delay before the BELL sounds and the disk starts running. If you have a Revision 0 board and a Disk II controller, the controller will generate a .1 second power-up reset. If you have a Revision 0 board and no Disk II controller, your unmodified Apple will not automatically reset at power up.

This, then, was the Apple keyboard before the Apple II Plus modifications: 52 keys, uppercase alphabetics only, no numeric keypad, no SHIFT LOCK switch, no user programmable function keys, no automatic repeat, no dedicated cursor move keys, no dedicated screen mode selection keys, teletype style placement of symbolic characters, latched output flagged by a program resettable strobe flip-flop whose state is read in the MSB of the keyboard input word. After the II Plus? Pretty much the same operationally but quite different in hardware mechanization with some unadvertised attainable capabilities. Take a look.

The Apple II Plus Keyboard

Figure 7.8 is a schematic diagram of the Apple II Plus keyboard. I drew the schematic after studying an actual keyboard. The II Plus keyboard has an underlying similarity to the old keyboard with some obvious changes. Most obviously it uses a different encoder ROM, a General Instrument AY-5-3600 instead of the MM5740. This encoder is very similar to the MM5740 with a 9 x 10 switch matrix, nine low level outputs and, +5V/-12V power supply requirements. As an option, Apple could have specified that the encoder would have an output enable pin like the MM5740, but they chose not to since the Apple design doesn't utilize this feature.

A major difference in the mechanical package in the II Plus keyboard is that all electronics are removed to a small card, physically separate from the main card. The big card contains only key switches and the power indicator. The small card, called the **encoder board**, is connected to the big card via a 25-pin connector with exposed pins. The pins of this connector, PLUG 1 on the encoder board, are shown in Figure 7.8 as numbers in circles.

The keyboard/motherboard jumper is connected to a 16-pin DIP socket, JACK 1, on the encoder board. The pin assignments are identical to the keyboard connector on the old keyboard. It may not be possible, but try not to confuse JACK 1 with PLUG 1 on the small board. They are not the same, but are considered part of two separate PLUG 1/JACK 1 combinations following some perverse engineering logic.

A third connector on the encoder board, JACK 2, is not actually there. There are mounting holes for JACK 2, and they are fully wired so you can solder a jack in and plug a numeric keypad into it. The four matrix scanners X5 through X8 are used only by the numeric keypad, which is shown installed in Figure 7.8. The pins of JACK 2 are represented by little squares with numbers in them in Figure 7.8. The dedication of four X-drivers to a numeric keypad is possible because the keyboard has only 47 matrix switches and can operate with just a 5 by 10 switch matrix. Apple has never sold a keypad that supports JACK 2, but at least one has been built by an outside source. Yet a 16-key numeric keypad for the Apple II Plus would be very cheap to manufacture, because the capability is built in. Also, it would be easy to build your own 16-key keypad from a surplus calculator entry panel. A 24-key keypad requires electronic circuitry to decode a 4 x 6 matrix and switch between the keypad and the keyboard at the keyboard connector.

The SHIFT keys and CTRL key are connected up very much as in the old keyboard, except there is no forced SHIFT-up at power on. The RESET key is different though. There is a slide switch, S1, by which the owner can select CTRL required or CTRL not required for RESET. Believe it or not, I knew an

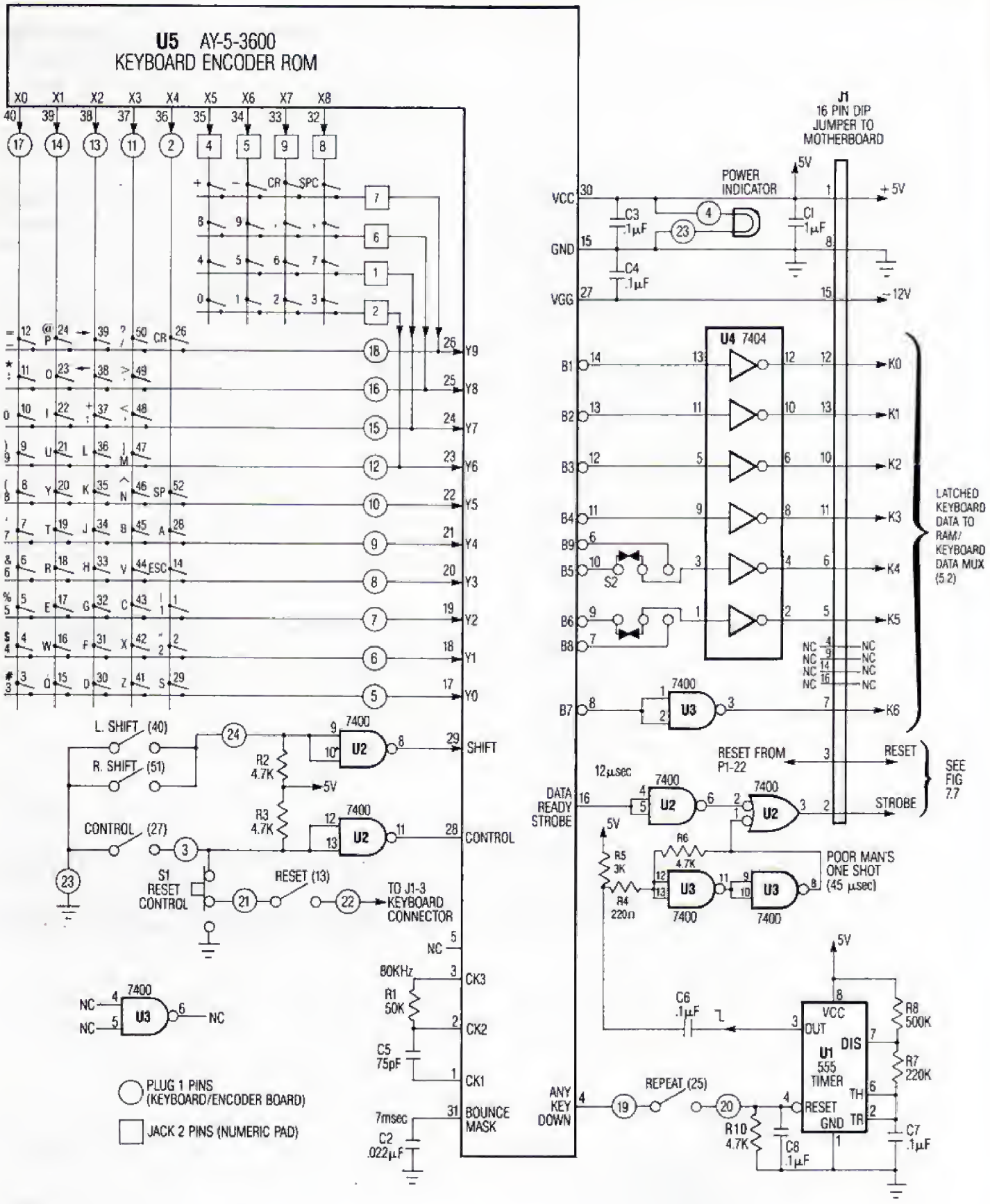


Figure 7.8 Schematic: The Apple II Plus Keyboard and Encoder Board.

owner in Japan who left this switch in the CTRL not required position. However, that person was quirky by nature.

The clock circuitry is mostly contained in the AY-5-3600. External components R1 and C5 set the operating frequency at approximately 80 KHz. Similarly C2 sets the keybounce mask period at 7 milliseconds.

The STROBE output works as it does in the MM5740 with a high level, one clock strobe output (12 microseconds) any time a matrix key is pressed. The REPEAT operation is different though. In the MM5740 a REPEAT input clock triggers the STROBE output if a matrix key is being held down. The AY-5-3600 is less sophisticated in this regard. It has an ANY KEY DOWN output which goes high when any key is held down. This is used in the Apple to enable a 15 Hz REPEAT oscillator when REPT and a matrix key are held. The REPEAT oscillator output is capacitively coupled to a circuit which generates a strobe of very roughly 45 microseconds when the REPEAT clock drops.

Capacitor C8 in the REPEAT oscillator circuit was apparently an afterthought. It is wired between pins 20 and 23 of PLUG 1 rather than being installed on the encoder board. It looks as though C8 is there to cause a short delay after REPT is held before the REPEAT oscillator is enabled.

The B1-B7 outputs of the II Plus encoder are routed to the keyboard/motherboard connector through inverters, just as on the old keyboard. There is a very important possibility incorporated, though. There are wired holes for mounting a slide switch which can select B9 and B8 instead of B5 and B6 as the K4 and K5 outputs. The B9 and B6 outputs are programmed for alphabetic shifting. This means that you can cut the two bow tie solder jumpers, install a slide switch, and have the option of enabling or disabling alphabetic shifting of the keyboard input. Table 7.2 shows the normal ASCII from the Apple keyboard and the ASCII from the II plus keyboard with alphabetic shifting enabled.

PERIPHERAL SLOT CONNECTIONS

It is hard to know where to start talking about peripheral slot I/O. You can do so much from the slots. They are as versatile as modern microcomputer architecture with full connection to the address bus and data bus. It's like someone designed a really neat computer but on the blueprints drew eight empty squares with the message, "user, please fill in the blanks." One never knows what lurks beneath the lid of an innocent looking Apple.

The capabilities of the peripheral slots seem more clear when you look at the connected signals in groups. Figure 7.9 illustrates the peripheral slot connections and groups the signals functionally.

Table 7.2 Relation of ASCII to Keypress. (1 of 2)

KEY	APPLE II SWITCH#	APPLE II Plus SWITCH#	NORMAL				APPLE II Plus LOWER CASE ENABLED			
			ALONE	CONTROL	SHIFT	BOTH	ALONE	CONTROL	SHIFT	BOTH
space	55	52	$A0	$A0	$A0	$A0	$A0	$A0	$A0	$A0
, <	50	48	$AC	$AC	$BC	$BC	$AC	$AC	$BC	$BC
- =	12	12	$AD	$AD	$BD	$BD	$AD	$AD	$BD	$BD
. >	51	49	$AE	$AE	$BE	$BE	$AE	$AE	$BE	$BE
/ ?	52	50	$AF	$AF	$BF	$BF	$AF	$AF	$BF	$BF
0	10	10	$B0	$B0	$B0	$B0	$B0	$B0	$B0	$B0
1 !	1	1	$B1	$B1	$A1	$A1	$B1	$B1	$A1	$A1
2 "	2	2	$B2	$B2	$A2	$A2	$B2	$B2	$A2	$A2
3 #	3	3	$B3	$B3	$A3	$A3	$B3	$B3	$A3	$A3
4 $	4	4	$B4	$B4	$A4	$A4	$B4	$B4	$A4	$A4
5 %	5	5	$B5	$B5	$A5	$A5	$B5	$B5	$A5	$A5
6 &	6	6	$B6	$B6	$A6	$A6	$B6	$B6	$A6	$A6
7 '	7	7	$B7	$B7	$A7	$A7	$B7	$B7	$A7	$A7
8 (	8	8	$B8	$B8	$A8	$A8	$B8	$B8	$A8	$A8
9)	9	9	$B9	$B9	$A9	$A9	$B9	$B9	$A9	$A9
: *	11	11	$BA	$BA	$AA	$AA	$BA	$BA	$AA	$AA
; +	38	37	$BB	$BB	$AB	$AB	$BB	$BB	$AB	$AB
A	29	28	$C1	$81	$C1	$81	$E1	$81	$C1	$81
B	47	45	$C2	$82	$C2	$82	$E2	$82	$C2	$82
C	45	43	$C3	$83	$C3	$83	$E3	$83	$C3	$83

Table 7.2 Relation of ASCII to Keypress. (2 of 2)

KEY	APPLE II SWITCH#	APPLE II Plus SWITCH#	NORMAL ALONE	NORMAL CONTROL	NORMAL SHIFT	NORMAL BOTH	APPLE II Plus LOWER CASE ENABLED ALONE	APPLE II Plus LOWER CASE ENABLED CONTROL	APPLE II Plus LOWER CASE ENABLED SHIFT	APPLE II Plus LOWER CASE ENABLED BOTH
D	31	30	$C4	$84	$C4	$84	$E4	$84	$C4	$84
E	17	17	$C5	$85	$C5	$85	$E5	$85	$C5	$85
F	32	31	$C6	$86	$C6	$86	$E6	$86	$C6	$86
G	33	32	$C7	$87	$C7	$87	$E7	$87	$C7	$87
H	34	33	$C8	$88	$C8	$88	$E8	$88	$C8	$88
l arrow	39	38	$88	$88	$88	$88	$88	$88	$88	$88
I	22	22	$C9	$89	$C9	$89	$E9	$89	$C9	$89
J	35	34	$CA	$8A	$CA	$8A	$EA	$8A	$CA	$8A
K	36	35	$CB	$8B	$CB	$8B	$EB	$8B	$CB	$8B
L	37	36	$CC	$8C	$CC	$8C	$EC	$8C	$CC	$8C
M]	49	47	$CD	$8D	$DD	$9D	$ED	$8D	$CD	$9D
RETURN	26	26	$8D	$8D	$8D	$8D	$8D	$8D	$8D	$8D
N ^	48	46	$CE	$8E	$DE	$9E	$EE	$8E	$CE	$9E
O	23	23	$CF	$8F	$CF	$8F	$EF	$8F	$CF	$8F
P @	24	24	$D0	$90	$C0	$80	$F0	$90	$D0	$80
Q	15	15	$D1	$91	$D1	$91	$F1	$91	$D1	$91
R	18	18	$D2	$92	$D2	$92	$F2	$92	$D2	$92
S	30	29	$D3	$93	$D3	$93	$F3	$93	$D3	$93
T	19	19	$D4	$94	$D4	$94	$F4	$94	$D4	$94
U	21	21	$D5	$95	$D5	$95	$F5	$95	$D5	$95
r arrow	40	39	$95	$95	$95	$95	$95	$95	$95	$95
V	46	44	$D6	$96	$D6	$96	$F6	$96	$D6	$96
W	16	16	$D7	$97	$D7	$97	$F7	$97	$D7	$97
X	44	42	$D8	$98	$D8	$98	$F8	$98	$D8	$98
Y	20	20	$D9	$99	$D9	$99	$F9	$99	$D9	$99
Z	43	41	$DA	$9A	$DA	$9A	$FA	$9A	$DA	$9A
ESC	14	14	$9B	$9B	$9B	$9B	$9B	$9B	$9B	$9B
RESET	13	13								
REPT	25	25								
CTRL	28	27								
SHIFT,l	42	40								
SHIFT,r	53	51								
II Plus KEYPAD										
0			$B0	$B0	$B0	$B0	$B0	$B0	$B0	$B0
1			$B1	$B1	$B1	$B1	$B1	$B1	$B1	$B1
2			$B2	$B2	$B2	$B2	$B2	$B2	$B2	$B2
3			$B3	$B3	$B3	$B3	$B3	$B3	$B3	$B3
4			$B4	$B4	$B4	$B4	$B4	$B4	$B4	$B4
5			$B5	$B5	$B5	$B5	$B5	$B5	$B5	$B5
6			$B6	$B6	$B6	$B6	$B6	$B6	$B6	$B6
7			$B7	$B7	$B7	$B7	$B7	$B7	$B7	$B7
8			$B8	$B8	$B8	$B8	$B8	$B8	$B8	$B8
9			$B9	$B9	$B9	$B9	$B9	$B9	$B9	$B9
space			$A0	$A0	$AO	$A0	$A0	$A0	$A0	$A0
+			$AB	$AB	$AB	$AB	$AB	$AB	$AB	$AB
,			$AC	$AC	$AC	$AC	$AC	$AC	$AC	$AC
-			$AD	$AD	$AD	$AD	$AD	$AD	$AD	$AD
.			$AE	$AE	$AE	$AE	$AE	$AE	$AE	$AE
RETURN			$8D	$8D	$8D	$8D	$8D	$8D	$8D	$8D

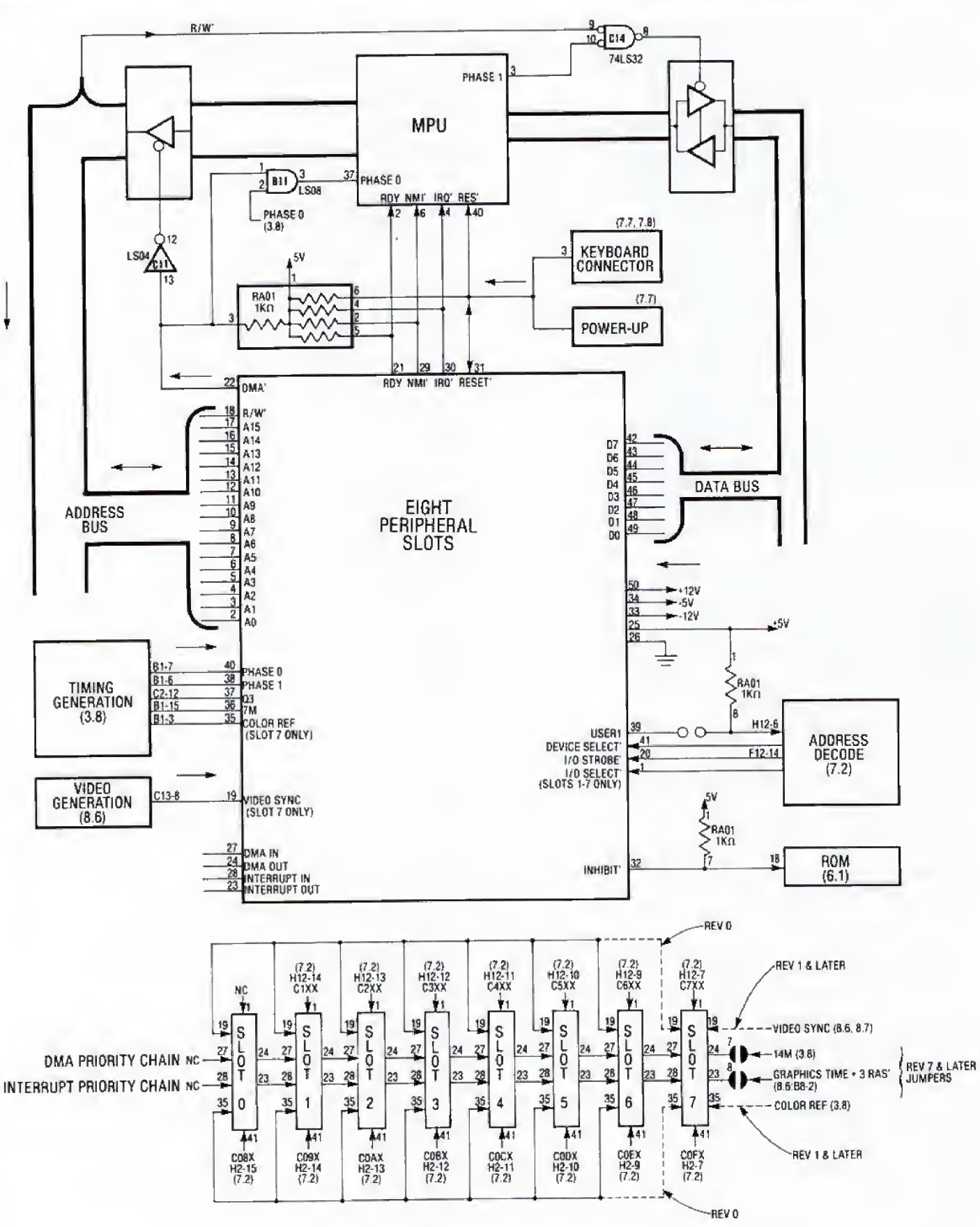


Figure 7.9 Peripheral Slot Connections.

The power supply voltages, for instance, are all grouped together. One quickly sees that all the **power supply** voltages available in the Apple are also available at the peripheral slots.

Paramount in importance among the peripheral slot signals are the **address bus** with **R/W'**, the **data bus**, and the **timing** inputs. Consider what we know about MPU control of the Apple. All data transfer is over the data bus under control of the address bus during PHASE 0. All I/O control is via the address bus. The correct inference is that you can duplicate any motherboard action with a peripheral card design. This gives you an idea of the variety of tasks that can be accomplished. Of course, most of the things people are going to stuff into Apple peripheral slots haven't even been dreamed of yet.

Other timing signals besides PHASE 0 are available at the peripheral slots. You can't have too many of these signals at hand to help synchronize peripheral card functions to the motherboard. PHASE 1, Q3, 7M, COLOR REFERENCE (Slot 7), and video SYNC (Slot 7) are present. Notably absent are 14M, RAS', CAS', AX, video scanner signals (like V5), and 6502 SYNC. With signals like these begging to be connected, it's surprising that pins 19 and 35 of Slots 0 through 6 have no signal connected to them. Pins 19 on these slots are all tied together, and so are pins 35, yet Apple has not defined any function for these connections.

Important **6502 control** inputs are connected to the peripheral slots. These are IRQ', NMI', RESET', and RDY. They are connected in a **wire-OR** configuration with one thousand ohm pull-up resistors so any card can cause an interrupt, reset the Apple, or stop the 6502 via the READY line. The RESET' line is also connected to the keyboard and the power-up reset circuit which generates a RESET' when the Apple is first turned on.

Other wire-OR lines from the peripheral slots are the DMA' line, the INHIBIT' line, and the USER1 line. **DMA'** allows a peripheral card to isolate the MPU from the address bus and data bus so it can gain control of the Apple for fast I/O or other purpose. **INHIBIT'** disables motherboard ROM and opens up $D000-$FFFF addressing for any sort of response. If the USER1 jumper is installed, **USER1** disables all address decoded signals in the $C000-$C7FF range and opens that range for peripheral card response. Note that between INHIBIT', USER1, and the I/O STROBE' disable protocol (reference to $CFFF disables all I/O STROBE' response), provisions exist for stealing the entire $C000-$FFFF address range from the motherboard. No signal exists, however, which allows a peripheral card to steal the $0000-$BFFF range. This is unfortunate since we now have 64K RAM chips and would like to bank switch RAM in banks of 64K. An Application Note at the end of the chapter on RAM (Chapter 5) shows how to steal the $0000-$BFFF address range with a DIP jumper to a motherboard chip socket (see Figure 5.16).

The USER1 inhibit signal represents a potentially powerful control feature that is rarely if ever used. As an example of how it could be used, suppose you had a peripheral card which needed to lock out the $C08X DEVICE SELECT' signal. The card could look for $C08X on the address bus during the last part of PHASE 1. A $C08X detection causes a USER1 flip-flop to flip, bringing USER1 low. PHASE 0 falling causes the flip-flop to flop returning USER1 to its normal high state. USER1 stays low for about three quarters of a cycle and inhibits the normal DEVICE SELECT' generated by $C08X on the address bus during PHASE 0. This is really pretty neat stuff, creating far ranging peripheral design possibilities.

The DEVICE SELECT', I/O SELECT', and I/O STROBE' are address decoded signals which identify addresses on the address bus in the ranges assigned to the peripheral slots. These address ranges could have been assigned by convention only. For example, a Slot 0 peripheral card could easily decode the $C08X address range without the aid of its DEVICE SELECT' input, but then how could you operate that card in a slot other than Slot 0? The card would have to have switches to configure it for different slots. Also, having DEVICE SELECT' decoded on the motherboard lends the force of hardware reality to the convention that $C08X belongs to Slot 0. This is fairly important considering the diversity of sources for Apple peripheral cards. Needless to say, decoding the address ranges on the motherboard also reduces the chip count of most peripheral cards.

The **DEVICE SELECT'** input to each peripheral slot identifies a 16-bit address range assigned specifically to that slot. This range is normally used to command a peripheral to do things like gate data to the data bus or disable one of its functions. A card design may require only one programmed command like a speech synthesis board which says a word when you store a value to its only address. This type of card can use the DEVICE SELECT' to trigger its action and it requires no on board address decoding circuitry. The Apple Firmware card has only two commands, enable and disable. It uses A0 to differentiate between the two possible commands.

A card can distinguish between 32 possible commands in the DEVICE SELECT' range by decoding the states of A0, A1, A2, A3, and R/W'.

The **I/O SELECT'** signal identifies a 256 byte addressing range uniquely assigned to each of Slots 1 through 7. This address range is normally used by a 256 byte program in ROM or PROM. It has to be taken up by a 256 byte program if the card is to be capable of response to BASIC **"PR#"** and **"IN#"** commands. What these commands do is cause program flow to vector to the first address of the I/O SELECT' range, so there must be a program stored there if PR# and IN# are going to work. PR#0 and IN#0 are different. They cause program flow to vector to the video output and keyboard input routines rather than to a nonexistent program in Slot 0. There is no I/O SELECT' for Slot 0 so the card in Slot 0 will not normally be the type of card which is capable of replacing the keyboard and television as the Apple's primary input and output devices. Slot 0 normally contains a memory expansion card of one sort or another which steals the $D000-$FFFF addressing range from motherboard ROM.

The **I/O STROBE'** identifies $C800-$CFFF addressing for all eight peripheral slots. This address range could have been taken up by a seventh ROM on the motherboard, but the Apple designer elected to make it available to all of the peripheral slots. The idea is to store a big I/O handling program on a 2K ROM or PROM on a peripheral card and then give that card sole access to $C800-$CFFF when it is active for input or output. As is described in detail in Chapter 6, the peripheral cards utilizing this **"seventh ROM"** must deactivate response to the I/O STROBE' when $CFFF is detected on the data bus during PHASE 0. This protocol prevents two ROMs from simultaneously trying to control the data bus when I/O STROBE' goes low. The "seventh ROM" capability makes possible such peripherals as smart printers, smart 80-column cards, and smart EPROM programmers with driving programs stored in firmware.

The remaining four peripheral signals are DMA IN, DMA OUT, INTERRUPT IN, and INTERRUPT OUT. These are the DMA and interrupt **priority chain** connections described in great detail in the 6502 chapter. They are there to keep two or more cards from trying to simultaneously perform similar functions. Only one card can perform DMA or interrupt the MPU at a time. The priority chains can be used to keep order by giving high priority to lower numbered slots. When Apple designed their firmware card, they realized that what they needed but didn't have was an INHIBIT' priority chain. Their solution was to steal the DMA priority chain and hang the consequences. Then they changed their minds with the 16K RAM cards and decided they didn't need an INHIBIT' priority chain after all; so they jumpered the DMA IN pin to the DMA OUT pin. With no published usage guide, and Apple's inconsistent example, the DMA and interrupt priority chains are really available for any sort of serial communication between slots.

Some changes were made in Revision 7 which did not get into the schematic diagram in the *Addendum* to the *Reference Manual*. I traced these out on a "dash C," RFI Revision Apple II. Figure 7.10 is the result. Basically, some video timing signals were made available to Slot 7 and to a 4-pin connector mounted between B2 and C2 on the motherboard. Also, there are mounting holes for a switch and resistor at the E1/E2 position on the motherboard. A reasonable guess is that these changes support some real or planned Slot 7 peripheral card.

THE APPLE I/O SYSTEM: KSW AND CSW

The peripheral slot capabilities are determined by the signals connected to them, but our perception of how they work is very much colored by the operating systems that normally control them. The precedents for I/O control in the Apple were established by the old **Monitor ROM**. The main precedent is that memory locations $36 and $37 always contain the address of the Apple's primary output routine, and locations $38 and $39 always contain the address of the Apple's primary input routine. $36 and $37 are referred to in the monitor listing as CSW (Character output SWitch), and $38 and $39 are referred to as **KSW** (Keyboard input SWitch).

The way the Apple presented itself to us with the old monitor was this: at power up, KSW was set to the address of a firmware routine **(KEYIN)** which waits for a keypress while it flashes a cursor; CSW was set to the address of a firmware routine **(COUT1)** which stores the accumulator in TEXT memory while keeping track of the next screen memory address. Then the monitor command interpreter was entered. This program, as well as BASIC, talks to humans through the **GETLN** (GET LiNe) routine. GETLN gets a series of characters from the primary input device which it finds by doing a JUMP INDIRECT to KSWL ($38). After it gets each character, it stores it in an **Input Buffer** (memory locations $200-$2FF) and sends it to the primary output device by doing a JUMP INDIRECT to CSWL ($36). GETLN continues to input and output characters until it receives a carriage

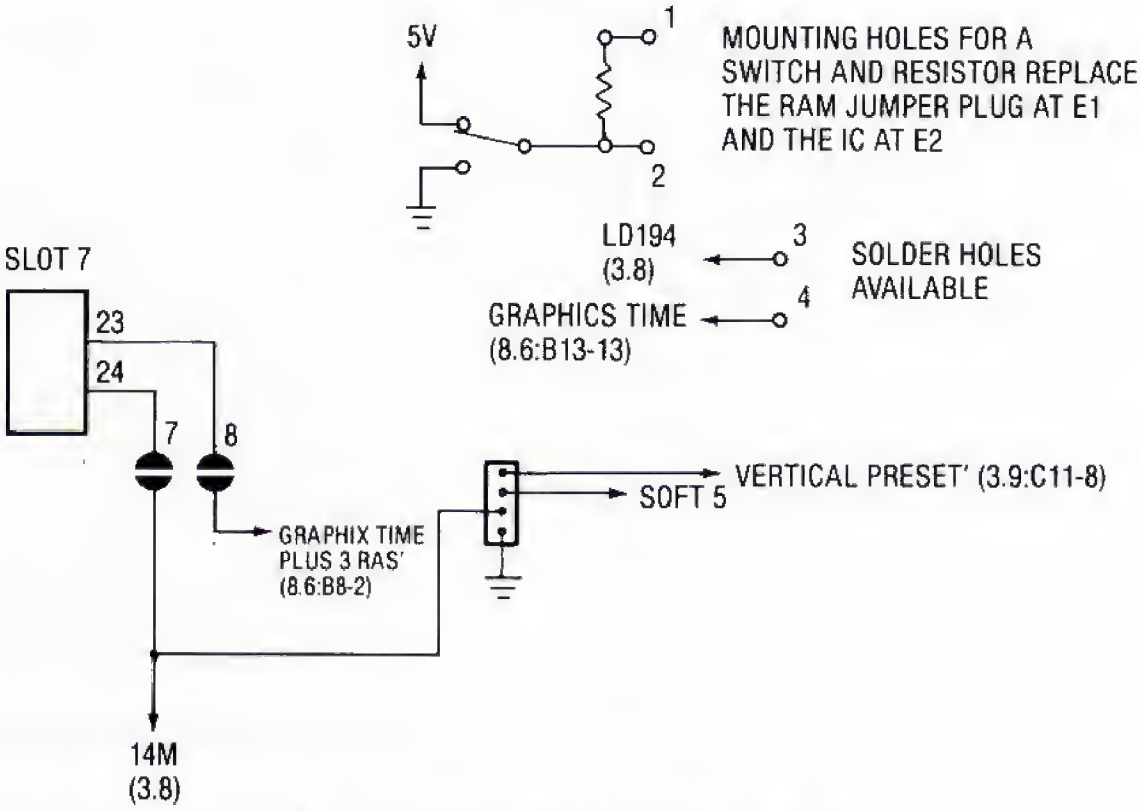


Figure 7.10 Some Revision-7 Additions.

return code from the input device. The program which called GETLN is then able to examine the "line" of data in the input buffer and take action based on its contents. The GETLN routine is largely responsible for our impression of how the Apple talks to us.

The **Autostart ROM** utilizes the same GETLN routine with its JUMP INDIRECTs to KSW and CSW for input and output. The main difference is that if a disk drive is installed, the Autostart ROM boots the DOS at power up, and the DOS connects itself to KSW and CSW. In both monitors, the same system is in effect where every input is followed immediately by output, and the input and output routines are determined by KSW and CSW.

CSW and KSW are the **I/O links**. You can link the driving program for any device to the Apple and make it the primary input or output device. This is because most programs perform input or output by jumping to the address contained in KSW or CSW. As an example, you can connect a serial output device to one of the annunciator ports and place a control program in RAM. You then make this device the Apple's primary output by placing the entry address of your control program at CSW. If your program is typical, after it outputs each character to your device, it jumps to the COUT1 routine so the character is also output to the screen.

Any peripheral slot can be assigned as the Apple's primary input or output device. Slots 1 through 7 can be so assigned by doing a "PR#n" or "IN#n" from BASIC or a "n CONTROL-P" or "n CONTROL-K" from the monitor. When a PR#1 is performed, $00 and $C1 are stored at locations $36 and $37. This means that if you do a PR#1, the card in Slot 1 had better respond to $C1XX addressing with a program, because the 6502 is going to be executing at that address real soon.

To assign the card in Slot 0 as the primary input or output, you must link a Slot 0 driving routine to the Apple via KSW or CSW. In normal practice, Slot 0 will not contain a card that is capable of being the Apple's primary input or output device.

In disk based systems KSW and CSW are normally set to addresses $9F81 and $9EBD in the DOS. Then all input/output passes through the DOS, which checks to see if it is disk related. For example, "CATALOG" is not a valid BASIC command, but it can be executed from BASIC while the DOS is connected, because it is a valid DOS command. While you are entering BASIC code from the keyboard with DOS connected, entries are checked for DOS validity before control is passed to the BASIC interpreter for command processing. If a

BASIC program is actually running, the DOS does little processing of input or output data except to check output data for a leading "CONTROL-D" character. The "CONTROL-D" is a flag which tells the DOS that a disk related command follows.

Entering PR#2 while the DOS is connected does not make Slot 2 the primary output slot. KSW and CSW still will contain $9E81 and $9EBD. The DOS intercepts the PR#2 and does its own output setting routine. DOS maintains its own I/O links—we will call them DOSKSW and DOSCSW. DOSKSW is $AA55 and $AA56, and DOSCSW is $AA53 and $AA54. The PR#2 with DOS connected results in $AA53 and $AA54 being set to $00 and $C2. Slot 2 becomes the secondary output behind the DOS. If a PR#2 is performed from a running program, Slot 2 will probably actually become the primary input and output device, or it may not become connected at all, depending on its $C2XX firmware. If the $C2XX firmware automatically sets both CSW and KSW to some value, then DOS will be disconnected and Slot 2 will be connected as primary input and output. If the $C2XX firmware leaves DOS connected at KSW, then the first time an input is performed, the DOS will disconnect Slot 2 and reset CSW to $9EBD. The way to do a PR#2 from a running BASIC program and leave the DOS connected is to do a PRINT CHR$(4); "PR#2". The CHR$(4), CONTROL-D, flags the DOS that a disk related command is following. The DOS type PR#2 is performed, making Slot 2 the secondary output behind the DOS.

Similar steps must be taken in assembly language programs. If you change CSW to $9000, then the first time a character input is called for, DOS will change CSW back to $9EBD. You can do one of three things to get around this. You can change CSW to $9000 and change KSW to $FD1B, disconnecting the DOS entirely and connecting the keyboard as primary input. You can modify DOSCSW (AA53/4) to $9000, leaving DOS connected. You can also store $9000 at CSW and do a "JSR $3EA." This is a DOS routine which takes the value you stored at CSW or KSW and transfers it to DOSCSW or DOSKSW, then restores CSW and KSW to $9EBD and $9E81. $3EA is easy to remember if you remember "3 EACH."

The various peripheral cards can be divided into three categories: those with onboard firmware at $CnXX, those capable of being the Apple's primary input or output device which have no $CnXX firmware, and those which would normally not be the Apple's primary input or output device. The first category includes such cards as 80-column cards, smart printer interfaces, remote terminal interfaces, and the disk controller. The presence of onboard firmware with response to the simple PR#n and IN#n commands should be an important factor in an Apple owner's choice among similar commercial products.

The second category of cards is like the first except the user must load the driving software from disk or other medium. This driving software will typically bury itself above "HIMEM" and link itself to the Apple via CSW and KSW or DOSCSW and DOSKSW. This is a definite step down in convenience from smart cards with firmware at $CnXX and possibly $C800-$CFFF. The program at $CnXX goes a little beyond offering the convenience of PR#n and IN#n commands. Commercial programs such as word processors, assemblers, and data base managers allow records to be output to any slot, if the slot has a $CnXX driver. These programs usually make no provision for linking output to a RAM address.

The third category of cards is not normally linked to the Apple via KSW and CSW. A 16K RAM card is not a conventional I/O device but simple memory expansion. A 128K card, however, may come with an associated disk emulator program which does get linked. The DMA based manual controller shown in Figure 4.8 is a device which would not be thought of in connection with the links. A secondary MPU card would not be linked. A speech synthesis card might be linked, but it would just as often be driven by special purpose subroutines in a larger program.

Understanding which of the three categories the cards in a given Apple fall into is a big step in understanding what is going on in that Apple. The concept of peripheral slots integrated with the bus structure of the motherboard is so powerful that the "spirit" of the Apple may be under the control of any card or associated control program. When the control breaks down and things do not function as they should, the owner has only his own intellect to fall back on to sort things out. Know your peripheral cards; know your motherboard; know your operating systems; know your Apple.

SOFTWARE APPLICATION

PROGRAMMING THE GAME PADDLES

The PREAD routine of the monitor is pointed out by the *Apple II Reference Manual* as a convenient subroutine for reading any of the four paddle inputs to the Apple. Actually, PREAD is used by the Applesoft and Integer BASIC PDL(n) expressions, and it is called by many programs you might purchase for the Apple. There are some limitations to PREAD which are not irreversible limitations of the Apple. They're just weaknesses in a program. This Application Note explores the PREAD routine and illustrates some alternate programming methods for reading the timers.

PREAD is designed to read the paddle whose number (0-3) is contained in the X-register. For instance, if you want to read Paddle 1, you place 1 in the X-register and do a "JSR $FB1E." PREAD will return with the Accumulator scrambled and a number from 0 to 255 in the Y-register which represents the position of the paddle. The way PREAD works is this:

1. It begins by triggering the four timers (LDA $C070).
2. After 10 cycles, it begins polling the pertinent timer ($C064,X) in an eleven cycle loop. The Y-register is incremented in the loop and thus accumulates the number of loop executions. Program flow exits from PREAD when the pertinent timer is found to be reset.
3. If the polling loop is executed 256 times, the routine is exited immediately with 255 in the Y-register.

The PREAD comment in the monitor listing says "COUNT Y-REG EVERY 12 USEC." This is not true; the Y-register counts approximately every 11 microseconds. Possibly the programmer made an error when computing the execution cycles of the instructions involved, and possibly the comment is the only error. The routine may have originally been written for 12 cycle loops then changed to 11 cycle loops because some marginal tolerance components would not work with 12 cycles. Perhaps they forgot to change the comment.

The basis of the workings of PREAD is this: you want a number between 0 and 255 returned. The timer duration will vary between 2 and 3302 microseconds with a 150000 ohm paddle and a .022 microfarad input capacitor. However if the values of both of these components are 10% low, the timer duration will vary between 2 and 2673 microseconds. The 256 loops of 11 cycles take 2760 microseconds in the Apple. 256 loops of 12 cycles take 3011 microseconds. Eleven cycle loops are a good duration to use with plus or minus 10% tolerance components, but a very small number of resistance/capacitance combinations might never allow the PREAD routine to reach a count of 255. It is possible that Apple specifies plus or minus 5% on the capacitors or the potentiometers. Most Apple paddles have a large slack area on the clockwise side where PREAD returns 255 no matter where you set the paddle. This is because PREAD allows for component tolerance. An improved Apple would have a large resistance trimmer pot across each paddle which would let the owner calibrate his paddle set to eleven cycle polling loops.

So the PREAD routine polls the timer in a loop which takes into account realistic possibilities of component variation. That is all to the good, and PREAD is an adequate utility for many purposes. There are, however, some weaknesses in PREAD. Try running the following Applesoft program:

```
10 FOR A = 0 TO 500 :  NEXT :
   A = PDL(0)
20 B = PDL(1) :  HOME :
   PRINT A;"---";B :  GO TO 10
```

It simply reads the paddle inputs and prints them. The FOR/NEXT loop is a short delay to minimize screen flicker. With the program running, leave Paddle 1 fully clockwise and change Paddle 0 to some low setting. The result is that Paddle 0 interferes with Paddle 1. This is because all four timers are triggered every time $C07X is accessed. In the Applesoft program, the "A=PDL(0)" causes PREAD to be called with 0 in the X-register. This triggers all four timers, and when the Timer 0 pulse is short, the PREAD routine is exited in a relatively short period of time. However Timer 1 will still be set if Paddle 1 is further clockwise than Paddle 0. When the "B=PDL(1)" statement is executed, PREAD is entered with X=1, and the timers are triggered again. However the timer pulse may still be high from the previous PREAD routine which read Timer 0, and **the timers are not retriggered by C07X if they have not yet reset from the previous trigger**. This means that the Timer 1 pulse will be dropping after a short period of time after PREAD is entered, even though Paddle 1 is a long ways clockwise.

There are two ways which this sort of interference may be avoided in BASIC programs. One is to always insure there is a time delay between reading different paddles. It does not take much in BASIC. "15 FOR C = 0 TO 0 : NEXT" in the above program does the trick, or just insert a few instructions between PDL(n) expressions. The second way is to poll the timer in BASIC to make sure it is reset before trying to read it. In the above program: "15 IF PEEK(-16283) > 127 THEN 15." Actually, the delay in this last cure is probably long enough to ensure that Timer 1 is reset by the time -16283 is actually examined, but you will be sure if you use it.

The interference between timers is more pronounced when calling PREAD from assembly language programs. This is because machine language is so fast that hundreds of instructions can be executed after a PREAD routine, and some of the timers may still be set. It is also possible for a PREAD to a timer to interfere with a subsequent PREAD to the same timer. Consider the following program sequence:

```
LDX  #Ø
JSR  PREAD
JSR  PREAD
STY  SAVEPØ
```

You may have wanted to cause a paddle variable delay with this sequence. Now if Paddle 0 is fully clockwise, the first PREAD is done normally, but the second one returns a low value instead of 255. This is because the first PREAD is exited as soon as 256 polling loops have been performed. Timer 0 is still set though, and it is therefore already set when the second PREAD is entered. The result is a return value in the Y-register of 70 or so instead of the expected 255.

Misreading a timer due to a previous call to PREAD can be avoided by preceding all calls to PREAD with a check like this:

```
        LDX  PDLNUM
NOTRDY  LDA  PDLØ,X
        BMI  NOTRDY
        JSR  PREAD
```

This simply waits until a timer is reset before attempting to trigger and read it. Of course, there would have been no problem in BASIC or assembly language if this check had been included at the beginning of PREAD.

An aspect of PREAD that should be well understood is that it takes a long time and that the time it takes gets longer as you turn the paddle clockwise. This can be used to advantage in a paddle variable delay routine which allows the user to vary execution speed by adjusting a paddle. More often, the time delay is a nuisance, causing unwanted time delays in a computer that can't afford them. One way to speed programs calling PREAD is to only use counts 0-63. The paddle tweaker becomes aware that there is no control when he goes too far clockwise and controls his Formula-1 racer with less paddle range. Average paddle reading time is reduced by 75% and the sensitivity of the computer action to the paddle tweaker's touch becomes greater. This is tolerable with a paddle set but less tolerable with a one inch joystick which is already very sensitive to the touch.

Assembly language programmers should not feel tied to PREAD. PREAD is handy and often adequate, but it's not the last word in reading paddles. Figure 7.11 is a program which reads Paddle 0 and Paddle 1 simultaneously, an obvious capability since all timers are triggered simultaneously. This paired paddle poller polls the pair of paddles precisely in 22 cycle loops and therefore returns values equal to one half of what they would have been if read by PREAD. The Paddle 0 value is returned in the Y-register and the Paddle 1 value is returned in the X-register. The values can each be shifted left for compatibility with PREAD if this is desirable. The advantage of reading the two paddles together is that paddle reading time is cut in half. The full mechanical range of each paddle is used and the number returned is 0 to approximately 160. It can be argued that no resolution is lost since 1/256 resolution exceeds the practical resolution of a 3/4" diameter carbon potentiometer and possibly the stability of a 558 timer. In other words, it's hard to find a point on the pot where the PREAD routine returns a single value that does not jump back and forth between readings. It is also very hard to adjust the paddle so as to increase or decrease the returned value by one. Resolution of 1/160 is easily good enough for this hardware.

A final recommendation for speeding paddle reading is to integrate the timer polling with other program execution. The time delay problem exists as long as normal program flow must wait thousands of microseconds for a timer to reset. When the paddles must be read often and speed is important, it may be necessary to arrange routines so that they can check the timer states occasionally, only interrupting program flow momentarily. For example,

```
SOURCE FILE: SIMULREAD
0000:              1 ************************************************************
0000:              2 *                                                          *
0000:              3 *                                                          *
0000:              4 *      SIMULTANEOUS READ OF PADDLE-0 AND PADDLE-1          *
0000:              5 *                                                          *
0000:              6 *                        BY JIM SATHER                     *
0000:              7 *                                                          *
0000:              8 *                          1/24/83                         *
0000:              9 *                                                          *
0000:             10 *                                                          *
0000:             11 ************************************************************
0000:             12 *
0000:             13 *
0000:             14 * PADDLE-0 DIVIDED BY 2 IN Y-REG
0000:             15 * PADDLE-1 DIVIDED BY 2 IN X-REG
0000:             16 *
0000:             17 * SIMULTANEOUS READ PROGRAM LOOP IS 22 CLOCKPULSES.
0000:             18 *
0000:             19 * MONITOR PREAD ROUTINE PROGRAM LOOP IS 11 CLOCKPULSES.
0000:             20 *
0000:             21 *
C064:             22 PDL0     EQU  $C064
C065:             23 PDL1     EQU  $C065
C070:             24 PTRIG    EQU  $C070
0000:             25 *
0000:             26 *
----- NEXT OBJECT FILE NAME IS SIMULREAD.OBJ0
1F00:             27          ORG  $1F00
1F00:AD 70 C0     28 DOIT     LDA  PTRIG
1F03:A2 00        29          LDX  #0
1F05:A0 00        30          LDY  #0
1F07:48           31          PHA              GIVE SOME SPACE FOR COUNT = 0
1F08:68           32          PLA
1F09:24 00        33 GOTPDL1  BIT  $0
1F0B:AD 64 C0     34 CHKPDL0  LDA  PDL0
1F0E:10 0D        35          BPL  GOTPDL0
1F10:EA           36          NOP
1F11:C8           37          INY
1F12:AD 65 C0     38          LDA  PDL1
1F15:30 02        39          BMI  NOGOTS
1F17:10 F0        40          BPL  GOTPDL1
1F19:E8           41 NOGOTS   INX
1F1A:4C 0B 1F     42          JMP  CHKPDL0
1F1D:             43 *
1F1D:             44 *
1F1D:24 00        45 GOTPDL0  BIT  $0
1F1F:AD 65 C0     46          LDA  PDL1
1F22:30 F5        47          BMI  NOGOTS
1F24:60           48          RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 7.11 Assembler Listing: A Paddle Read Program.

suppose that you are computing HIRES plot coordinates in a 100 cycle loop. At the end of each loop, you can check the previously triggered timer to see if it has reset yet and increment a counter if it hasn't. You wind up reading the timer with a resolution of about 1/33 which really is sufficient for many tasks. Of course, this would be a complicated program, but the results would be rewarding. Complicated programs are within the capabilities of any reader of this book who has the time and the urge.

HARDWARE APPLICATION

EXTENDING THE GAME I/O SOCKET

Have you ever seen an Apple with two joysticks plugged in? Why not? It's a capability of the Apple. The answer is that when a standard joystick or paddle set is plugged in to the game I/O socket, the pins for one pushbutton input, two paddle inputs, four annunciator outputs, and the C040 STROBE' become inaccessible. Several game I/O extenders are available commercially. This Application Note shows two extension circuits you can build yourself. One is simple, allowing you to plug a joystick or paddle set in and still have a 16-pin DIP socket available with the remaining I/O pins accessible. The other is more complex, allowing you to have two paddle sets and two joysticks simultaneously connected with switched control between paddles or joysticks. This **game I/O extender** also contains an extension socket for connection to other devices.

Let's look at the simpler circuit first. It is pictured in the photos of Figure 7.12. What this is is a paddle set with an extension socket soldered on top of its 16-pin plug. Pins 6 and 10 are removed from the upper socket because these are the PADDLE 0 and PADDLE 1 inputs which are being used by the paddle set. PUSHBUTTON 0 and PUSHBUTTON 1 are fed to the extension socket even though they are used by the paddle sets. Switch inputs can be paralleled, so one of several switches can operate a given pushbutton input. Potentiometers, however, cannot be connected simultaneously to a timer input. They would interfere with each other.

The benefits of the extension socket are obvious, but the modification must be performed carefully to ensure mechanical strength. The first step is to buy a high quality 16-pin DIP socket. You will also need to buy a 16-pin plug and cover like the one used on Apple paddle sets. We assume that the plug to be modified on the paddle set or joystick is so thoroughly glued and sealed that you cannot hope to solder a socket to it. Here is the procedure to mount the extender socket on your paddle set:

1. Separate the cover from the plug on your paddle set. If you think you can solder a socket to this mess, proceed to step 6.
2. If you cannot salvage the old plug, remove the two 560 ohm resistors from it and cut the wires from it which lead to the paddles.
3. Strip one inch of the outer insulation from the wire bundle going to each paddle. This exposes three insulated wires in each bundle. If Apple is consistent, the green wire goes to +5V, the white



Figure 7.12 A Modified Game I/O Plug.

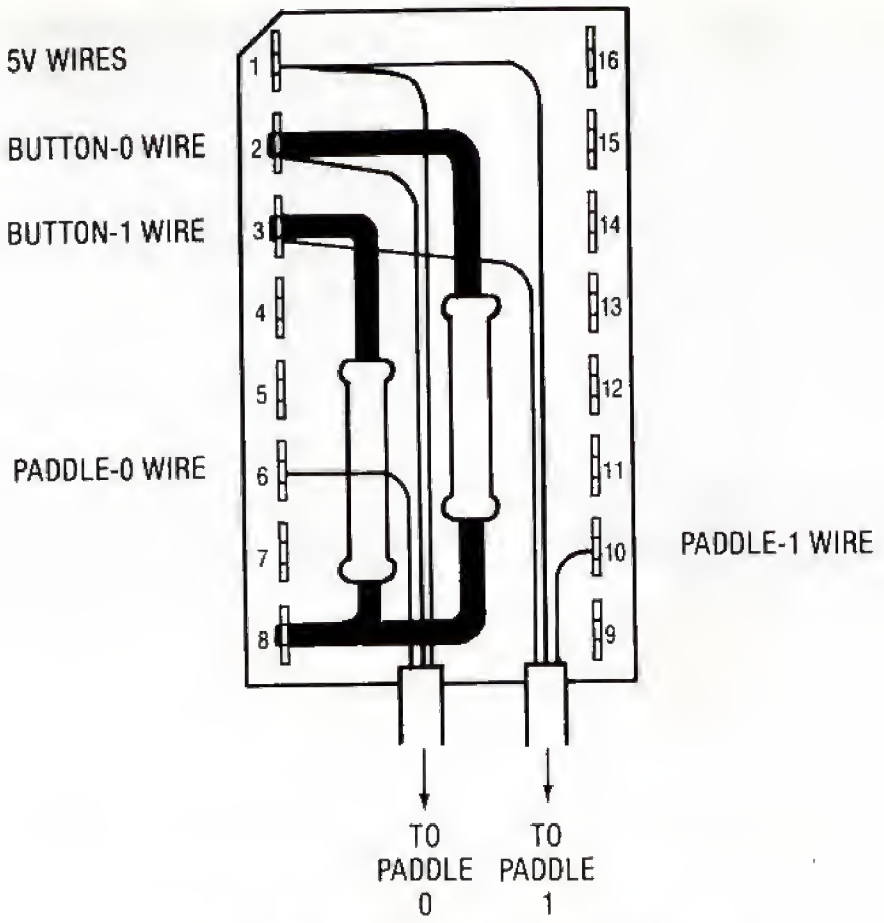


Figure 7.13 Wiring a Paddle Set Plug.

wire is the pushbutton wire, and the black wire is the paddle wire. This should be verified with an ohmmeter. With the ohmmeter, find the two wires connected to the pot. The resistance across them will vary between 0 and 150,000 ohms as the paddle is turned. The wire left over is the pushbutton wire. Now find which wire is shorted to the pushbutton wire when the pushbutton is pressed. This is the 5 Volt wire, and the other wire is the paddle wire.

4. Cut the paddle wires for each paddle back 1/2 inch. Leave the other four wires at their present length. Strip 1/8 inch of insulation off the end of all six wires.
5. Figure 7.13 shows the wiring of the plug. Install the two 560 ohm resistors first, making sure the leads do not extend very far beyond the solder posts. Connect the wires and solder. Use a low wattage iron and do not overheat the pins or the plastic base will be damaged. Insert the plug into a spare socket while soldering to keep the pins aligned if the plastic becomes soft from overheating.
6. If necessary sand down the corner of your socket so the plug cover will slip over it. Pull pins 6 and 10 out of the socket or cut them off if the plastic is molded around the pins.
7. Fit the socket over the paddle set plug and hold this assembly lightly together in a soft jawed vise. All wires should be dressed inside the pins of the socket and the socket pins should be outside of the plug pins. Solder the fourteen pins of the socket to the appropriate pins on the plug.
8. Check out the operation of your paddle set and extension socket.
9. If you desire, fill the area between the plug and expansion socket with epoxy or a sealant like RTV. This will give your assembly more mechanical strength. Do not seal the assembly until you are certain it works correctly.
10. Cut the top off the plug cover so the topless plug cover is 5/16" high. Cut out a small notch for the wires to pass through. Slip the cover over your assembly and glue it on with a small amount of epoxy cement. Remember that you may want to get back in there some day.

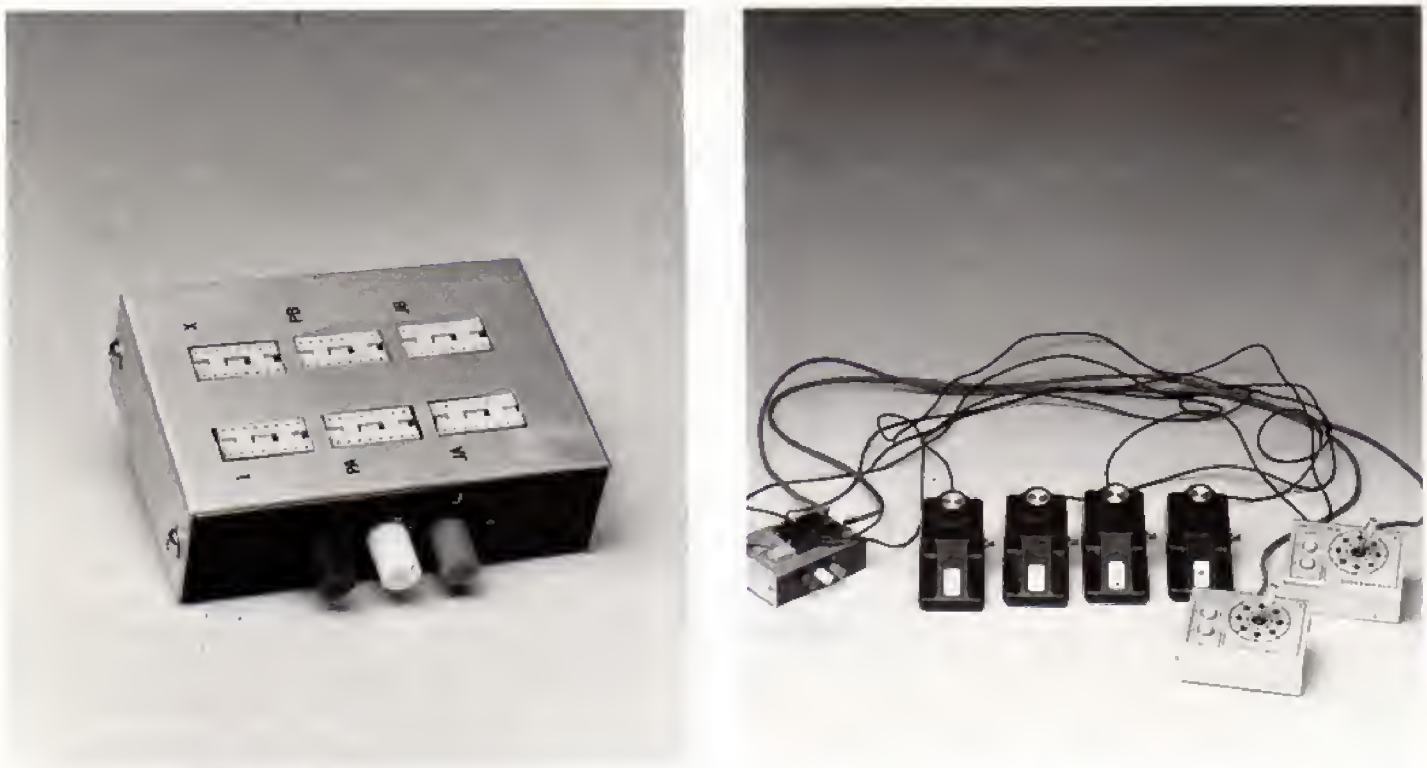


Figure 7.14 This Game I/O Extender Can Support Two Sets of Paddles and Two Joysticks Simultaneously.

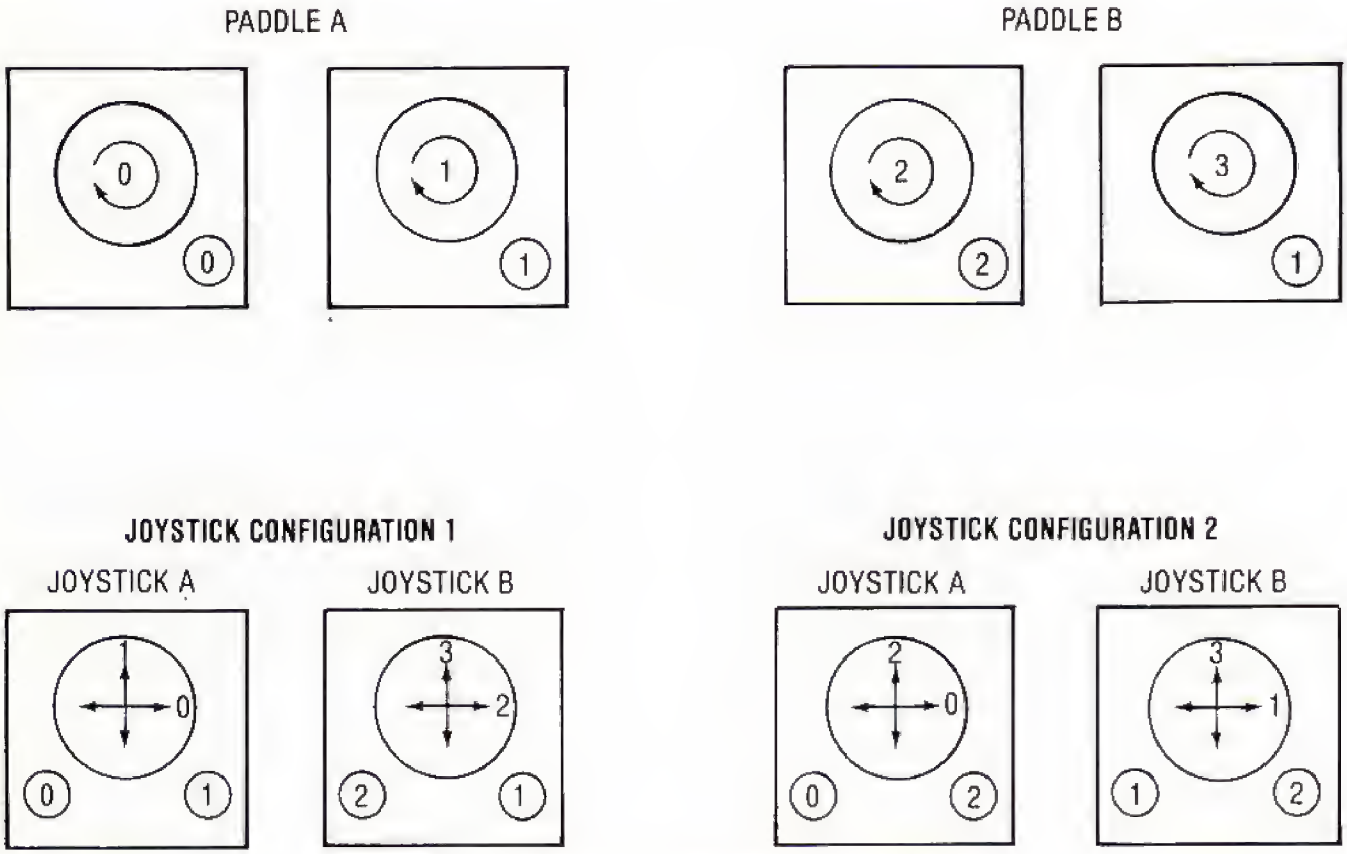


Figure 7.15 Game I/O Extender Configurations.

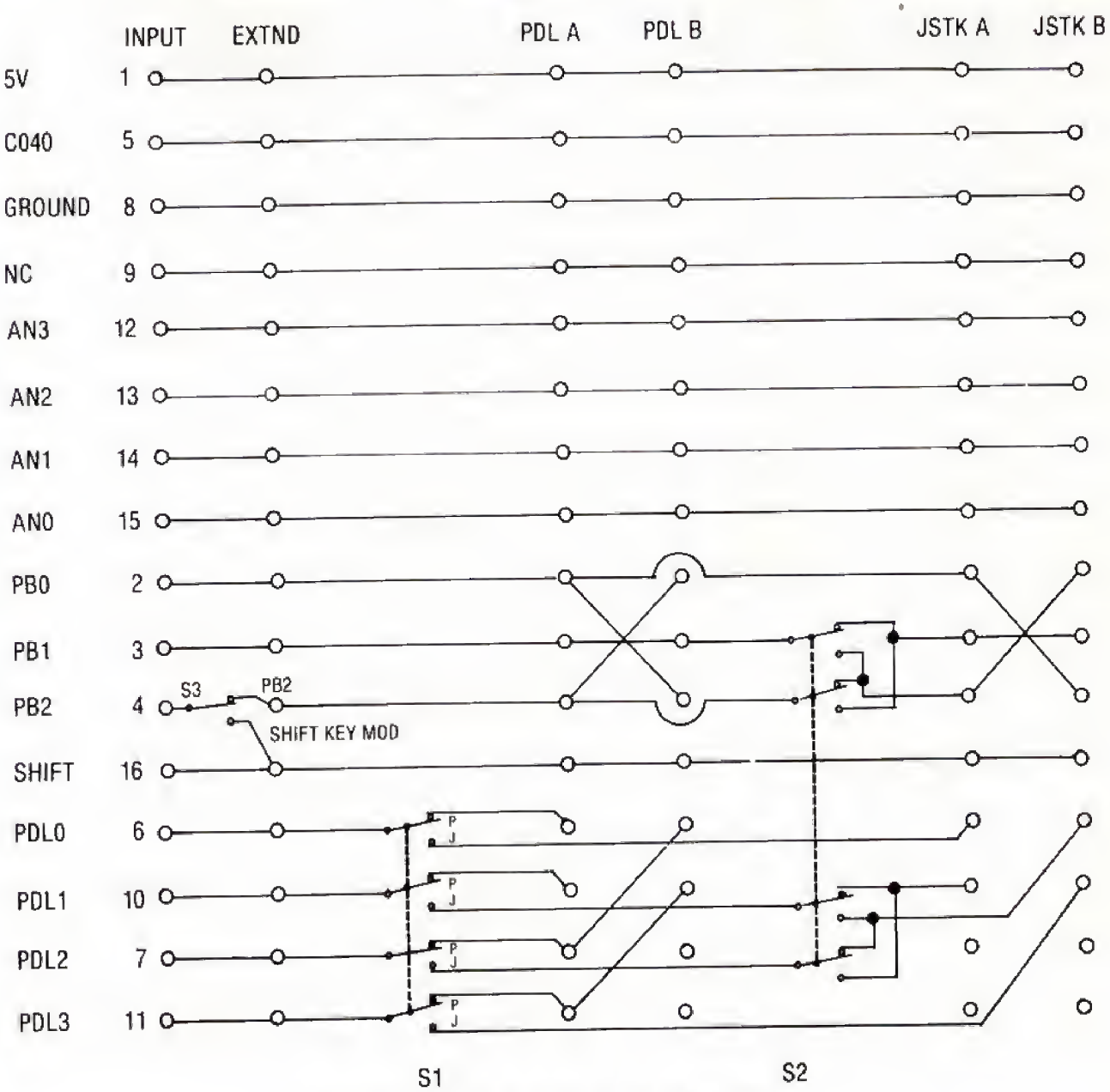


Figure 7.16 Schematic: Game I/O Extender.

Figures 7.14 and 7.16 are photos and a schematic diagram of the more ambitious **game I/O extender**. This unit is meant to sit outside of the computer case, connected to the game I/O socket via a 16-pin DIP jumper. It is basically six 16-pin sockets wired together with some configuration switches. The scheme is this: two of the sockets are meant for paddle sets. One of the paddle sockets is connected normally but the other is connected so a standard paddle set will control Timers 2 and 3. Two of the sockets are meant for joysticks. The joystick sockets are wired so all four timers are utilized by two joysticks. Switch S2 places the joysticks in one of two possible configurations as shown in Figure 7.15. The paddles and joysticks may be connected at the same time. Switch S1 enables either the joystick or the paddles.

A third switch is necessary if you have a button connected to PUSHBUTTON 2 and the SHIFT key mod is installed. The SHIFT key mod works by connecting the SHIFT key to PUSHBUTTON 2, but neither the SHIFT key mod nor a pulled down pushbutton will work if both are connected to PUSHBUTTON 2 at the same time. The game I/O extender allows you to have both installed by selecting between them via S3. The normal SHIFT key mod is to connect one of the SHIFT keys to pin 4 of the game I/O socket. With the game I/O extender the SHIFT key should be connected to pin 16 (normally not connected) of the game I/O socket. Then

switch S3 can select between pin 16 and a paddle or joystick pushbutton for routing to the PUSHBUTTON 2 input.

The construction technique is to mount the six sockets on a general purpose IC board. Use the type with feed through solder holes so wires can be connected on both sides of the board. The board is mounted in a case with six holes which the sockets fit through. A nibbling tool is good for cutting the holes. The installation procedure is: install the sockets in the board; wire the board and switches as shown in Figure 7.16; mount the board and switches in the case. The appearance of your extender will vary with your selection of switch styles and enclosure. Enjoy your extender. It's really pretty handy.

HARDWARE APPLICATION

USING PADDLES AND JOYSTICKS NOT DESIGNED FOR THE APPLE

Suppose you see a special on 100,000 ohm joysticks at your local electronics store and wanted to plug one into your Apple. Could you do it? Sure you could, but you'd need to make a little modification to your Apple so the nonstandard joystick would work with PREAD.

The modification is to change capacitors C7 and C8, the Timer 0 and Timer 1 input capacitors so the two timers will output 2760 microseconds pulses or greater when the joystick is against the high resistance stop. This will allow PREAD to return a value in the range from 0 to 255. If your joysticks are exactly 100,000 ohms, the required capacitance value is 2760/100000 = .027 microfarad. The closest standard capacitor value higher than .027 is .033 microfarad so this is the value you would install. This would give the identical time constant that is found in a standard paddle set with unmodified motherboard (150000 x .022 = 100000 x .033).

You can buy 16-pin DIP plugs with covers just like they use on the Apple paddle sets. One mail order supplier that carries them is Jameco Electronics, 1355 Shoreway Road, Belmont, CA., 94022. You can wire your joystick to this plug and glue the cover on for a clean installation. The paddle schematic in Figure 7.5 should be your guide to connecting the joystick (see also Figure 7.13). The vertical pot should be connected between pin 10 and pin 1. The horizontal pot should be connected between pin 6 and pin 1. Pushbutton connections will depend upon what type of switches your joystick has. It will most commonly have normally open, on/off, momentary switches just like the Apple paddle set. These require a 560 ohm pull down resistor to pin 8 which can be installed on the DIP plug.

You may also wish to install small calibration potentiometers across the main pots so you can exactly calibrate your joystick to PREAD. A good calibration pot, in the case of 100,000 ohm paddles and .033 microfarad capacitors, would be a one megohm pot. You could calibrate a given Apple/joystick combination with a calibration pot, then measure the pot to get a calibration resistance. Then you buy a fixed resistor of this value and solder it across your main joystick potentiometers. Your calibration program, written in BASIC, simply reads the joystick and prints the values on the screen. You push the joystick into the vertical or horizontal stop as appropriate and adjust the calibration pot so the screen just reads 255.

You can also calibrate a standard Apple paddle set so there is no unused 255 slack at the clockwise end. This is done by taking the paddle apart and connecting a 1.5 megohm calibration pot across the two connected terminals of the 150,000 ohm pot. Run a paddle read and display program and adjust the calibration pot so 255 is just displayed when the paddle is fully clockwise. Then measure the calibration pot with a high resistance ohmmeter (a digital volt/ohm meter works) and solder the equivalent fixed resistor across the 150,000 ohm pot.

Here's another suggestion for a modification to a paddle set or joystick. If Timer 2 and Timer 3 are not connected to anything in your Apple, why not connect fixed resistors from their input pins to 5 Volts. This will give you two fixed timing references for programming use, a poor man's real time clock. For example, a 4700 ohm resistor would give a 100 microsecond time reference, and a 47,000 ohm resistor would give a 1000 microsecond reference. The exact duration of the fixed time references in any machine could be determined by PREAD.

To install these fixed resistors, buy a new 16-pin DIP plug and cover like the ones already connected to the paddle set. Then pry the cover off the old paddle set plug. If the old plug is full of glue or if you damage it prying it apart, you will have to use your new plug, wiring it identically to the old plug. Solder a 4700 ohm resistor between pins 7 and 1 and solder a 47,000 ohm resistor between pins 11 and 1. This makes Timer 2 the 100 microsecond reference and Timer 3 the 1000 microsecond reference. Glue the cover back on and you're on the road.

Rebuilding the plug is also a cheap way of extending the game I/O socket. If you've got an input or output you wish to connect in addition to a paddle set, pry or cut the cover off the paddle set plug and solder your add-on device directly to the appropriate pin. Your paddle set is now a combination paddle set/light pen or whatever. Reiterating, be sure you have a spare plug before you start prying apart the old one. You may well destroy the old plug because of the big glob of glue they use when they put the plug together.

HARDWARE APPLICATION

MODIFYING THE KEYBOARD SO CTRL AND RESET MUST BE PRESSED TO CAUSE A RESET

A notorious foible of the original Apple II was the location and shape of the RESET key. It sits next to the dash/equal key and the RETURN key where it is often accidentally pressed by the operator causing aggravation, crashes, and fear of equal signs. Additionally every one year old offspring of a thousand home enthusiasts learned where the button was that caused all the commotion. It was bad enough that various RESET lockout schemes were sold commercially, and Apple competitors advertised that you couldn't hit their RESET key accidentally. The author used to keep a cardboard cover over his RESET key and, to this day, does not reach for the upper right hand corner of the keyboard without second thoughts.

Apple pretty well solved the problem in the Apple II Plus by installing the CTRL-required slide switch on the keyboard. If this switch is in the correct position, both CTRL and RESET must be held down simultaneously to make RESET' go low. This nicely prevents accidental RESETs and takes little getting used to. The purpose of this Application Note is to show how to modify older keyboards so CTRL will be required to make the RESET key function.

As Figure 7.7 shows, both the CTRL switch and RESET switch operate by placing a ground on a line pulled up by a resistor. The modification consists of removing the ground connection from the RESET switch and connecting the RESET switch to the CTRL line. After this, the RESET switch causes the RESET' line to go low only if the CTRL line is low.

The ground is removed from the RESET switch by cutting current traces on the keyboard PC card and wire jumping the broken ground path around the RESET key. Then another wire jumper connects the CTRL line to the RESET switch. The ground connection should be made with a heavier gauge wire since ground for the entire keyboard will flow through this wire. 22 gauge wire is heavy enough. The wire jumpers can be connected to pins extending through the PC board, and they should be soldered. Keep wire length as short as possible so wires will not hang loosely. The following is an installation procedure:*

1. Separate the bottom assembly from the white case following the procedure in Appendix J.
2. Remove the keyboard from the case by removing the three screws on either end.
3. There are two ground traces which must be cut, one on each side of the PC board. Their location is shown in Figure 7.17. Cut these two traces with a pocket knife, verifying discontinuity. As shown in Figure 7.17, one trace must be peeled back slightly to form a little tail to which a wire can be connected and soldered.
4. Solder a jumper between pin 8 of the 16-pin DIP jumper socket and the peeled back ground trace. This wire should be 22 gauge.
5. Solder a jumper between the right terminal of switch 13 and pin 13 of U1 as shown in Figure 7.17.
6. The installation is complete. Operation can be verified by setting the keyboard on an insulated surface next to the base assembly, connecting the keyboard jumper to the motherboard, and applying power. After verification, disconnect the keyboard from the motherboard, install the keyboard in the white case, and install the base assembly on the white case. Don't forget to reconnect all wires, including the keyboard jumper.

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

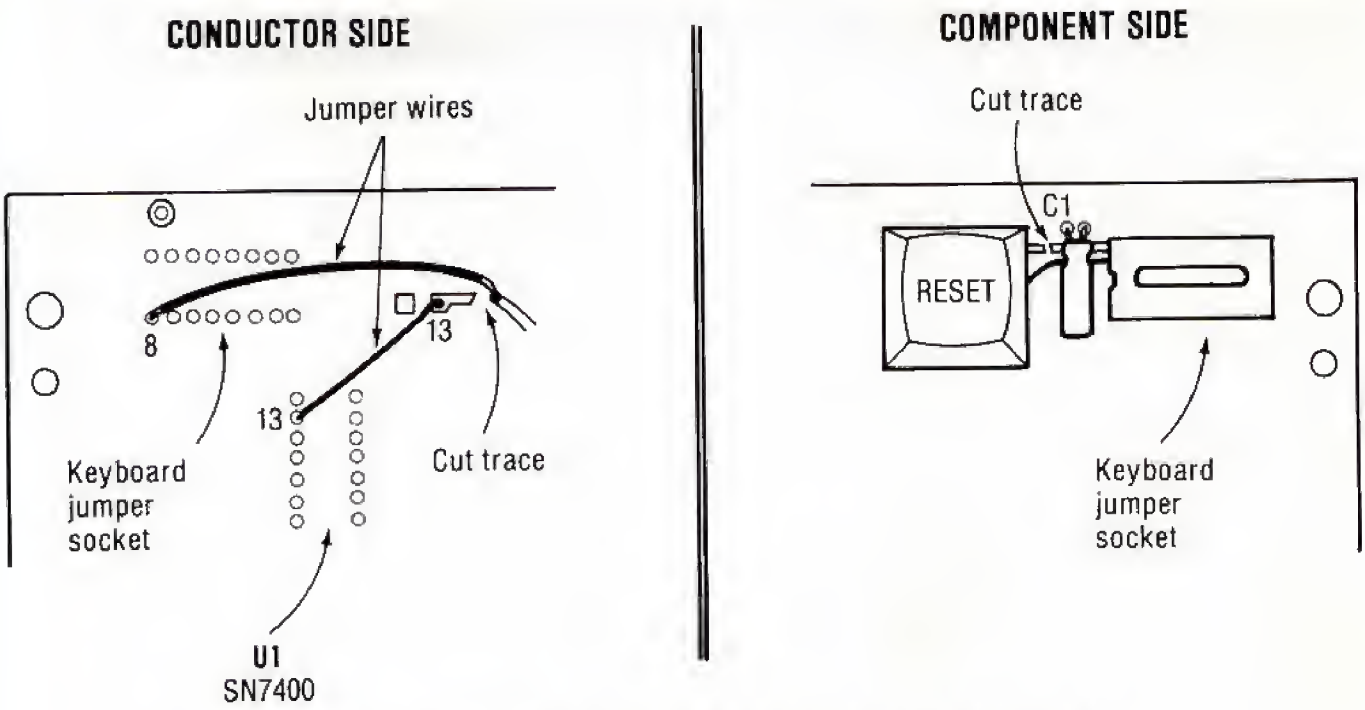


Figure 7.17 The CTRL-RESET Modification, for Older Keyboards.

HARDWARE APPLICATION

MAKING THE SHIFT KEY MODIFICATION

When the Apple was first put together, its designer could hardly have envisioned the variety of purposes for which his creation would be used. He obviously did not envision word processing as an Apple task, or he would have included an upper/lower case keyboard and an 80 column text display. Undaunted, thousands of people use the Apple for word processing anyway, and make the Apple an acceptable word processing computer by installing an 80 column video board and the SHIFT key modification.

The SHIFT key mod was the bright idea of some forgotten soul who noted that the PUSHBUTTON 2 input to the game I/O socket was rarely used. He connected the SHIFT key output to pin 4 of the game I/O socket and interpreted PUSHBUTTON 2 as determining upper or lower case for alphabetic input from the keyboard. This modification has become significant because most word processing programs support it, allowing persons to enter upper and lower case to word processors via the SHIFT keys. The monitor GETLN routine, of course, does not support the SHIFT key mod, so BASIC still will communicate with the keyboard only in upper case.

The SHIFT key mod will not work if a pulled down pushbutton is also connected to the PUSHBUTTON 2 input. This is because the SHIFT key line is pulled up, and you cannot wire-OR a pulled up line and a pulled down line. For this reason, some joysticks with three buttons will be incompatible with the SHIFT key mod.

The SHIFT key mod is most easily installed on the Apple II Plus. This is because the SHIFT' line is highly accessible on the exposed pins of PLUG 1, which connects the main keyboard PC card to the keyboard encoder card . This plug can be seen by removing the Apple's cover and looking in from the rear. The SHIFT' line is pin 24, the second pin from the left. Take a wire which is terminated on both ends with a spring loaded clip. With the computer turned off, clip one end to pin 24 of PLUG 1. Remove your joystick or paddle set from the game I/O socket and clip the other end of the jumper to pin 4 on the joystick plug. Reinsert the joystick or paddle set with the clip attached and you're finished.

Here is a method which takes a little more work but is more usable. Take a wire which is terminated on one end with a spring loaded clip. Solder the other end to the base of pin 4 on a high quality 16-pin DIP socket. Clip the spring clip to pin 24 of PLUG 1 and insert the socket into the game I/O connector. Other devices may then be inserted into the socket.

The SHIFT key mod is more difficult to install on the older keyboards predating the Apple II Plus. There is no convenient exposed point on the SHIFT' line to which a spring loaded clip can be attached. You may solder a wire directly to Key 42 or Key 53 and connect the other end to pin 4 of the game I/O socket. Whatever connection method is used should allow the jumper to be easily disconnected in case the white enclosure needs to be separated from the base assembly. The method shown here is to connect the SHIFT' line to the game I/O socket using a spare line on the 16-pin keyboard connector. This takes a little effort, but when you're finished the SHIFT key mod is fully integrated with the Apple mechanical package with no loose wires to be connected or disconnected. Here is the installation procedure:*

1. Separate the bottom assembly from the white case following the procedure in Appendix J.
2. Remove the keyboard from the white case by removing the three screws on either end.
3. Lay the keyboard face down and solder an insulated wire jumper between Key 53 and pin 4 of the keyboard connector as shown in Figure 7.18.
4. Remove the motherboard from the base assembly following the procedure in Appendix J.
5. Place the motherboard bottom side up and solder an insulated wire jumper between pin 4 of the game I/O socket and pin 4 of the keyboard socket as shown in Figure 7.18. This wire can be held tightly to the motherboard with spots of glue. A hot glue gun works well for this purpose.
6. Reinstall the motherboard on the base assembly.
7. The installation is complete. Operation can be verified by setting the keyboard on an insulated surface next to the base assembly, connecting the keyboard jumper to the motherboard, and applying power. After verification, disconnect the keyboard from the motherboard, install the keyboard in the white case, and install the base assembly in the white case. Don't forget to reconnect all cards and wires, including the keyboard jumper.

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

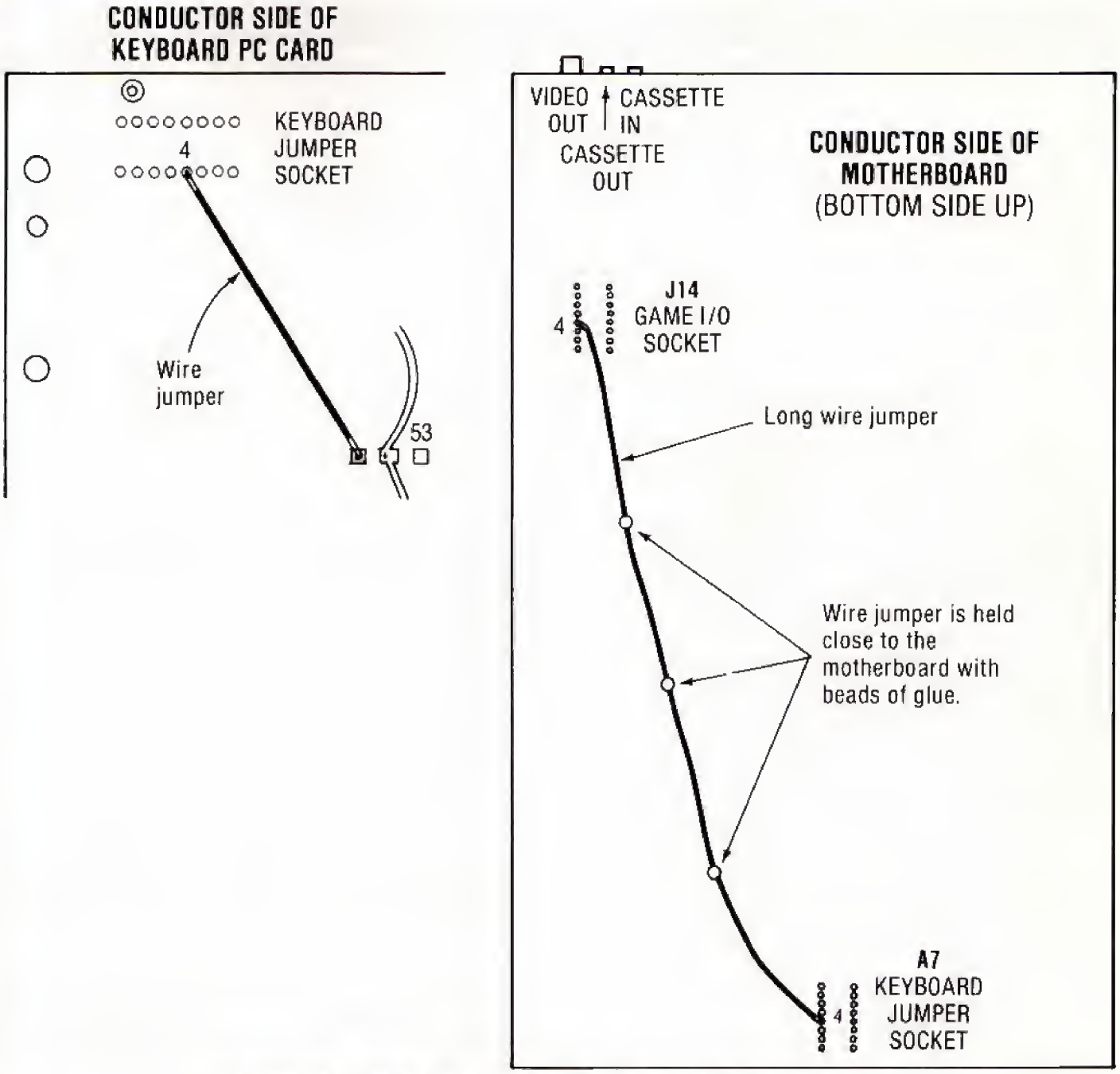


Figure 7.18 Installation of SHIFT Key Mod, for Older Keyboards.

There is an alternate method to the SHIFT key mod for obtaining the upper/lower case alphabet from the keyboard in the Apple II Plus. The keyboard encoder ROM is fully programmed for upper and lower case in the II Plus, and there are holes and connections for a slide switch to enable or disable alphabetic shifting. The switch connections are normally jumpered so alphabetic shifting is disabled. You can change the configuration so that alphabetic shifting is enabled, or you can mount a slide switch and disable or enable alphabetic shifting at your pleasure. Even though the keyboard will output upper and lower case ASCII to the MPU, the monitor GETLN routine assumes everything is upper case only. The keyboard will communicate in upper and lower case only to the limited number of programs written for the Apple which look for lower case alphabetic ASCII in the keyboard input. Your word processor may or may not support this capability in addition to the more prevalent SHIFT key mod.

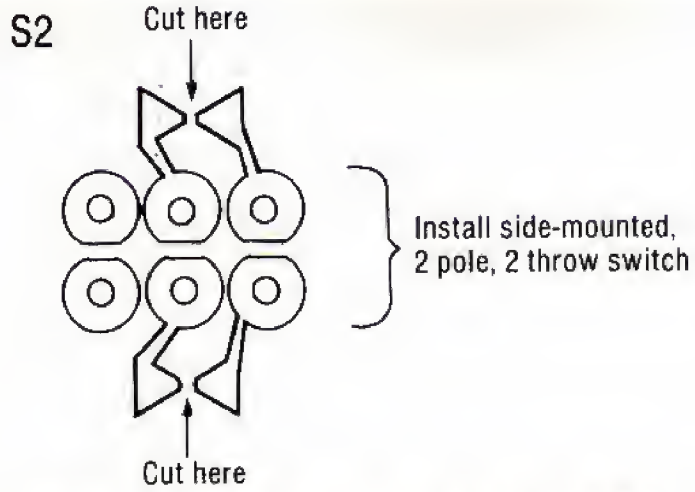


Figure 7.19 Enabling Lower Case, Apple II Plus.

A drawback to consider when enabling alphabetic shifting this way is that you will have to press SHIFT in addition to I, J, M, and K to make CURSOR moves in the ESCAPE mode. This is another case of a hardware capability being minimized by lack of program support. Of course, you can easily install the switch then disable alphabetic shifting if you find its use is not convenient in your system. Here is a procedure for enabling alphabetic shifting on the Apple II Plus keyboard:*

1. Separate the base assembly from the white case following the procedure in Appendix J.
2. Lay the white case upside down on a soft surface. Remove the small electronic assembly board from the main keyboard PC card by pinching the two white plastic retainers while working the board off.
3. Orient the small board so the holes marked S2 are as shown in Figure 7.19.
4. Cut the two bow tie solder jumpers with a pocket knife. Verify discontinuity.
5. Install a miniature double pole, double throw slide switch in the holes and solder.
6. Reinstall the small board on the main keyboard PC card. Install the motherboard base assembly back on the white case. Be sure to connect all cards and wires including the 16-pin keyboard jumper.

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

HARDWARE APPLICATION

INSTALLING A VOLUME CONTROL ON THE APPLE'S SPEAKER

Did you ever wish you could turn down the computer tooter on your Apple when the family was sleeping, but you were being a midnight computer fool? How did I guess? Here are two suggestions on how to silence your Apple. They are illustrated in Figure 7.20.

First, you can connect a 2500 ohm or greater potentiometer in series with the speaker. This potentiometer should be mounted on the case where you can reach it, in a slot in the back or in a hole you drill near the keyboard perhaps. You need to connect the pot to one of the two speaker wires using some sort of plug/jack rig. Don't solder wires directly from case mounted components to base assembly components. Make it so the pot can be easily unplugged. Buy any sort of two wire plug jack connection. Break one of the speaker wires at a convenient place, and connect the two resulting wire ends to the terminals of your jack. Your jack may be permanently attached to the base assembly. Connect two wires to your plug, then connect the other ends to the center terminal and one side terminal of the potentiometer. Make certain volume increases as the pot is turned clockwise. If necessary reverse the pot connections. Mount the pot, put a knob on it, join your plug and jack together, and you're done.

An easier alternative is available if your television or monitor has an audio input jack to an amplifier/speaker associated with a front mounted volume control. You can disconnect the Apple's speaker and rig a connection between the Apple audio signal and the audio input to your external amplifier. Unfortunately, the speaker jack mounted on the motherboard is unsuitable for connection to most audio amps, because one pin is connected to 5 Volts. The circuit is not meant to drive an audio amplifier.

A suitable signal for connection to an audio amp is the signal at the cathode of CR1. You can tap into this signal as follows:*

1. Remove power from the Apple.
2. Cut a length of twin lead wire long enough to reach from the motherboard to your amplifier input. This wire should be terminated on one end with a plug which will fit into your amplifier input jack.
3. Locate CR1, which is marked IN914 or CR1 on the motherboard. It is situated between the game I/O socket and the cassette output jack close to integrated circuit J13 (see Figure 7.20).
4. You have two cut wire ends on your connecting lead. One is a ground lead which goes to the grounded portion of your audio input plug. Wrap the ground lead around the anode of CR1 and solder. The anode is the end of CR1 which is away from the game I/O socket.
5. Wrap the other lead around the cathode of CR1 and solder. The cathode is toward the game I/O socket.
6. Disconnect the Apple speaker from the motherboard, wind the wire up in a coil, and tape it to the base assembly.
7. Plug your connecting lead into the audio amp, turn the Apple and audio amp on, and verify operation. If the Apple won't turn on, turn the power switch off immediately. You probably have the leads in backwards, or you may have the Apple connected to another grounded device with the wires backwards. Determine which leads are reversed and swap them out in the most convenient place.

*Please read the NOTE OF CAUTION at the beginning of the book before making any modification to your hardware.

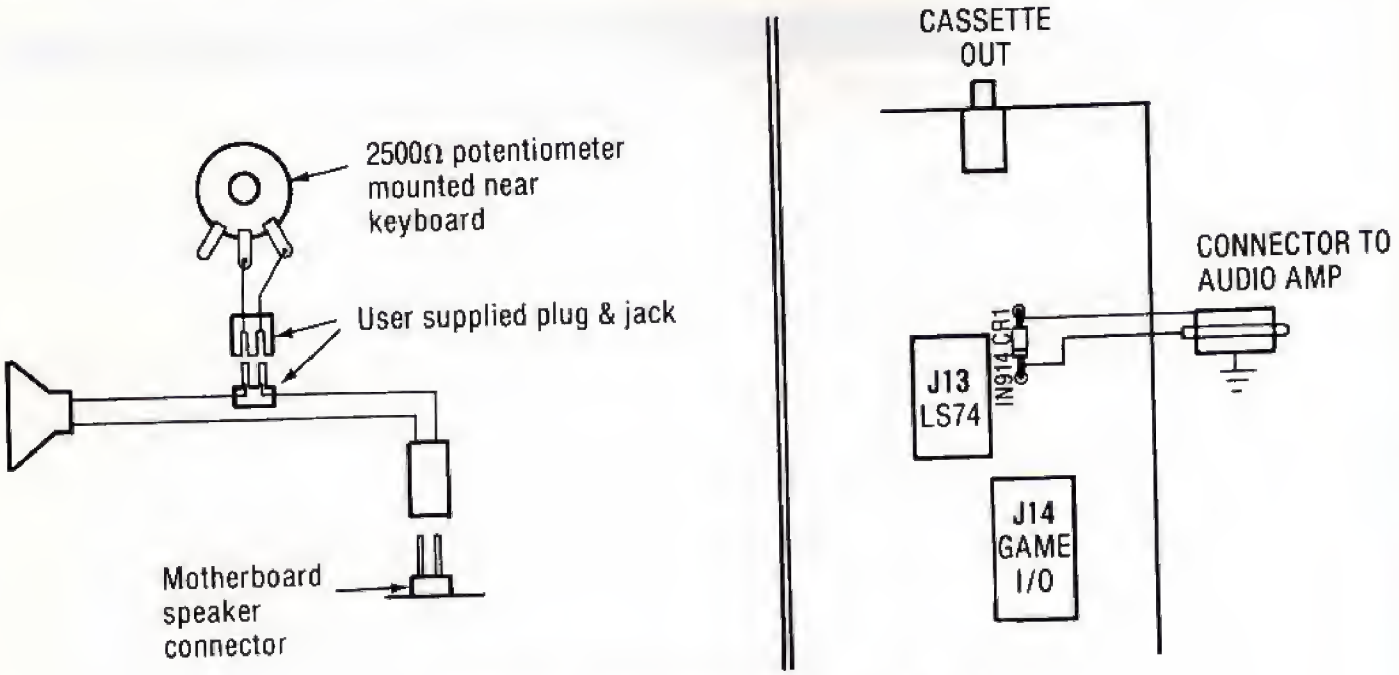
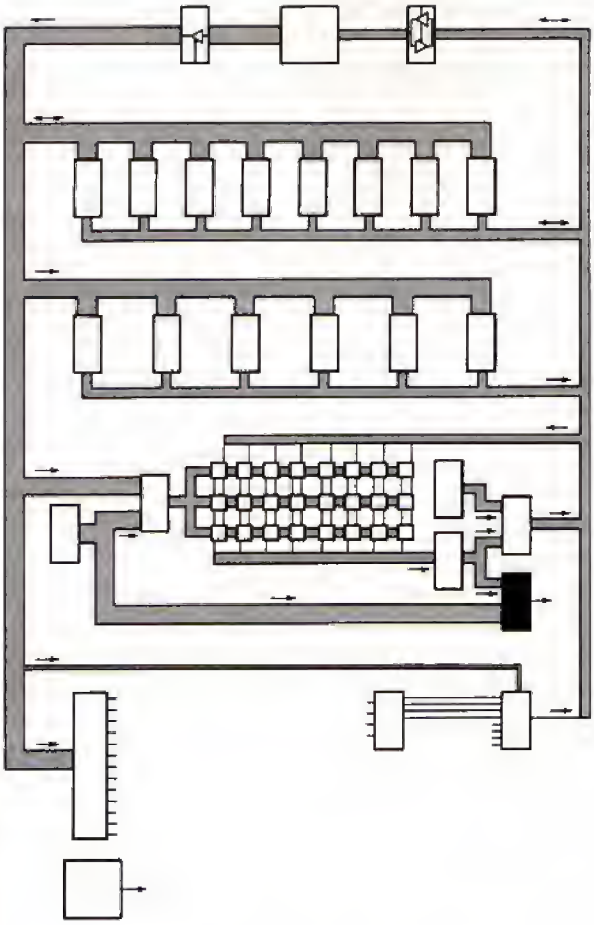


Figure 7.20 Two Volume Control Methods.



chapter 8

Video Generation

The marriage of data processing and video display technology has been one of the most important developments in the advance of computers. Combined with the keyboard, the video display provides a direct communication link between people and computers that makes the old, expensive, physically large computers seem to be just machines. Imagine communication with your Apple using a teletype terminal with no video display. How would that affect important applications like word processing, spread sheet accounting, data base management, graphics display, and Donkey Kong? Without video, the Apple wouldn't be worth owning.

Of course, the Apple does have a video display capability, and a large portion of the motherboard hardware is related to the generation of video. We have seen in other chapters that many features of bus structure, timing, and RAM addressing in the Apple II are dictated by the fact that the dissimilar tasks of stored program execution and video display generation are performed simultaneously in this computer. Additionally, the **video scanner** and **video generator** are functional areas that exist for the sole purpose of making up the video display. All of these functional areas are interconnected in a scheme which allows Apple programs to control the video output.

Figure 8.1 is a simplified diagram of the video display control processes of the Apple. As Figure 8.1 shows, the MPU controls the output of video in a very indirect way. Under direction of the controlling program, the MPU sets the screen mode, computes a correct address in memory, and stores selected code at that memory address. In so doing, the MPU is setting up a small area of the **screen map**. The extent of MPU, and, by extension, programmer involvement in outputting video consists entirely of setting up this screen map. The actual output of video is controlled by the video scanner which scans memory and drives out the map, and by the video generator which processes the map to produce the **VIDEO signal**. You can actually stop the MPU by pulling READY or DMA′ low, and the Apple will continue to output the video to the screen under the control of the map which was set up by the MPU before you stopped it.

Compare this indirect MPU involvement to a printer output port where the MPU—under program control as always—actually stores coded data at a special address to output it to the printer. The

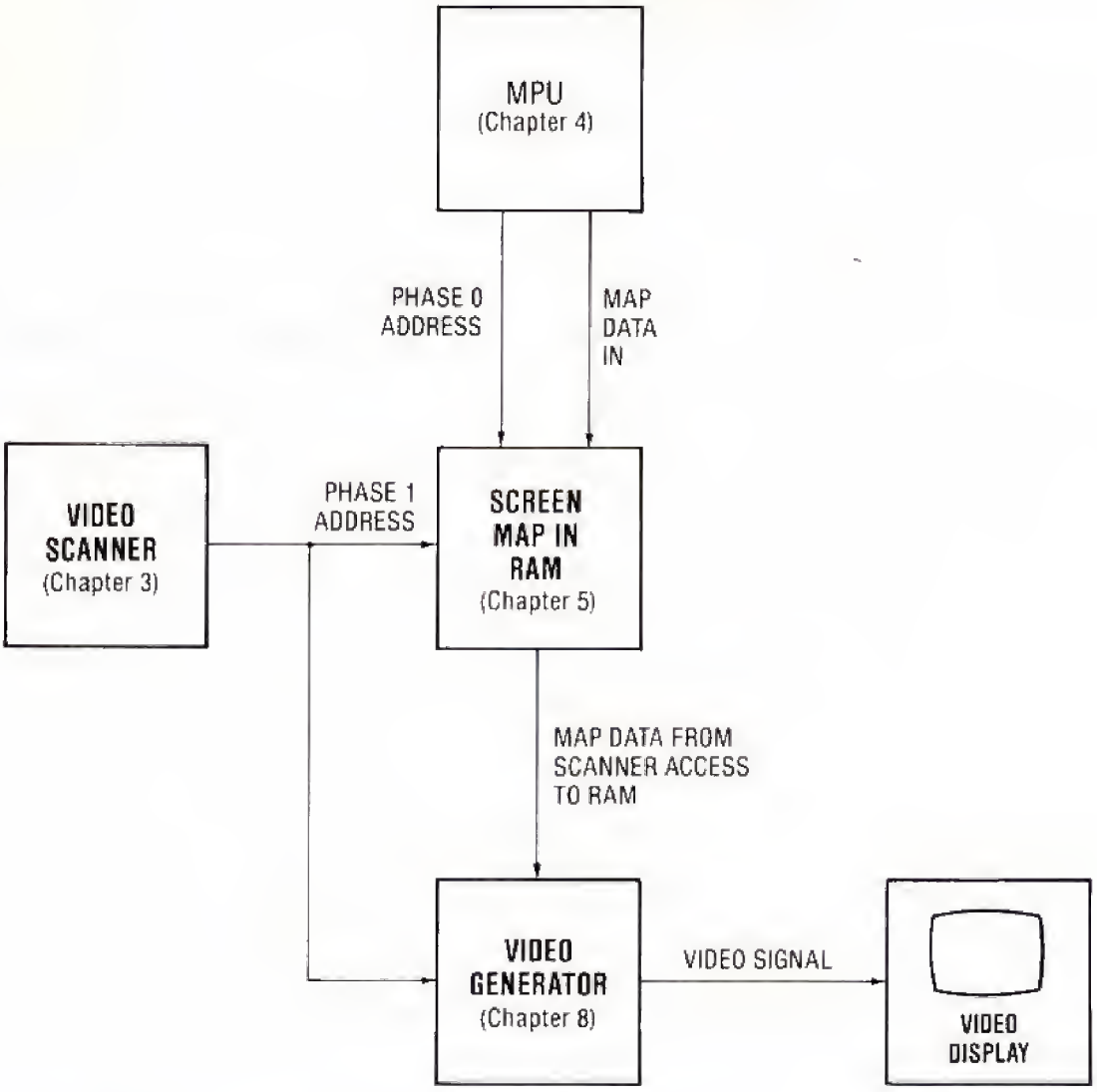


Figure 8.1 The MPU, Video Scanner, RAM, and Video Generator All Play a Part in Creating the Video Display.

sneaking access to RAM by the video scanner is an example of **DMA**, which is like some one else sleeping with your spouse. RAM in the Apple is very promiscuous. It goes to bed with the video scanner every other night. Then on most off nights, it goes to bed with the MPU. The MPU has no idea what unfaithful RAM is up to during PHASE 1.

The video generator must take the offspring of the scanner/RAM affair and interpret it as text, LORES graphics, or HIRES graphics to produce a signal which causes a television or monitor to produce the computer display. This signal is referred to as the VIDEO signal, and it is one of the more complex signals in the Apple. The purpose of this chapter is to discuss the nature of the VIDEO signal, how it is produced in the video generator, and operational features of the Apple II resulting from the way the VIDEO signal is produced. Other chapters of the book contain detailed descriptions which are important in achieving a broad understanding of how the tasks of video generation and program execution are intergrated in the Apple II. These include descriptions of overall video generation (Chapter 1), video scanning within the context of bus structure (Chapter 2), the video scanner (Chapter 3), and RAM addressing (Chapter 4). The subject at hand is the video generator, and we begin our discussion with a description of the Apple II VIDEO signal.

THE APPLE VIDEO OUTPUT SIGNAL

Let's watch a television show for a minute; how about...*Taxi*? That Louie is really something. The picture we see originates with a camera which outputs **composite video**, a signal composed of picture, synchronization, and color information. This signal is routed to a transmitter which modulates a **radio frequency** signal with the composite video and with audio from a microphone. The radio frequency signal is distributed nationwide to local stations which transmit the signal to receivers in their area. The television in your home is a receiver/processor which extracts the audio and composite video from the radio frequency signal and processes it to form the picture we see and the sound we hear.

The previous paragraph could be describing television in any number of countries or continents in the world which broadcast television signals based on similar principles. The exact details of various signals vary, however, among several standard systems used in various areas of the world. The American standards were formulated by the **NTSC** (National Television System Committee) and adopted by the FCC, which allowed black and white television broadcasting after July 1, 1941. Updated NTSC standards for color television broadcasting were adopted by the FCC on December 17, 1953. The American television must be designed to receive and process NTSC standard signals.

When the Apple was built, the FCC frowned on the idea of computers outputting radio frequency signals to a television because it is possible for a tiny amount of the computer signal to be radiated and cause interference with television reception in the neighborhood. Please note that this level of radiation leakage is not a health hazard and is smaller than the man-made electromagnetic fields we all live in. To avoid conflict with FCC regulations, the Apple was designed to output composite video which will drive an NTSC standard composite video monitor. If you modulate a radio frequency signal with the Apple's VIDEO signal, that radio frequency signal will drive an NTSC standard television receiver. Of course, your thirty dollar RF modulator may tend to leak RF radiation and interfere with neighborhood television reception. The television front end takes the radio frequency signal and converts it back to the same VIDEO signal that left the Apple's video output jack. From this point in its circuitry, the television is identical to a composite video monitor.

Figure 8.2 shows the characteristics of the Apple VIDEO signal. It is made up of three components: the **PICTURE signal**, **SYNC**, and the **COLOR REFERENCE BURST**. The signals are added together in such a way that a television can tell them apart. The television can separate the SYNC from the VIDEO signal because, during SYNC pulses, the VIDEO signal is at a lower voltage than at any other time. It also can detect the COLOR BURST, because it knows where to look for it—right behind the horizontal sync pulse on the "back porch" of the horizontal blanking gate.

There is a voltage point on the VIDEO signal called the **black reference**. Voltages above the black reference cause the picture tube electron beam to strike the interior face of the tube with enough velocity for light emission to result. The VIDEO signal goes above the black reference only when it is time to paint, and it stays below the black reference the rest of the time. The SYNC pulses are blacker than black, extending below the blanking signal to a point where they are detected as sync, not a picture signal, by the television. We thus have three signal levels, the white level, the black level, and the sync level. The Apple signal is immensely less complicated than normal television composite video in this regard. The level of the picture signal in composite video can be any voltage between the black level and white level at any instant. This is the way television reproduces the remarkable variety of lighting shades found in a normal television picture. Even black and white television is really black and white and innumerable shades of gray. The Apple video, when color information is not present, is truly black and white.

The horizontal and vertical sync pulses both descend below the black level. Any sharp negative edge in the sync level is interpreted by the television as horizontal sync. Any long duration negative pulse in the sync level is interpreted as vertical sync. The vertical sync pulse in the Apple lasts for four complete horizontal scans. As in normal television video, there are sharp serrations in the vertical sync pulse so that horizontal sync can be detected, even in the middle of the vertical sync pulse.

There are 262 horizontal sync pulses for every vertical sync pulse in the Apple. The horizontal sync pulses occur in the middle of **HBL**, the 25 cycle **horizontal blanking gate**. During HBL, the VIDEO signal is held below the black reference level, creating the right and left black margins on the screen. The picture signal occurs between the

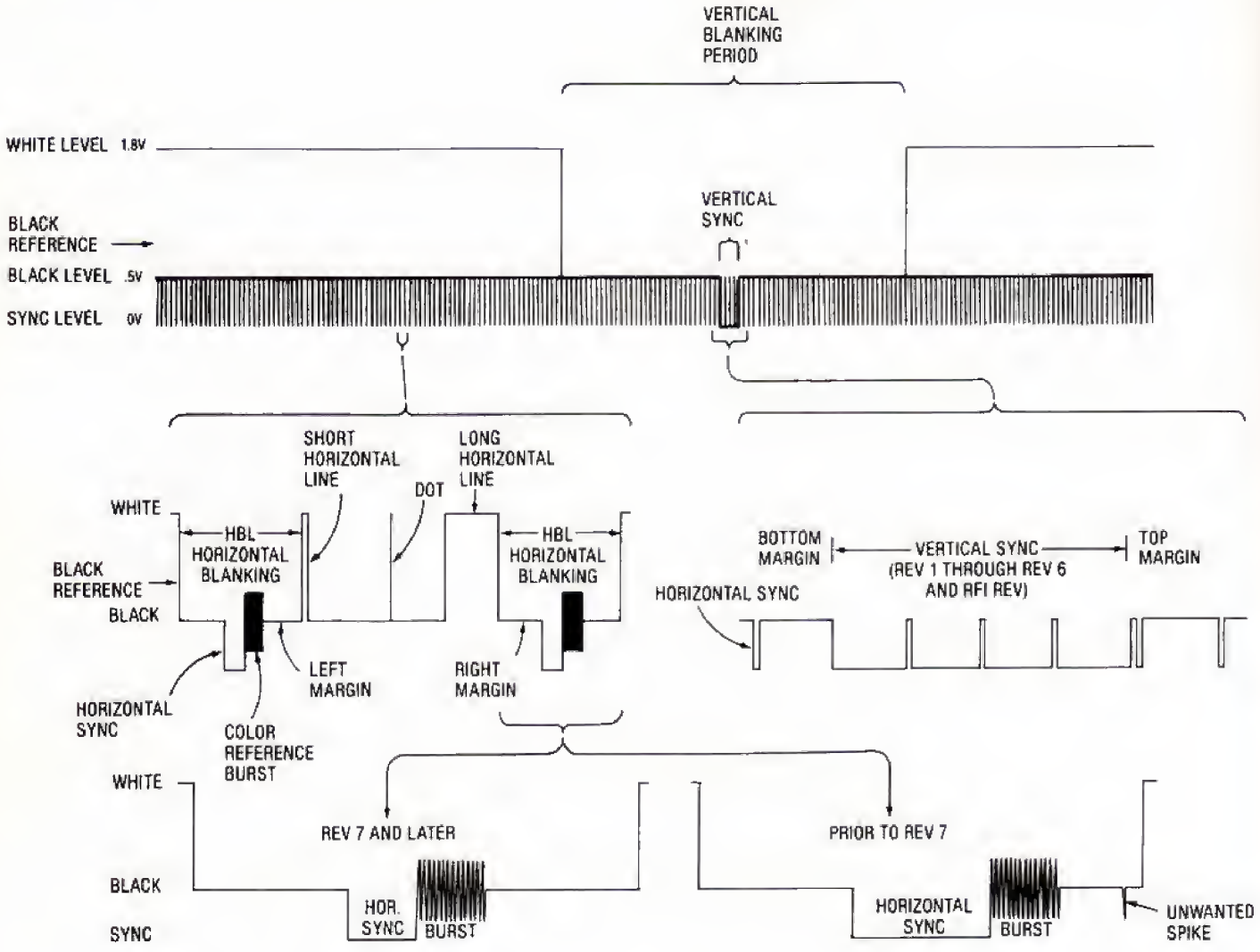


Figure 8.2 The Apple II Video Signal.

horizontal blanking gates, naturally, creating the screen display between the left margin and right margin. The vertical sync pulse occurs in the middle of VBL, the 4550 cycle (70 horizontal scans) **vertical blanking gate**. The VIDEO signal is held below the black reference level during the entire VBL period, creating the lower and upper black margins on the screen.

Assume the electron beam is at the top left corner of the screen, and that the Apple display window on your television is 10 inches across. The beam is moving left to right as you look at the screen, at about 10 inches per 40 microseconds or about 14,000 miles an hour. Since we are at the top, the vertical sync pulse has just occurred and it is the second half of VBL. The beam scans across to the right side of the screen, but we don't see light on the screen because VBL holds the VIDEO signal in the black. The horizontal sync pulse causes the beam to retrace very rapidly to the left side (none of this slow poke 14,000 MPH stuff) so it can scan across left-to-right again. This cycle continues as the beam moves across again and again and less speedily down the screen to the first displayed line, about an inch from the top of the screen.

In the last undisplayed line, VBL ends about an inch from the right side, but HBL begins at the same time, so the screen is still blanked. After the beam scans past the right edge, horizontal sync occurs, causing retrace, and the beam begins the first displayed line. When the beam gets about an inch from the left side of the screen, HBL ends and the VIDEO signal begins switching back and forth between the black level and the white level, decreasing and

increasing the energy in the electron beam to cause bright spots and lines on the screen interspersed with black spots and lines.

About an inch from the right side of the screen, HBL forces the VIDEO signal into the black where it remains until the beam has scanned past the left margin of the next line. This cycle continues for 192 displayed horizontal scans. At the end of the display period of the last displayed line, VBL and HBL begin together, marking the start of the bottom margin. The beam scans the rest of the way to the bottom in the black, then the vertical sync pulse causes a rapid retrace to the top of the screen, where we began our description of the continuous cycle of the electron beam.

In NTSC standard television scanning, a process known as **interlacing** takes place. In interlacing, alternate vertical scans are displaced vertically by half the distance between two horizontal scans. This means it takes two vertical scans to actually paint the complete television picture, and it is a tricky way of increasing vertical resolution without increasing flicker. In NTSC scanning, there are 262.5 horizontal scans in each vertical scan for a picture composed effectively of 525 lines. The Apple SYNC is not set up to cause interlacing. It causes a straightforward vertical scan of 262 lines, 192 of which are displayed.

In earlier Apples, there is a little negative spike on the VIDEO signal at the end of HBL, about a microsecond before the first display spot on the screen. This spike is an undesirable switching glitch generated in the logic which produces the horizontal sync pulse. The spike causes a vertical black line in the left margin of the screen about one text character width before the first displayed dot. It is especially noticeable when television brightness is turned up so there is some brightness level in the margins. Under some circumstances the line appears to be reddish, and it can be a little distracting. It's no big deal, but you've probably noticed it and now you know what causes it.

There are differences in sync generation among the various revisions of the Apple II. In Revision 0, vertical sync lasted 16 horizontal scans and had no horizontal serrations. Horizontal sync lasted through eight states of the video scanner. In Revision 1, the 50 Hz Eurapple jumpers were added, vertical sync was reduced to four horizontal scans with horizontal serrations as is shown in Figure 8.2. In Revision 7, the horizontal sync pulse was reduced in width to four states of the video scanner. Also, possibly through oversight, the horizontal serrations in the vertical sync were changed to double pulses. In the RFI Revision, the double pulse serrations were converted back to single pulse serrations as they were prior to Revision 7. Through all of these sync changes, the Apple was able to sync most televisions and monitors. Presumably the changes were made to offset problems found in certain types of video display equipment. Also, a side effect of reducing the horizontal sync width in Revision 7 was that the little voltage spike at the end of HBL was eliminated.

COLOR SIGNALS

The COLOR REFERENCE BURST is a 14 cycle sample of the COLOR REFERENCE signal from the timing generator. Color television sets are designed to look for a 3.58 MHz signal after the horizontal sync pulse. From this short burst, the television is capable of reconstructing the whole COLOR REFERENCE signal. It does this by "phase locking" an oscillator to the COLOR BURST. By this method, the COLOR REFERENCE is transmitted to the television, but it is not present at the same time as picture information, so it does not interfere with the picture. Since the COLOR BURST occurs during HBL, it is not displayed on the screen. On some television sets, with Apples prior to Revision 7, you can actually see the COLOR BURST on the screen if you turn the brightness and contrast way up. It appears as a series of vertical lines at the far left side of the screen.

In the oldest Apple IIs (Revision 0), the COLOR BURST was always present on the VIDEO signal. This resulted in distracting green and violet text characters. A Color Burst Killer was added in Revision 1 to eliminate the COLOR BURST in TEXT MODE. This removed the colors from the text because the television uses the presence of the COLOR BURST to determine whether an incoming signal is a color signal or a monochrome signal. When no COLOR BURST is present, it completely inhibits color generation in the television so the picture is black and white with no color noise. In MIXED mode, the four lines of text at the bottom of the screen are still green and violet because the COLOR BURST is present.

Picture information in an NTSC standard television signal is divided into two components, the color component and the brightness component. The **chrominance** signal contains the color information of the picture, and the **luminance** signal contains the brightness information of the picture. The two

signals are transmitted simultaneously and processed together in the radio frequency and intermediate frequency stages of a television set. Once the television signal has been converted from radio frequency to composite video, the television separates the chrominance signal from the luminance signal and processes them individually.

The chrominance signal is a highly complex combination of two 3.58 MHz signals 90 degrees out of phase with each other. In an amazing mathematical/electronic manipulation, the chrominance signal contains the color information of each spot on the screen while the luminance signal contains the brightness information of each spot. Part of this mathematical manipulation is that the chrominance and luminance signals are present together in overlapping frequencies, yet do not interfere with each other. In television video processing, the chrominance signal is separated from the composite video, then red, blue, and green color signals are extracted from the chrominance signal by comparing it to the color reference.

In Apple video, the PICTURE signal takes the place of the luminance and chrominance signals of normal broadcast NTSC composite video. The PICTURE signal is a simple binary signal that goes high or low in accordance with text or graphics patterns produced from the screen map during display periods.

In television processing, the Apple VIDEO signal is recovered from the modulated carrier wave and is present at the output of the "second detector." The higher frequency components will, however, look different than they did at the video output jack of the Apple. This is because the square waves of the VIDEO signal are converted to their sine wave components that are within the bandwidth of the television signal paths. In televisions with the normal 4.2 MHz IF bandwidth, those square waves greater than about 1.37 MHz are converted to sine waves of the square wave frequency. This includes the COLOR REFERENCE BURST and higher frequency PICTURE signals, such as those which produce color in the Apple's display. This modified Apple video is present at the input to the luminance and chrominance amplifiers, as well as other sections of the television.

The modified PICTURE signal (a high frequency sine wave, a low frequency square wave, or medium frequency combination) is passed by the luminance amplifier and ultimately controls the brightness of the display. If the modified picture signal is oscillating at 3.58 MHz, it will also be passed by the chrominance amplifier to the synchronous demodulator where it is compared with the reconstructed COLOR REFERNCE to produce red, green, and blue color signals. Thus, the Apple VIDEO signals which produce colored displays on the screen are those with a 3.58 MHz PICTURE signal.*

Processing in a composite video monitor is similar to that in the video sections of a television, but the high frequency square waves may or may not have been converted to sine waves by the time they reach the chrominance and luminance amplifier inputs. If they are not already sine waves, the high frequency square waves will be converted to sine waves in the luminance and chrominance amplifiers. In high frequency response, monochrome monitors, there is no chrominance amplifier. The video amplifiers of these monitors will pass the square waves of the Apple VIDEO signal with little distortion. An exception is LORES gray PICTURE signals, which will be converted to sine waves by monitors of less than 21 MHz frequency response.

The features of Apple graphics are largely dependent on the way the Apple passes color intelligence to the television. There is no need to store HIRES color information in memory. The color information is part of dot positioning. Dot position determines color. Think of the savings in memory over a system where the color information of dots is stored as separate intelligence. The flip side of this coin is: think of the elaborate programs required to produce colored displays dependent on dot position.

There are four classes of Apple video concerning color. One is **black**, the absence of luminance. Two is **white**, the absence of color caused by PICTURE signals less than 3.58 MHz. This occurs when adjacent HIRES dots are turned on, increasing signal pulse width and decreasing frequency so there is no 3.58 MHz signal for the TV to interpret as chrominance. Additionally, white occurs in a "COLOR = 15" LORES block, which is identical to seven adjacent HIRES dots in four adjacent horizontal scans. White also occurs in TEXT mode where there is no COLOR BURST. Three is **gray**, the absence of color caused by 7 MHz PICTURE signals. This occurs in "COLOR = 5" and "COLOR = 10" LORES Blocks. LORES colors 5 and 10 are identical to each other in shade and intensity and are the same as white except less bright. White horizontal lines are caused by long periods of white level VIDEO signal. Gray is caused by 7 MHz oscillation of the VIDEO signal

*Some exceptions to this rule and further discussion on television processing are contained in a Technical Note at the end of this chapter.

between white level and black level. The alternating black level causes gray to be less bright than white. Fourth is **colored video**. This video is 3.58 MHz whether it is HIRES or LORES. There are four such HIRES colors and twelve such LORES colors. Four of the LORES colors are identical to HIRES colors: green, violet, orange, and blue. The remaining eight colors come in four tones: light and dark blue, light and dark magenta, light and dark blue-green, and light and dark brown. Of these eight colors, seven can be produced in HIRES as interference between adjacent bytes of graphics data with DL7 set on one byte and reset on the other.

This explanation of Apple color is based on analysis of the hardware and timing of the video generator. The details are a little involved, but tread thou in my footsteps, tenacious reader, and thy tootsies will not freeze.

VIDEO GENERATOR HARDWARE OVERVIEW

A block diagram of the video generator is shown in Figure 8.3. The upper part is video scanner **logic gating** and the lower part is **PICTURE signal generation**. Video scanner logic gating produces television SYNC, the VIDEO BLANKING gate, and HBL from the states of the video scanner. It also uses the scanner states to gate the COLOR BURST in GRAPHICS modes and to determine GRAPHICS time and TEXT time in MIXED mode. The PICTURE signal generation circuitry monitors latched data from RAM and the screen mode to produce the part of the VIDEO signal which controls the displayed picture. The PICTURE signal is added to the SYNC and COLOR BURST to produce the Apple VIDEO signal.

The heart of screen mode selection is the **PICTURE signal selection multiplexor** at A9. It selects one of seven possible patterns for the PICTURE signal based on its three select inputs. There are four LORES inputs, two HIRES inputs, and one TEXT input to the picture MUX. The selection of inputs depends on current screen mode, DL7 in HIRES, and VC and H0 in LORES. The nature of the seven inputs will become apparent as we progress.

PICTURE signal generation is a load and shift process. Video patterns are loaded every cycle, based on the data resulting from the video scanner access to RAM. Then the patterns are shifted to the PICTURE signal line. There are two shift registers, one for GRAPHICS and one for TEXT. In a fairly complex scheme, the **GRAPHICS shifter** can be configured for HIRES shifting (an 8-bit shift register) or for LORES shifting (two 4-bit end around shift registers). The two configurations are shown separately in Figure 8.3 for illustration, but there is only one GRAPHICS shift register, and it is contained on chips B4 and B9.

The GRAPHICS patterns are loaded directly from the RAM data latch. In HIRES, the lower seven bits of the loaded data are shifted to the PICTURE signal where they control the white and black level of seven dot positions. In LORES only half of the loaded data is used in a given video cycle. Remember that in TEXT/LORES scanning, each displayed memory line is scanned eight times consecutively. During the first four scan lines, the lower four bits of RAM data are used to create the upper block. The upper four bits are used during the second four scan lines, creating the lower block. The first four scan lines are identified by VC', and the second four are identified by VC. In the LORES shift the 4-bit patterns are circulated 3.5 times per block width. This creates colored patterns which seem like solid color blocks on the screen.

The TEXT patterns are not loaded directly from the RAM latch to the **TEXT** shifter. Rather, the latched RAM output addresses a ROM which contains TEXT dot patterns (see Figure 8.4). Recall that the data stored in TEXT screen memory is ASCII. In TEXT scanning, the ASCII for a given character position is driven out in eight consecutive scans. VA, VB, and VC contain the information as to which of these eight lines is currently being scanned, and are therefore part of the TEXT ROM addressing. Suppose the code for "A", $C1, has been latched at the RAM output and VA, VB, and VC are all high. The TEXT ROM is programmed to output the eighth row of the "A" dot pattern when it is addressed this way. A dot pattern is loaded to the TEXT shifter every cycle and shifted out similarly to a HIRES dot pattern.

The two upper bits of screen TEXT ASCII do not address the TEXT ROM in earlier Apples. They are connected instead to some circuits which interpret them to cause NORMAL, INVERSE, or FLASHING dot patterns to be shifted out. DL7 high forces a NORMAL character (white on black). If DL7 is low, DL6 low forces an INVERSE character (black on white), and DL6 high forces the character to alternate between NORMAL and INVERSE at about four alternations per second. This scheme was changed slightly in Revision 7. A much larger

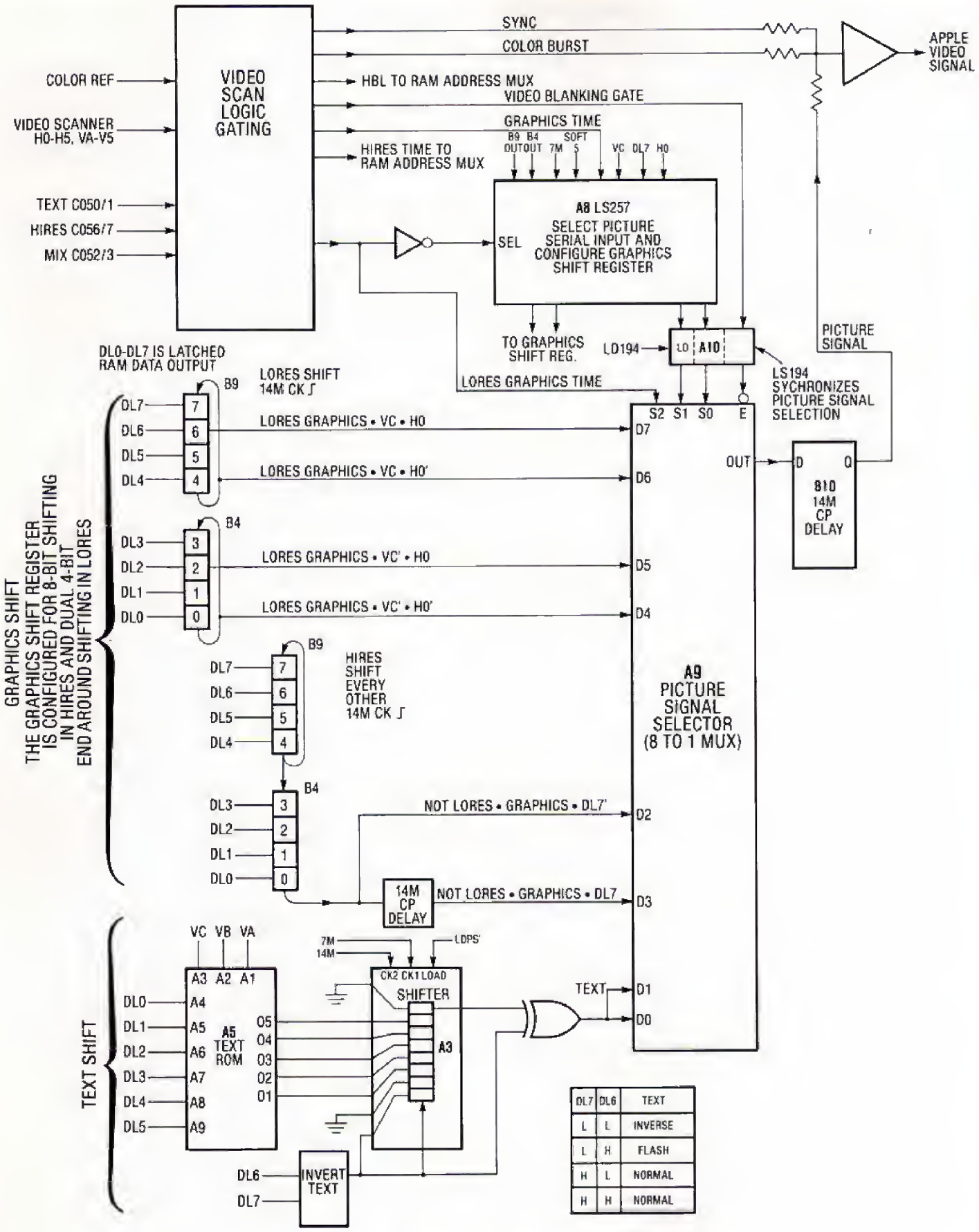


Figure 8.3 Video Generator Functional Flow Diagram.

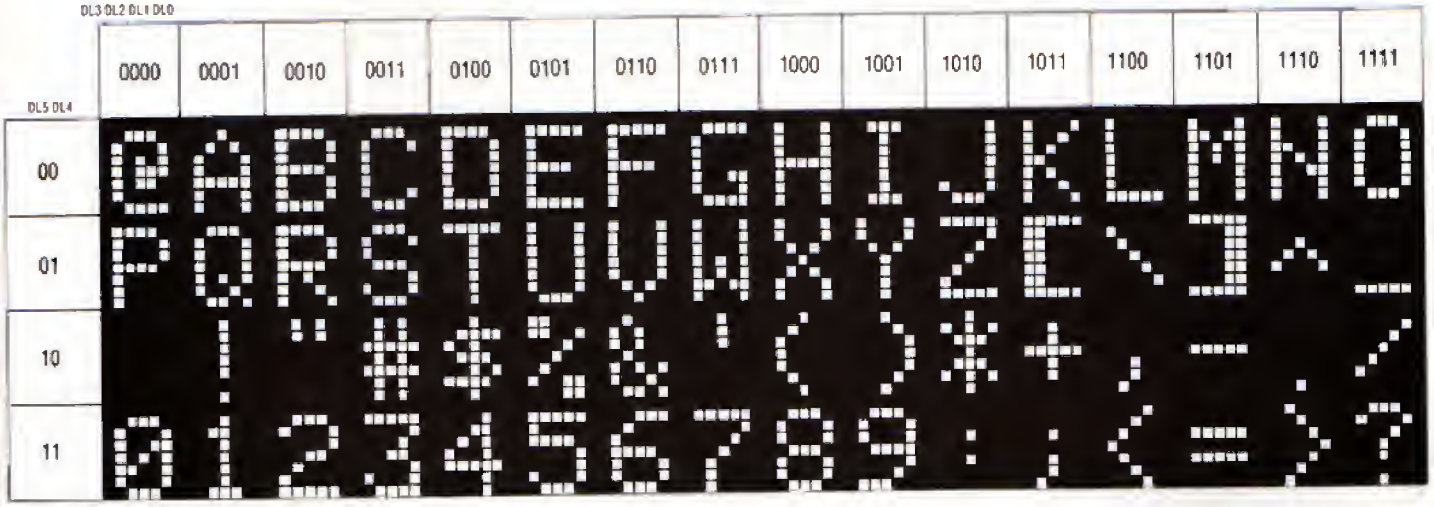


Figure 8.4 Apple Text Patterns.

TEXT ROM is used and DL6 and DL7 are addressing inputs to it. These later Apples function the same way as the older ones when you buy them, but the TEXT ROM can be replaced by your personal EPROM with alternate screen character sets controlled by DL6 and DL7.

VIDEO SCANNER LOGIC GATING

The video scanner logic gating is pretty straightforward. COLOR BURST, television SYNC, and video blanking gates are generated by simple logic gates with modifications possible via "Eurapple" jumpers. The Revision 1 schematic diagram of this logic gating is shown, along with the rest of the video generator, in Figure 8.6. The parts of the video generator which are different in later revisions are shown in Figure 8.7.

VBL (Vertical BLanking) is V4 • V3, true during 70 states of the vertical portion of the video scanner. HBL (Horizontal BLanking) is H5′ • (H3′ + H4′), true for 25 cycles starting with HPE′. The VIDEO BLANKING gate is VBL + HBL, and it is connected to the enabling input of the PICTURE signal MUX via a flip-flop which synchronizes it to LD194, an important video output timing signal. When the synchronized VIDEO BLANKING gate goes high, the VIDEO signal is forced into the black. The LD194 synchronization results in blanking starting and ending just as PICTURE signal generation ends and begins.

The television SYNC signal goes low for short periods of time (horizontal sync) in the middle of HBL and for long periods of time (vertical sync) in the middle of VBL. The television SYNC in the middle of long blanking gates creates the Apple displayed window, surrounded by black margins on all sides (see Figure 8.5). It is normal to think of horizontal scan lines as beginning with horizontal sync, but in the Apple it is logical to think of horizontal scan lines as beginning with HPE′, the first cycle of HBL. This is because HPE′ is the preset of the horizontal portion of the video scanner, and it is the point where the vertical portion increments. Inside the Apple, video related signals tend to switch just after the display period ends. So as not to confuse it with the television horizontal scan, we will refer to the 65 cycle period beginning with HPE′ as the horizontal PERIOD.

Generation of the video SYNC signal was changed in every major revision to the Apple II, so it's a little difficult not to get confused. The HIRES memory scanning map of Figure 5.9 shows very clearly when SYNC occurs in the RFI Revision Apple. SYNC generation in the other revisions differs slightly. Reference to Figures 5.9, 8.2, 8.6, and 8.7 should help clarify the points of the following discussion.

In Revision 0, horizontal sync is HBL • H3. This makes SYNC go low for eight cycles beginning on the tenth cycle of HBL. The Revision 0 horizontal sync pulse is shown in detail in the lower right corner of Figure 8.2. The vertical sync in Revision 0 is V4 • V3 • V2 • V1′. This makes SYNC go low during the 16 vertical scanner states 111100000-111101111 (horizontal PERIODs 224-239 assuming the top displayed line is scanned during PERIOD 0). The Revision 0 vertical sync pulse would be illustrated in Figure 5.9 as solid rows of #s in lines 224-239.

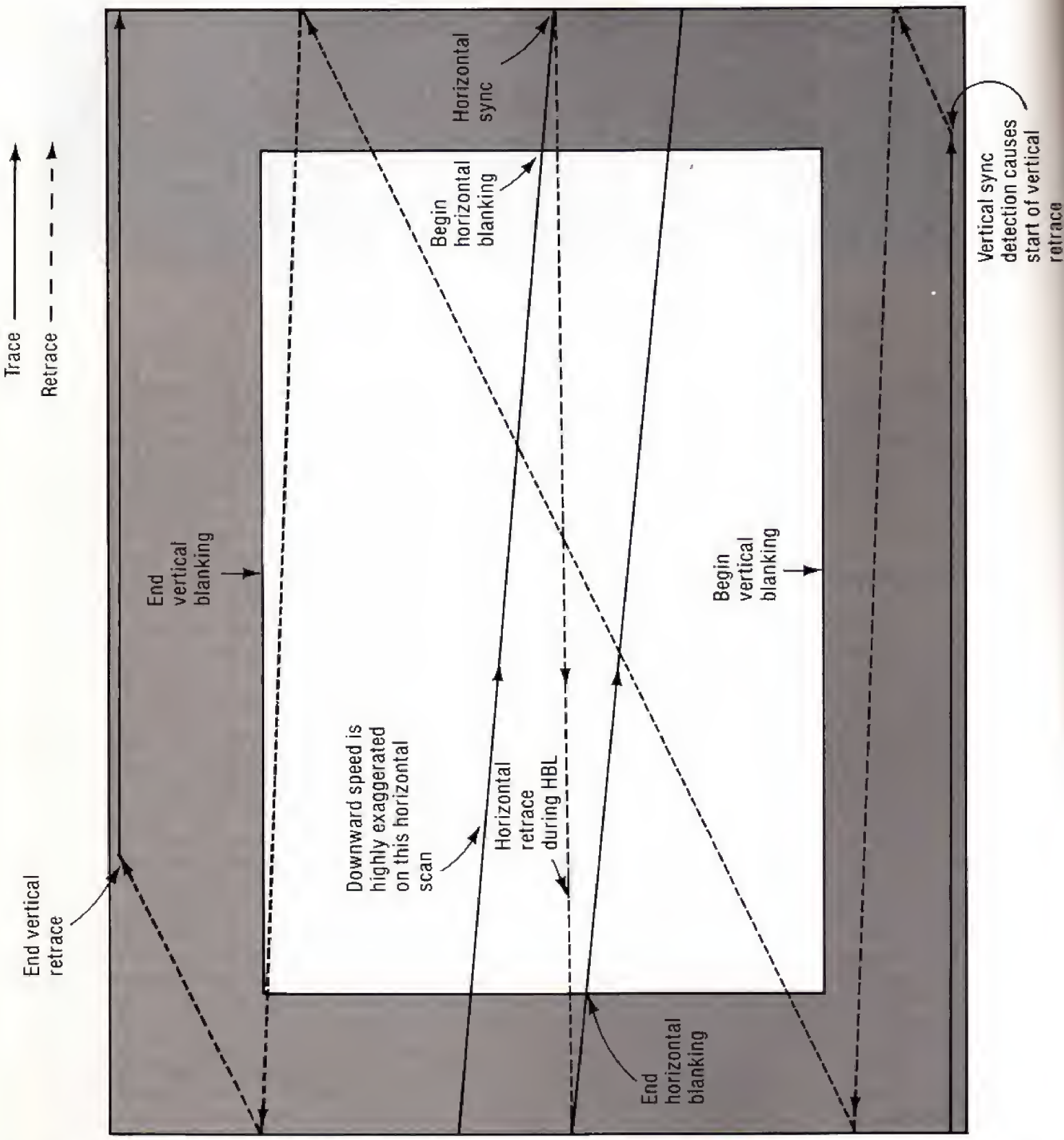


Figure 8.5 Blanking During Television Scanning Causes the Black Margin Around the Apple Display.

In Revision 1, the vertical sync equation was changed to V4 • V3 • V2 • V1′ • V0′ • VC′ • (H5 + H4). Adding V0′ • VC′ to the equation reduces vertical sync to four horizontal PERIODS. Adding (H5 + H4) puts little positive pulses or **serrations** in the vertical sync, so a negative edge occurs right where horizontal sync would normally cause a negative edge to occur. These serrations are a feature of an NTSC standard signal, and Apple probably added them in Revision 1 to improve horizontal stability. The Revision 1 vertical sync is accurately illustrated in Figure 8.2 and in lines 224-227 of Figure 5.9. The horizontal sync in Figure 5.9 is not accurate for Revision 1, because it shows only four #s. Eight #s would accurately portray the eight cycle pulse width of horizontal sync in Apples prior to Revision 7.

In Revision 7, the horizontal sync equation was changed from HBL • H3 to HBL • H3 • H2′. This reduced the pulse width of the horizontal sync from eight cycles to four cycles as is shown in the lower left corner of Figure 8.2. A change was also necessary to the COLOR BURST gating logic so it would still be positioned correctly. The COLOR BURST equation was changed from COLOR REFERENCE′ • HBL • H4 • H2′ to COLOR REFERENCE′ • HBL • H3 • H2. In either equation, the COLOR BURST is a sample of fourteen COLOR REFERENCE cycles occuring immediately after the horizontal sync.

Prior to Revision 7, the horizontal sync held the SYNC signal low during the first eight cycles of the vertical sync period. When the horizontal sync pulse was reduced in width in Revision 7, a four cycle gap was created in the vertical sync. This would be illustrated in Figure 5.9 by changing the fifth through eighth #s in lines 224-227 to +s. This effectively makes the horizontal serrations into double pulses and probably detracts from sync stability. Some digital video monitors which interpret every negative edge on the SYNC signal as a horizontal sync pulse would not be able to operate with this double pulse. This situation was corrected in the RFI Revision.

The author isn't certain why the horizontal sync pulse width was reduced in Revision 7, but he does know of one effect. It eliminates the little switching spike that causes the vertical black line in the left margin of older Apples. The spike is present on the SYNC signal at C13-8 of the older Apples. It is there because HBL and H3 do not switch simultaneously. Since HBL is generated from the video scanner outputs, it goes low a few nanoseconds after H3 goes high at start display time of every scan. The unwanted spike could have been eliminated by routing H3 through a couple of LSTTL devices before connection to pins 10 and 11 of C13. In Revision 7, H2′ became part of the horizontal sync logic. Its propagation delay through Q7 (A14 in RFI Apples) is sufficient to mask the unwanted spike at C13.

A new IC was added to the motherboard at A14 in Revision 8 or Revision 9. The purpose of A14 was to add logic gating to the COLOR BURST generation circuitry to cure a problem in the Color Burst Killer. The Q6 Color Burst Killer was added to the Apple in Revision 1 to eliminate color from the screen characters in text mode. This is good, but Q6 lets a small residual amount of the COLOR BURST get through to the video signal. Some televisions and monitors occasionally lock up on this residual COLOR BURST and create an unreadable display of white text with colored ghosts. The Revision 8/9 change causes the COLOR BURST to be gated off in TEXT mode using LSTTL logic before the signal ever gets to Q6. This cures the unreadable text problem.

The RFI Revision eliminated the double serration that was introduced to the vertical sync in Revision 7. It also eliminated transistor Q7, which is present only in Revisions 7, 8, and 9. The double serration was removed by changing the (H5 + H4) portion of the vertical sync equation to (H5 + H4 + H3). Q7 was eliminated by performing its functions in A14. The new gating logic caused an inversion in the COLOR REFERENCE sample used for generation of COLOR BURST. This was countered by distributing COLOR REFERENCE′ to the video generator rather than the COLOR REFERENCE signal of earlier Apples.

Video scanner logic gating is also used in switching between GRAPHICS and TEXT in the Apple's MIXED screen mode. In MIXED mode, V2 • V4 identifies TEXT time. It is true during the last 32 displayed horizontal PERIODS and during the last 38 undisplayed horizontal PERIODS of VBL. This means that in MIXED mode the Apple switches to GRAPHICS then back to TEXT during VBL, but it is not significant because the screen is blank during VBL. Any time the screen mode is switched between GRAPHICS and TEXT, including a MIXED mode switching, the switchover is delayed until the third time RAS′ rises after the switch logic changes. The purpose of this is to delay the MIXED mode switch from HIRES to TEXT until the last HIRES pattern is shifted out. The fact that RAS′ is used for synchronizing mode switching is coincidental. There are four RAS′ clocked flip-flops not used by the

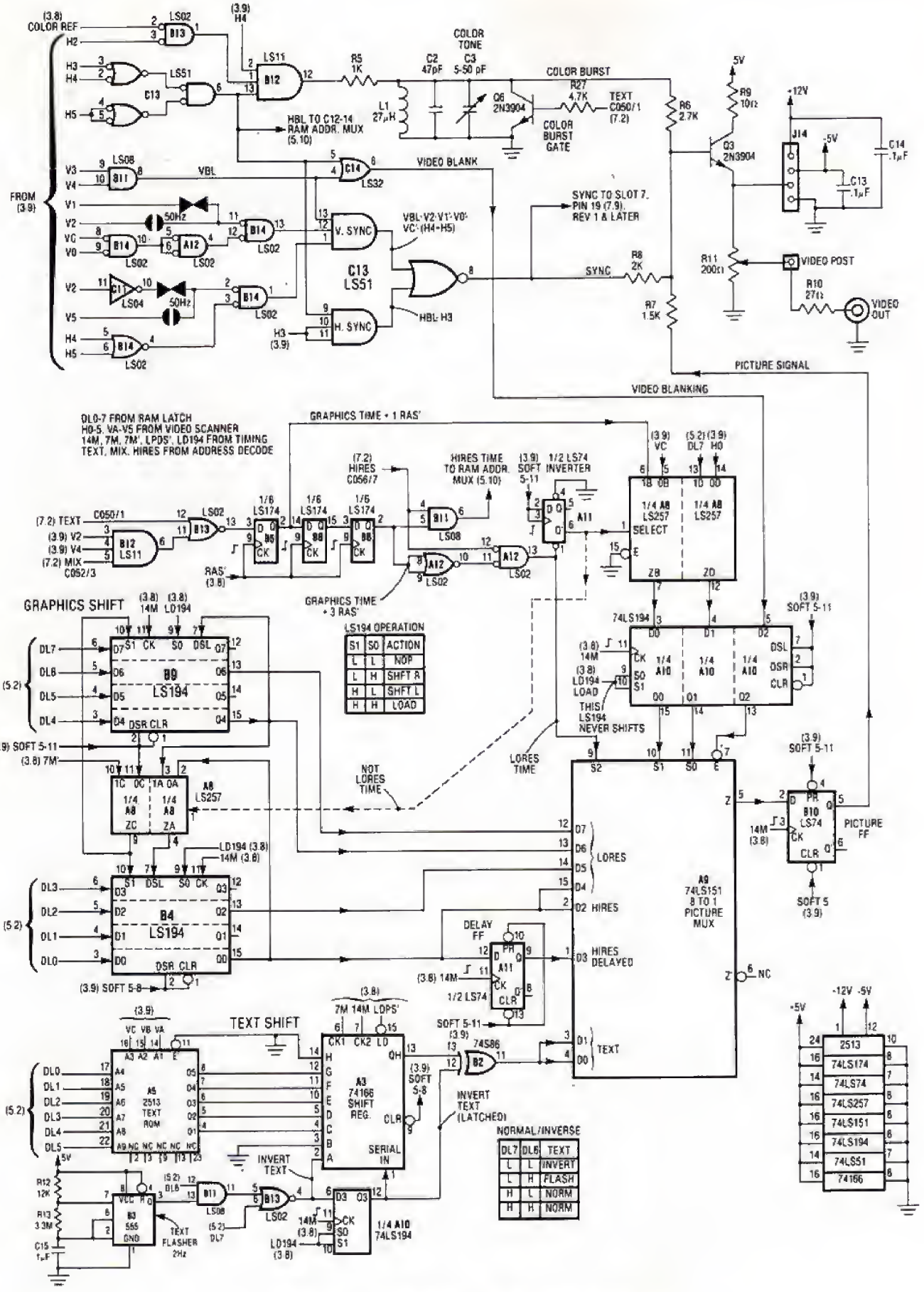


Figure 8.6 Schematic: Video Generation.

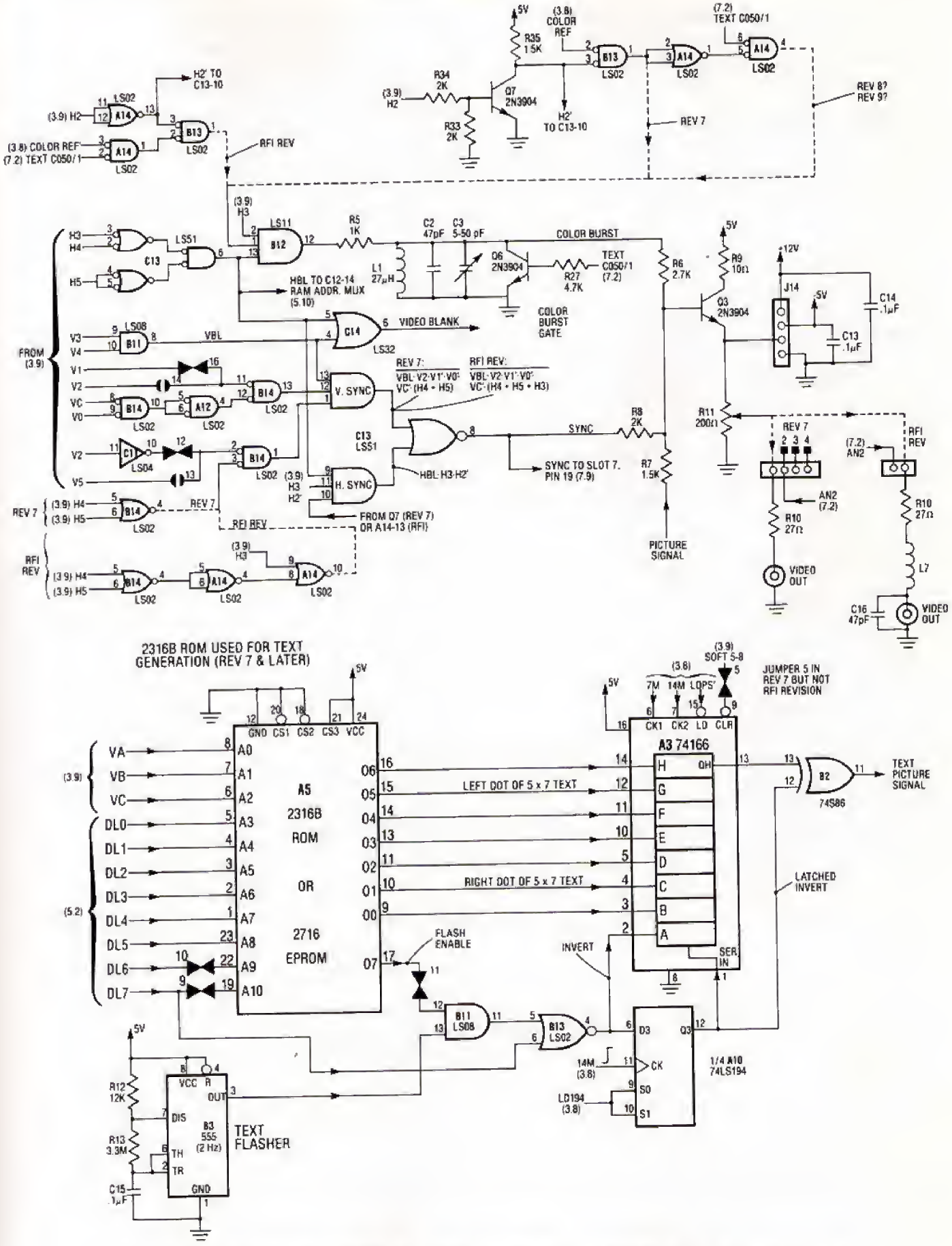


Figure 8.7 Schematic: Revision-7 and RFI Version Changes to Video Generation.

RAM data latch at B5 and B8. Three of these flip-flops are connected as a shift register to produce the mode switching delay required by the Apple design.

Eurapple Scanning

The discussions have been assuming the Apple would be driving an NTSC standard TV, just as the original Apple designer assumed it would be. In the Revision-1 board, Apple changed this by adding "Eurapple" jumpers which change the scanning features of the Apple to be compatible with European TV. This is accomplished by adding 50 horizontal scans to the video scanner and by shifting the vertical sync to maintain approximately equal black margins at the top and bottom of the screen. The horizontal scanning is not affected by the Eurapple jumpers.

European television has 625 horizontal scans in two interlaced fields as compared to NTSC standard 525 scans. Also, European TV scans vertically about 50 times per second as opposed to NTSC standard 60 vertical scans per seconds. The Eurapple jumpers in the Apple cause the vertical portion of the video scanner to preset to 011001000, fifty less than the normal 011111010. This adds 50 scans to the normal 262 for 312 horizontal scans in a Eurapple. All 50 of these scans are added to VBL so that VBL is 120 scans long instead of 70.

In the video generator, Eurapple jumpers change the vertical sync equation to V5′ • V4 • V3 • V2′ • V0′ • VC′ • (H5 + H4). The (H5 + H4) portion is ($\overline{H5}$ + H4 + H3) in RFI Revision Apples and this horizontal logic adds serrations to the vertical sync identically to 60 Hz configured Apples. In Eurapple scanning, the vertical sync lasts for four horizontal PERIODS, just as in American scanning. These are horizontal PERIODS 73, 74, 75, and 76 of VBL in Eurapple scanning. By way of comparison, normal Apples have 36 PERIODS in VBL up through vertical sync and 34 PERIODS in VBL afterwards. The Eurapple has 76 PERIODS in VBL up through vertical sync and 44 PERIODS in VBL afterwards. The critical states of VA-V5 in both NTSC standard Apples and Eurapples are summarized in Table 8.1.

The Eurapple jumpers give an Apple scanning compatibility with European television, but not color signal compatibility. Also as part of Revision 1, Apple brought television SYNC and COLOR REFERENCE to previously unused pins 19 and 35 of peripheral Slot 7. One can install a Eurocolor card in Slot 7 which outputs a video signal compatible with PAL or SECAM system televisions, both of which are found in Europe. The 14M crystal must also be changed to a value equal to four times the frequency of the chrominance signal used in a given system.

VIDEO GENERATION TIMING SIGNALS

Video generation timing is based on several signals developed in the timing generator with **LDPS′** and **LD194** defining video output cycles. These and other timing signals are shown in Figure 8.8 along with several examples of video output in the three modes. Actually, LDPS′ and LD194 are used to define the same time period, with LDPS′ being used when an active low signal is required. In the video generator, TEXT and GRAPHIC patterns are loaded by 14M rising when LDPS′ is low, 7M is low, and LD194 is high. The patterns are shifted the rest of the time.

Table 8.1 Eurapple/NTSC Differences.

	START VBL **V543210CBA**	VERTICAL SYNC **V543210CBA**	PRESET ON OVERFLOW **V543210CBA**	VERTICAL SYNC **V543210CBA**	END VBL **V543210CBA**
NTSC	111000000	1111000XX	011111010	------	100000000
EURAPPLE	111000000	-------	011001000	0110100XX	100000000

A video output cycle lasts .98 microseconds, just like a normal 6502 cycle. In this period of time there are 14 cycles of 14M, 7 cycles of 7M, and 3.5 cycles of COLOR REFERENCE. During a video cycle, one byte of data from memory is processed for video output, so there are 40 video cycles per display line. In HIRES and TEXT modes, seven dot positions are scanned per video cycle and the output is therefore shifted every other 14M rising. More specifically, HIRES and TEXT loading and shifting are clocked by 14M rising when 7M is low. Note that this is not the same as 7M rising because of propagation delay of 7M after 14M. The LORES shift clock is 14M rising.

Video output timing is related to RAM timing and video scanner timing, naturally. Data from the scanner access is latched at the RAM output by RAS' rising during PHASE 1. Then 14M rises during LD194 before the next RAS', loading GRAPHICS or TEXT patterns at the video generator. Then the video scanner increments just before shifting for this video cycle begins. This means that the video output lags the scanner address by approximately one video cycle. LORES picture selection is based on scanner outputs **H0** and **VC** as they were for most of the previous video cycle. In LORES, H0 and VC are latched every LD194 and held for the entire following video cycle. Therefore even though H0 may be high for most of the first video cycle shown in Figure 8.8, LORES picture selection is based on H0 low as was the case when 14M went high during LD194. Because of this latching action, Figure 8.8 illustrates video processing of an even screen memory address followed by an odd screen memory address.

The saving of H0 and VC during LD194 is accomplished in a 4-bit latch at A10. Besides H0 and VC, other signals which change during the video cycle are synchronized to the video cycle by A10 and used in PICTURE signal selection. These signals are GRAPHICS TIME, DL7, the VIDEO BLANKING gate, and the INVERT TEXT signal.

A fact of life, when dealing with a 1 MHz video cycle and a 3.5 MHz COLOR REFERENCE, is that there are 3.5 COLOR REFERENCE cycles per video cycle. The result is that the COLOR REFERENCE begins every cycle 180 degrees out of phase from the way it was on the previous video cycle. Timing is set up so that COLOR REFERENCE is always low at the beginning of even video cycles (cycles which process data stored at even memory addresses). This is the purpose of the long cycle, to maintain this relationship in spite of an odd number of cycles (65) in a horizontal scan. Still, the fact remains that identical dot patterns produce colors 180 degrees out of phase in adjacent video cycles. In LORES this is compensated for by switching the phase of the video pattern based on H0. There is no compensation in HIRES, so the programmer must process even memory locations different than odd memory locations when producing colored HIRES displays. As an example, to produce a short green line, 00101010 is stored at an even address or 01010101 is stored at an odd address.

TEXT VIDEO OUTPUT

Text processing is very straightforward in the Apple. The pattern from the TEXT ROM is loaded to A3 when 14M rises during LDPS' and it is shifted to an exclusive-OR gate. The exclusive-OR acts like a selectable inverter, feeding the pattern to the picture MUX when INVERT TEXT is low and feeding the complement of the pattern to the picture MUX when INVERT TEXT is high. INVERT TEXT is latched at the same time the dot pattern is loaded, so it will not switch in the middle of a TEXT output cycle. As has been mentioned before, the inversion of video is dependent on DL6 and DL7 of the text ASCII and the current state of the 2 Hz text flasher when DL6 is high and DL7 is low.

At the same time the dot pattern is loaded, the INVERT TEXT signal is loaded into bit A of the TEXT shifter. Then, during the TEXT shift, INVERT TEXT determines whether the serial input to the shifter is high or low. This is mentioned only because it is true and in spite of the fact that it serves no function. It makes a schematic diagram investigator very nervous to say something like this. Why would they bother connecting wires that serve no function? Be that as it may, whatever is loaded or shifted into bit A of the TEXT shifter never is shifted to bit H before the shifter is reloaded by 14M rising during LDPS'.

In the original Apple II, the TEXT ROM was a General Instrument 2513 upper case character generator ROM. It is the contents of this character generator, illustrated in Figure 8.4, which determine the screen characters that are available in the Apple TEXT mode. The 2513 contains 512 5-bit words arranged into 64 eight word characters. In Revision 7, the wiring of the TEXT ROM socket was changed to accommodate the 8 x 2048 bit 2316 ROM. Apple puts exactly the same upper case character set into the 2316, but it is set up so you can put more character sets in your own 2716 EPROM and substitute it for the supplied 2316 ROM. That was a pretty neat thing to do. In any case, the modification did nothing to alter the TEXT load/shift cycle.

Figure 8.8 shows the output of the bottom dot patterns of an ampersand and an inverse ampersand to the first two character positions at the left of the screen. At the far left of the TEXT waveforms in Figure 8.8, H5 through H0 of the video scanner are at 011000 meaning HBL and VIDEO BLANKING have just gone low. The RAM Latch contains $A6, ASCII for a normal ampersand, driven out of RAM by the scanner access. The combination of VA, VB, and VC all high and DL5-DL0 equal to 100110 drives 01101 out of the 2513 TEXT ROM or 00011010 out of the 2316 ROM used in Revision 7 and later Apples.

About 390 nanoseconds after the video data becomes valid at the RAM Latch, 14M rises with LDPS' and 7M low, loading the dot pattern to the shifter. Leading and trailing zeroes are loaded on either side of the five-dot pattern, although this can be changed with the TEXT ROM in later Apples.

The access time of the TEXT ROM in the Apple must be under 390 nanoseconds. The typical time of the 2513 is 250 nanoseconds, but the guaranteed maximum is 450 nanoseconds, so Apple was shading their margin of error with this ROM. It is possible in later Apples that they use 350 nanosecond 2316 ROMs, and certainly you should use 350 nanosecond EPROM in your custom TEXT ROM. Of course, if you have a 450 nanosecond EPROM that gives a good display in a warm or cold room at power up and after two hours of operation, use it. There is no law against a 450 nanosecond memory which can be accessed in 390 nanoseconds.

At the same time the dot pattern is loaded, the picture MUX ENABLE' signal is latched low, and the INVERT TEXT signal is latched low. The former event shows the logic behind synchronizing VIDEO BLANKING to LD194. The blanking becomes inactive at the exact time dot patterns become ready for output at the left side of the screen. The latter event occurs under control of DL7. When DL7 is high at LD194, INVERT TEXT is latched low for the whole video cycle.

The QH output of the TEXT shifter is felt at the D0 and D1 inputs to the picture MUX shortly after the dot pattern is loaded. Either D0 or D1 will be selected by the MUX in TEXT mode, because S2 (LORES TIME) and S1 (GRAPHICS TIME + 1 RAS' + 1 LD194) are both low. Therefore, QH is clocked to the **picture flip-flop** at B10 by the first 14M rising after the pattern is loaded. It is the output of this flip-flop that is added to SYNC and COLOR REFERENCE to make up the VIDEO signal. As the dot pattern is shifted through the TEXT shifter, the picture flip-flop will follow QH, lagging it by approximately one 14M period.

The picture flip-flop ensures that all types of Apple PICTURE signal are begun at identical points on the screen. For example, if it takes TEXT patterns longer than LORES patterns to be propagated through the picture MUX, it doesn't matter. All types of video are clocked by 14M rising on the picture flip-flop. An interesting point about the picture flip-flop is this: its low-to-high propagation delay (time after 14M rises for pin 5 to go from low to high) is 13 nanoseconds typical, but its high-to-low propagation delay is 25 nanoseconds typical. Since the PICTURE signal rises more quickly than it drops, the white portions tend to be a little wider than otherwise equal black portions. For example, a white dot of a normal TEXT pattern would be about 24 nanoseconds longer than the black dot of an inverted TEXT pattern. You can easily prove this to yourself by displaying some normal TEXT next to some inverted TEXT on a high frequency response video monitor. The "black dots" are not as wide as the white dots.

The shifting of the normal ampersand progresses until the value that was loaded to QB has been shifted to QH. The next TEXT clockpulse will occur during LDPS' low, so it will cause a load rather than a shift. At the time of this loading clock, $26 (inverse ampersand) will have been valid at the RAM Latch for 390 nanoseconds. The identical pattern is driven out of ROM that the normal ampersand caused since the ROM address inputs are the same. When the loading clock rises, the pattern is loaded and INVERT TEXT is latched high (because DL7 and DL6 are low). This pattern is shifted to the exclusive-OR just as the previous one was, but because INVERT TEXT is low, the inversion of the pattern is propagated through the exclusive-OR.

If ASCII is driven out of RAM with DL7 low and DL6 high, the INVERT TEXT signal is latched high or low depending on the 2 Hz flasher. This results in the flash coded screen positions alternating between INVERSE and NORMAL four times a second. This is true even if the ASCII is $60, code for a flashing space. In fact, a flashing character is the way the monitor's RDKEY routine forms the flashing cursor while waiting for a key to be pressed. More often than not, this is a flashing space.

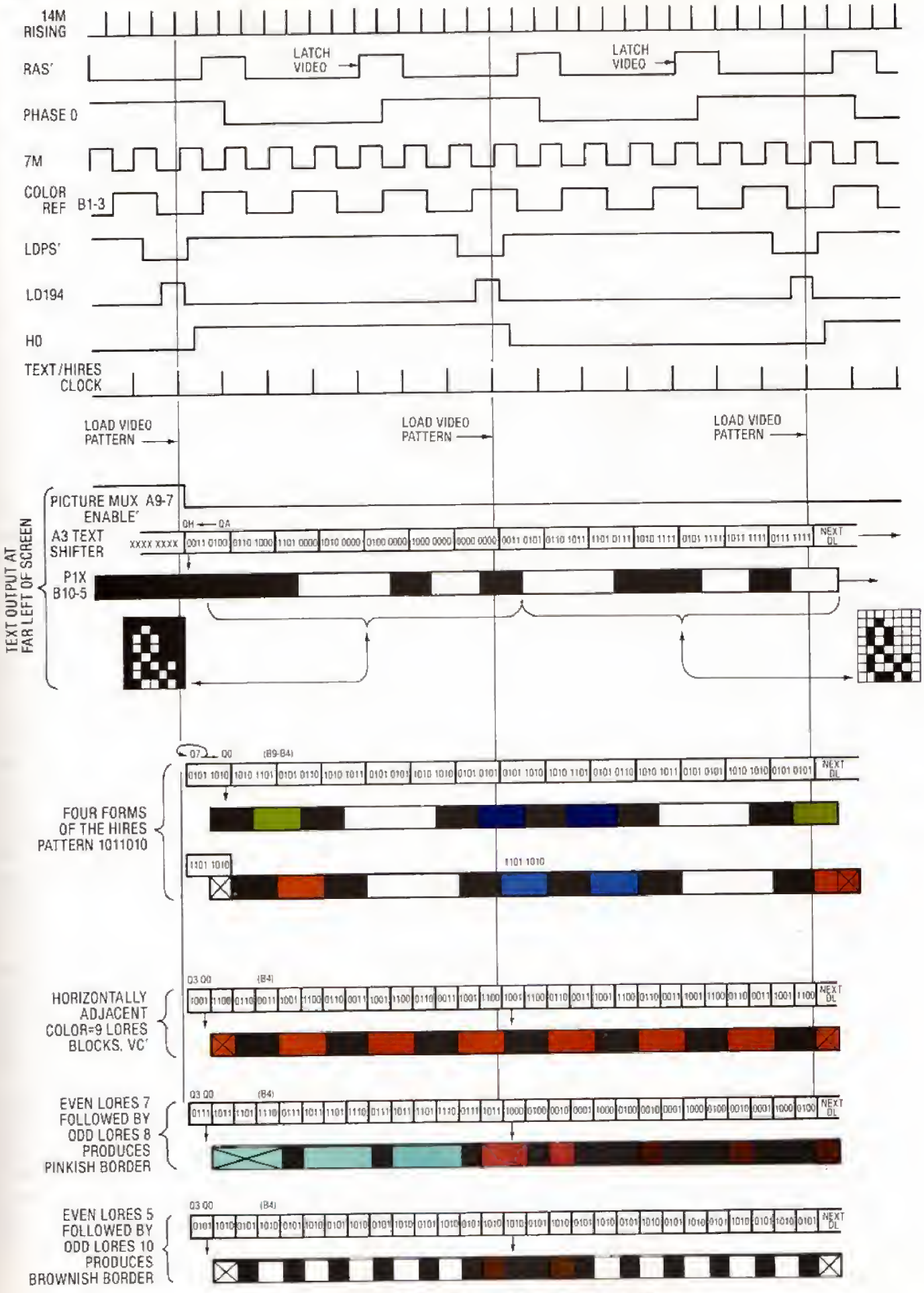


Figure 8.8 Video Output Examples.

HIRES GRAPHICS OUTPUT

The HIRES output has similarities to the TEXT output. Both the HIRES and TEXT PICTURE signals are straightforward serial outputs of 7-bit dot patterns. For this reason, it is possible to draw text using HIRES graphics with the same 5 by 7 dot patterns generated by the 2513 ROM. The HIRES TEXT will have coloring however, because the COLOR BURST is enabled. Besides the COLOR BURST, important differences are:

1. HIRES patterns are stored directly in RAM, seven dots per byte. TEXT ASCII is stored in RAM, one character per byte, and TEXT patterns are stored in ROM, eight rows of seven dots per ASCII code.
2. The eighth-bit of the HIRES pattern, DL7, is not output to the screen. Rather its high/low state is latched at load pattern time, and the entire HIRES pattern is delayed by a 14M period if DL7 was latched high.

Figure 8.8 shows the same 7-dot HIRES pattern output in four different phase relationships with the COLOR REFERENCE. The result is the same picture pattern colored four different ways. The four different colorings are produced by storing the pattern 1011010 at adjacent memory locations with D7 reset and at adjacent memory locations with D7 set. We'll look at these output sequences, but first we need to see how the video generator is configured in HIRES mode.

Since the Apple is in HIRES mode, the following logical setup exists:

PIX MUX S2 = LORES TIME	- low
PIX MUX S1 = GRAPHICS TIME	- high
PIX MUX S0	- DL7 latched during LD194
GRAPHICS SHIFT MODE	- 8-bit serial
GRAPHICS LOAD/SHIFT CLOCK	- 14M rising while 7M low
HIRES TIME	- high

The setup gets more complicated during switching between modes, but we assume the Apple is not currently switching modes. The burden of establishing this configuration lies primarily with the LS257 at A8. Its select input is tied to LORES TIME and the logic of configuration goes like this: if it is LORES TIME, configure for LORES; if it is not LORES TIME, configure the GRAPHICS shifter for HIRES and select HIRES or TEXT based on the signal, GRAPHICS TIME + 1 RAS' latched by LD194. The HIRES dot pattern is shifted through Q0 of the GRAPHICS shifter to D2 of the picture MUX and through a delay flip-flop to D3 of the picture MUX. D2/D3 is selected by the picture MUX, because S1 and S2 are low in the HIRES configuration. DL7, latched by LD194, selects between HIRES (D2) and HIRES delayed (D3).

The top HIRES pattern in Figure 8.8 is formed by 01011010 being driven out of RAM by scanner access to an even address followed by an odd address. Just as in TEXT shifting, the pattern stored at LD194 is shifted every other 14M rising, and the pattern is clocked to the PICTURE signal by the picture flip-flop. As the leading 010 is shifted out, the PICTURE signal starts black, swings white, then swings black, creating a square wave identical in frequency to the COLOR REFERENCE, 3.58 MHz. The television will pass this signal through its chrominance amplifier and phase compare it to the COLOR REFERENCE which it has reconstructed from the COLOR BURST. The result of this phase comparison will be color signals resulting in a HIRES green coloring of the dot on the screen. Compare the green dot position to the COLOR REFERENCE. Any PICTURE signal which goes white then black in this relationship with the COLOR REFERENCE will produce green coloring on the television.

Shifting along we come to the two adjacent white dots. These dots are produced by a signal that goes white then black at 1.79 MHz, one half the frequency of the COLOR REFERENCE. Very little of this signal can get through the chrominance amplifier. The result is absence of color signals and subsequent white illumination. Anywhere on the screen, COLOR BURST or no COLOR BURST, bringing the PICTURE signal to the white level for a full period of COLOR REFERENCE will result in a white picture.

The white pulse is followed by a violet pulse, identical in pulse width to the green pulse but occuring in opposite phase relationship when compared to COLOR REFERENCE. HIRES green and HIRES violet complement each other; that is, they are 180 degrees out of phase.

When the identical 1011010 pattern is loaded from an odd address and shifted out, the PICTURE signal swings from white to black and back just as it did when the pattern was loaded from an even address. The COLOR REFERENCE, however is 180 degrees out of phase from the way it was during the adjacent video cycle. The coloring of the screen

dots is therefore the complement of the pattern output in the adjacent video cycle. Green becomes violet, violet becomes green, and white remains white.

The nature of HIRES green and violet video should now be fairly clear. The HIRES dot is exactly the width of half a COLOR REFERENCE period. Visualize the COLOR REFERENCE alternating up and down as the beam crosses the screen, starting with COLOR REFERENCE high as the display begins. If even position dots are turned on, they coincide with COLOR REFERENCE high and are violet. Similarly odd dots are green, and two or more adjacent dots are white. This was the extent of HIRES patterns in the Revision 0 Apple II: 280 programmable dots per scan with violet, green, or white coloring. In Revision 1, the **D7 delay option** was added which enables the delay of each 7-dot pattern by one fourth of a COLOR REFERENCE period.

The output of the same 1011010 dot pattern, stored at adjacent memory locations with D7 set is also shown in Figure 8.8. The picture pattern is identical, but since it is delayed by one 14M period, a new pair of complementary colors are produced. The new colors are a result of the fact that delayed dots have a different phase relationship with COLOR REFERENCE than undelayed dots. The mechanization of the delay is straightforward. DL7 high, latched at LD194, causes the picture MUX to select delayed Q0 for the HIRES PICTURE signal.

Delaying the 7-dot patterns was a tricky way of sprucing up the HIRES display; violet and green got a little old. Programming HIRES video became even more of an abstract art, however, with each group of seven dots having a color and position characteristic. Add this to the facts that alternating dot positions produce different colors, screen memory addresses are difficult to compute, and delayed HIRES patterns interfere with adjacent undelayed HIRES patterns; and you've got real programming complexity.

Delayed and undelayed HIRES patterns interfere with each other? They sure do, but before we get into that, let's summarize the characteristics of HIRES video based on the discussion to this point. First, there are 192 horizontal rows of dots. In each row, 280 dots (40 x 7) may be turned on or off, but since each group of seven dots may be shifted right half a dot width, there are 560 dot positions in a row. Color depends on position, and there are 140 violet positions, 140 green positions, 140 blue positions, and 140 orange positions. Any two adjacent dots turned on will be white. We will see shortly that there are really only 139 orange positions. This is because an orange dot on the far right of the screen will be cut off by HBL to make it dark brown. Also, two adjacent delayed dots at the far right will be light blue green, not white, and the far left blue dot will be violet or blue depending on the contents of the last byte of memory scanned during the preceding HBL.

If color is of no concern, there is uninhibited programming of a 192 x 280 matrix of dots. With restrictions, this becomes 192 x 560. The main restriction is that a delayed dot cannot be in the same 7-bit group as an undelayed dot. For example, you can draw a slanted straight line close to the vertical with 192 x 560 resolution. You can also draw a very nice vertically oriented parabola with 560-dot horizontal resolution at the portions where the slope is more vertical. The other restriction on 560-dot resolution is the interference at the boundary between adjacent delayed and undelayed patterns which will be detailed shortly.

For coherent violet, green, blue, and orange colored displays, there is 192 x 140 dot resolution as long as certain pairs of colors don't get too close to each other. Anytime you plot a green dot in the same 7-dot pattern as an orange dot, that orange dot turns to green, because D7 had to be reset in that memory location to plot the green dot. Similar considerations exist for mixing blue and violet. Any two adjacent dots will always be white.

Why do colored HIRES objects appear solid if every other dot is turned off? Shouldn't a violet object appear to be numerous horizontal rows of dots rather than solid lines? The object appears solid vertically because the horizontal scans are so close together. If you look at the violet object up close, you will see that it appears to be numerous horizontal lines. The reason that the object appears solid horizontally is that a multichannel color television is not capable of turning its beam intensity on and off cleanly at 3.58 MHz. Instead, the dots are blurred into continuous horizontal lines. For the same reason, Apple text is fairly blurred when displayed on a television set.

Now if you inject the same VIDEO signal into a high frequency response video monitor, you will clearly see the black spots between the dots in the lines that were violet on the television. It is very educational to compare all forms of Apple video to simultaneous displays on a television and high frequency monitor. HIRES and LORES graphics modes use the "slowness" of a television to display

colored solids, but the monitor shows the dot patterns which produce the solids. The "slowness" of a television is why computers that are designed to output TEXT to a television have a display of 40 TEXT characters or less. It is also for this reason that if you use an 80-column card with the Apple, you must support it with a high frequency response video monitor.

Interference Between Adjacent Delayed and Undelayed HIRES Patterns

The 7-dot HIRES patterns fit snugly together if the adjacent patterns are all delayed or undelayed, but problems can be caused when they are mixed together. This can be seen by comparing the delayed and undelayed HIRES patterns of Figure 8.8.

The sequence of starting a delayed pattern shift goes like this:

14M 1: Load pattern and latch DL7 high.
14M 2: Shift delayed Q0 to picture and load delay flip-flop with first dot of pattern.
14M 3: Shift first dot to picture.

The point is that delayed Q0 is selected for output before the delay flip-flop has loaded the state of the first dot position. What does the delay flip-flop contain at the first output clock of this video cycle? It contains the state of the last dot position from the previous 7-dot pattern. This means that the first 14M period of a delayed pattern to the screen is controlled by the last dot of the previous pattern.

Now think about the last 14M period of a delayed pattern. It is not shifted to the PICTURE signal before the next video cycle begins. If the next video pattern is also delayed, it will begin by completing the current dot pattern. If the next video pattern is not delayed, it will cut off the tail of the current pattern. The result is that continuous undelayed or delayed patterns fit snugly together, but a delayed pattern extends the last dot of a preceding undelayed pattern by a 14M period, and an undelayed pattern cuts the last dot of a preceding delayed pattern in half. Cutting off or extending a dot has the effect of slightly changing the dot pattern and, more noticeably, changing the coloring of the border dots. As a result the HIRES programmer has one more thing that affects color to educate himself about and take into account. On the plus side, the programmer can draw vertical lines at pattern borders in nine colors that are not otherwise available in HIRES. He does this simply by turning on a right hand dot then extending or cutting it off via DL7 of the following pattern. In some instances, no dots need be turned on in the following pattern.

Figure 8.9 is a photograph illustrating the generation of LORES colors at borders between delayed and undelayed 7-dot HIRES patterns. The program which generated this display is listed in Figure 3.10. The mixed LORES/HIRES display is created by switching screen modes in a 8515 cycle loop as is discussed in an Application Note in Chapter 3. As the photo shows, any LORES color except dark blue-green (4) can be produced at a limited number of screen positions. LORES colors 3, 6, C, and 9 are natural equivalents of the HIRES colors. LORES colors 7 and 2 can be produced at even/odd memory

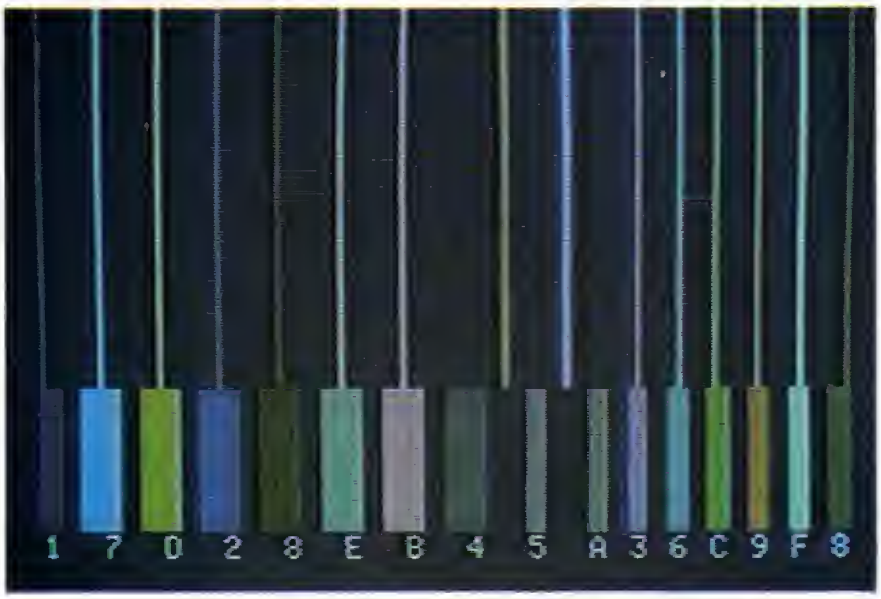


Figure 8.9 The Output of the Screen Splitting Program (Figure 3.10).

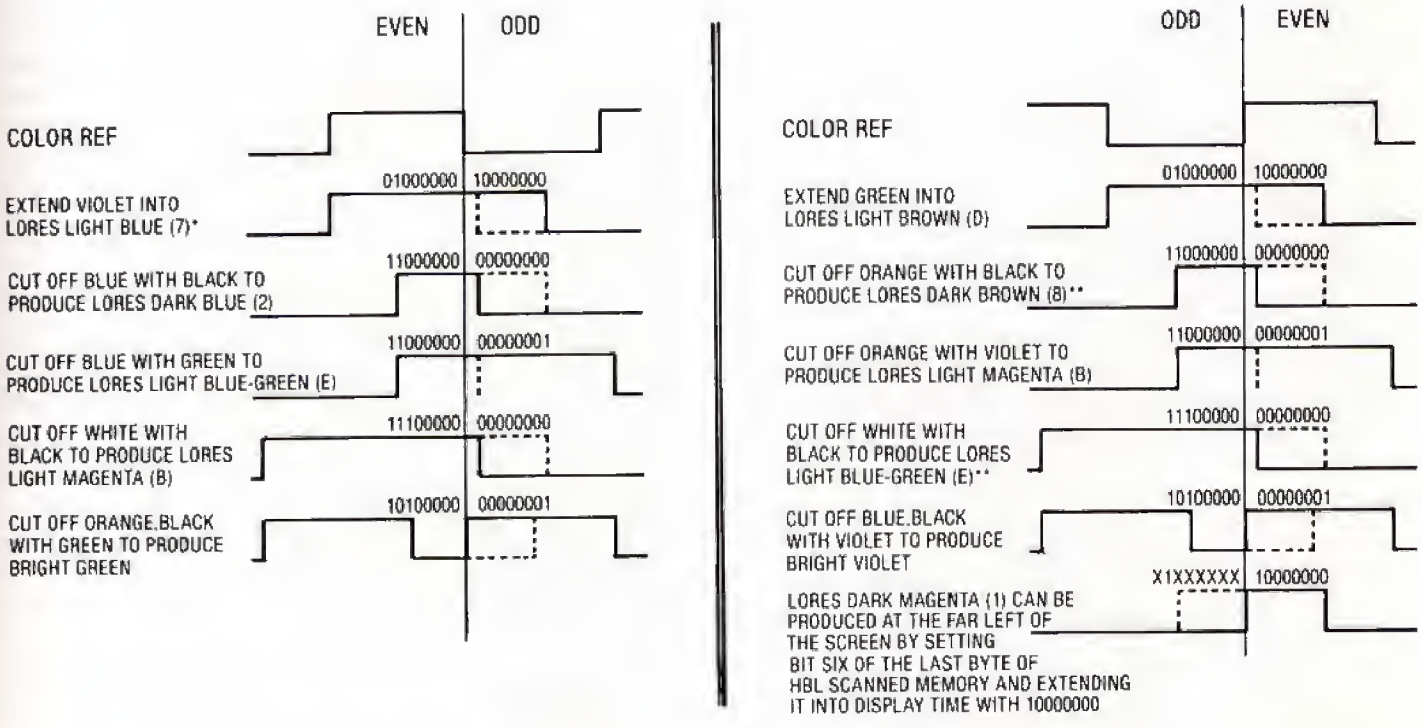
addressing borders. Colors D and 8 can be produced at odd/even borders. Colors B and E can be produced at odd/even or even/odd borders. LORES color 7 can also be produced at the far left of the HIRES screen, and LORES color E can also be produced at the far right of the HIRES screen. Color 1 can be produced only at the far left of the HIRES screen. Finally, orange HIRES dots at the right side of the screen are LORES dark brown (8).

Figure 8.10 shows some patterns created by mixing delayed and undelayed dot patterns. Comparison to Figure 8.11 in the following LORES section will show that these patterns have the same phase relationship with COLOR REFERENCE as various LORES colors. The exception is the case where a trailing 1-0 delayed pattern is cut off by an undelayed 1-0 pattern. This creates a green or violet with a different shade than any HIRES or LORES color. The general rule of all these HIRES interference patterns is that delayed extends undelayed, and undelayed cuts off delayed.

The extreme left and right dot positions are special cases for delayed patterns. The VIDEO BLANKING gate is precisely aligned to undelayed video, and this doesn't change if the first and last patterns are delayed. Therefore, a delayed pattern in the first video cycle of a display period extends bit D6 of the last byte of HBL scanned memory into the display. In other words, half of an undisplayed memory bit gets displayed. On the right side of the screen the last delayed dot is in an orange dot position. This dot is cut off by the VIDEO BLANKING gate so the dot is dark brown rather than orange. Similarly, a delayed white pair at the far right is cut off to make it light blue-green.

The right side cut off is no great problem to deal with, because it is predictable. The left side display of a HBL scanned dot can be more confusing. The following Applesoft program is an example of what can happen:

```
10 HGR : HCOLOR=5 : HPLOT 5,159
   TO 5,0 TO 279,0 TO 279,159 :
   REM HCOLOR 5 IS ORANGE
```



* LORES LIGHT BLUE (7) MAY ALSO BE PRODUCED AT THE FAR LEFT OF THE SCREEN BY SETTING BIT SIX OF THE LAST BYTE OF HBL SCANNED MEMORY AND EXTENDING IT INTO DISPLAY TIME WITH 10000001

** THIS IS IDENTICAL TO THE CUT OFF OF A DELAYED PATTERN BY THE VIDEO BLANKING GATE AT THE RIGHT SIDE OF THE SCREEN.

Figure 8.10 HIRES Interference at Pattern Borders.

What do you think will be drawn and in what color? If you think the four points (5,159), (5,0), (279,0), and (279,159) will be connected by three orange lines, then you're wrong. What happens is that (5,159), (5,0), and (279,0) are connected by two orange lines. (279,0) and (279,159) are connected by a dark brown line, because VIDEO BLANKING cuts the orange dots off and makes them brown. Then (0,191) is connected to (0,64) by a dark magenta line. This last line is there because having the far left pattern delayed extends the last dot of the last byte of HBL scanned memory into the display area. The last byte of HBL scanned memory for lines 64 through 191 is the last byte of display scanned memory for lines 0 through 127. Therefore, the last display dot of the top two thirds of the screen is extended to the first display dot of the bottom two thirds by left hand delay patterns. Dot (279,0) is extended to dot (0,64); dot (279,1) is extended to dot (0,65) and so on.

Here's another line to add to the above program:

```
20 FOR A = 8319 TO 16383 STEP
   128 : POKE A, 64 : NEXT
```

This line completes the magenta line at the left side up to (0,0). Did you catch those numbers? If BASIC used hexadecimal addresses, the line would read:

```
20 FOR A = $207F TO $3FFF STEP
   $80 : POKE A, $40 : NEXT
```

Those addresses are the last byte of the eight byte unused memory segments spread throughout HIRES memory. They are the last bytes scanned before display time in the top third of the screen. They are supposed to be undisplayed, but line 20 causes dots to be displayed by POKING values to them. Here's another example. Enter these keystrokes:

```
HGR          ;HIRES mode, clear
             ;screen
CALL-151     ;enter monitor
2000:80      ;upper left pattern
             ;delayed
207F:40      ;set last dot of
             ;undisplayed byte
```

This is an example of the same thing. You should be able to predict what happens. Try it.

The result of the extension of HBL scanned memory into the display is that far left delayed patterns will often be more trouble than they're worth. Color and the presence of dots at the left margin depends on the presence or absence of dots at the right margin or in undisplayed memory. One helpful hint: erase all $XX7F and $XXFF bytes of HIRES memory as part of your erase routines. Then don't use these bytes except to purposely manipulate the far left dots in the upper third of the screen.

This section has shown how interference borders can be used to display isolated dots or vertical line segments in HIRES that are not one of the four HIRES colors. Mostly, though, interference borders are a nuisance. Anytime two different colors get close to each other horizontally, the video pattern at their border is different than either of the solid colors when compared to COLOR REFERENCE. Awareness of the causes of off color fringes should help you experiment with color combinations that produce eye pleasing displays.

LORES GRAPHICS OUTPUT

LORES blocks are not generated the way you would expect from looking at a television. You would expect big one microsecond pulses on the PICTURE signal would be required to produce those big one microsecond wide blocks. In reality, the only pulses that are one microsecond wide in LORES are white blocks. The colored blocks are made up of a string of narrow pulses, too narrow for a television to paint without blurring them into blocks, and narrow enough that they will be passed by the television's chrominance amplifier.

Like the HIRES colored picture, the LORES colored picture signal swings back and forth between the black and white levels at 3.58 MHz. But where the HIRES colored signals are nearly symmetrical, the LORES colored signal may or may not be symmetrical. This is why there is a greater variety of colors available in LORES.

The logical configuration in LORES is as follows:

PIX MUX S2 = LORES TIME	- high
PIX MUX S1	- VC latched by LD194
PIX MUX S0	- H0 latched by LD194
GRAPHICS SHIFT MODE	- dual 4-bit end around
GRAPHICS LOAD CLOCK	- 14M rising during LD194
GRAPHICS SHIFT CLOCK	- 14M rising

This establishes the nature of the LORES shift. Only four of the eight pattern bits are used in a video cycle; DL0-DL3 in B4 are selected during latched VC', and DL4-DL7 in B9 are selected during latched VC. Latching the scanner states during LD194 synchronizes them to the video cycle. The VC control of the picture MUX results in bits DL0-DL3 controlling the upper block and DL4-DL7 controlling the lower block.

Since each 4-bit section of the GRAPHICS shifter circulates as clocked by 14M, the sections circulate 3.5 times per video cycle. Nice coincidence, that: 3.5 million circulations per second—the same frequency as COLOR REFERENCE. As the selected 4-bit pattern is rotated, either its least significant bit (Q0) or its third least significant bit (Q2) is clocked to the picture flip-flop. Q0 is selected in video cycles where H0 was latched low (even memory addresses), and Q2 is selected in video cycles where H0 was latched high (odd memory addresses). This is what compensates for the alternating phase relationship between the video cycle and COLOR REFERENCE.

The bottom of Figure 8.8 shows the output of three different pairs of LORES blocks. These examples illustrate the nature of LORES timing. The examples shown all assume that latched VC is low. Timing is identical when latched VC is high except B9 outputs are selected for the PICTURE signal instead of B4.

The first example shows two adjacent 1001 blocks being output. In the even cycle, this pattern is output 3.5 times beginning with the LSB: 10011001100110. In the odd cycle, the pattern is output beginning with the third LSB: 01100110011001. In either an even or odd cycle the PICTURE signal is a symmetrical square wave with the same relationship to COLOR REFERENCE as HIRES orange. The coloring of the left and right edge of the block depends on the pattern of the adjacent blocks. If adjacent clocks are the same pattern, the PICTURE signal is continuous, meaning orange mates to orange with no off color fringe between two blocks. Different colors mate together with a joining pattern which is not the same as either of the joining colors, creating a color fringe which is more or less prominent depending on the colors and whether they meet at an odd-even or even-odd junction. The examples are marked with an X at the left and right sides to show that the color there depends on the adjacent patterns.

The second example in Figure 8.8 is even 0111, light blue, followed by odd 1000, dark brown. These patterns produce assymetrical 3.58 MHz square waves whose 3.58 MHz sinusoidal component is passed by the television's chrominance amplifier to produce different colors. The assymetrical square waves are produced by patterns with only one bit set or only one bit reset. Those with only one bit set produce dark colors, because the PICTURE signal spends most of its time in the black. Conversely, the patterns with only one bit reset produce bright colors. As the example shows, the picture pattern at the border between colors 0111 and 1000 is a combination of the two separate patterns. Even 0111 followed by odd 1000 produces a pinkish border.

The light blue block shows that some things are predictable about bordering colors in LORES blocks. Any odd pattern which ends in the white level will combine with the left side of even 0111 to form a white border. The bright colors are particularly prone to forming white borders, because they're only one black period away from being white themselves.

The third LORES example in Figure 8.8 is even 0101 followed by odd 1010. These are the two gray LORES patterns. They are gray, because the PICTURE signal they produce is 7 MHz, which will not be passed by the television's chrominance amplifier. Now gray is really white in a dark disguise. White light can come in many intensities as evidenced by a black and white television picture, and LORES patterns 0101 and 1010 are just less intense white. They are equal to each other in intensity, and are therefore identical shades of gray. This is why the technical overview stated there were 15, not 16, LORES colors including black and white.

Even though the two grays produce the same medium intensity, colorless blocks, they are 180 degrees out of phase with each other. Thus, when 1010 follows 0101 there is a discontinuity in the waveform at the border between them and a resulting color fringe. This can be done on purpose to separate two gray solids horizontally, or it can be avoided by using only 0101 or 1010 in a display. A good practice would be to choose one gray over the other to minimize unpleasant fringe borders with other colors.

When the LORES colors are displayed side by side in numerical sequence, there is no apparent color continuity between them. The fact is that they form a circular pattern of eight color tones determined by the phase relationship to COLOR REFERENCE. This is not apparent when they are in numerical sequence, because video processing treats the 4-bit color data as a dot pattern, not a numerical value.

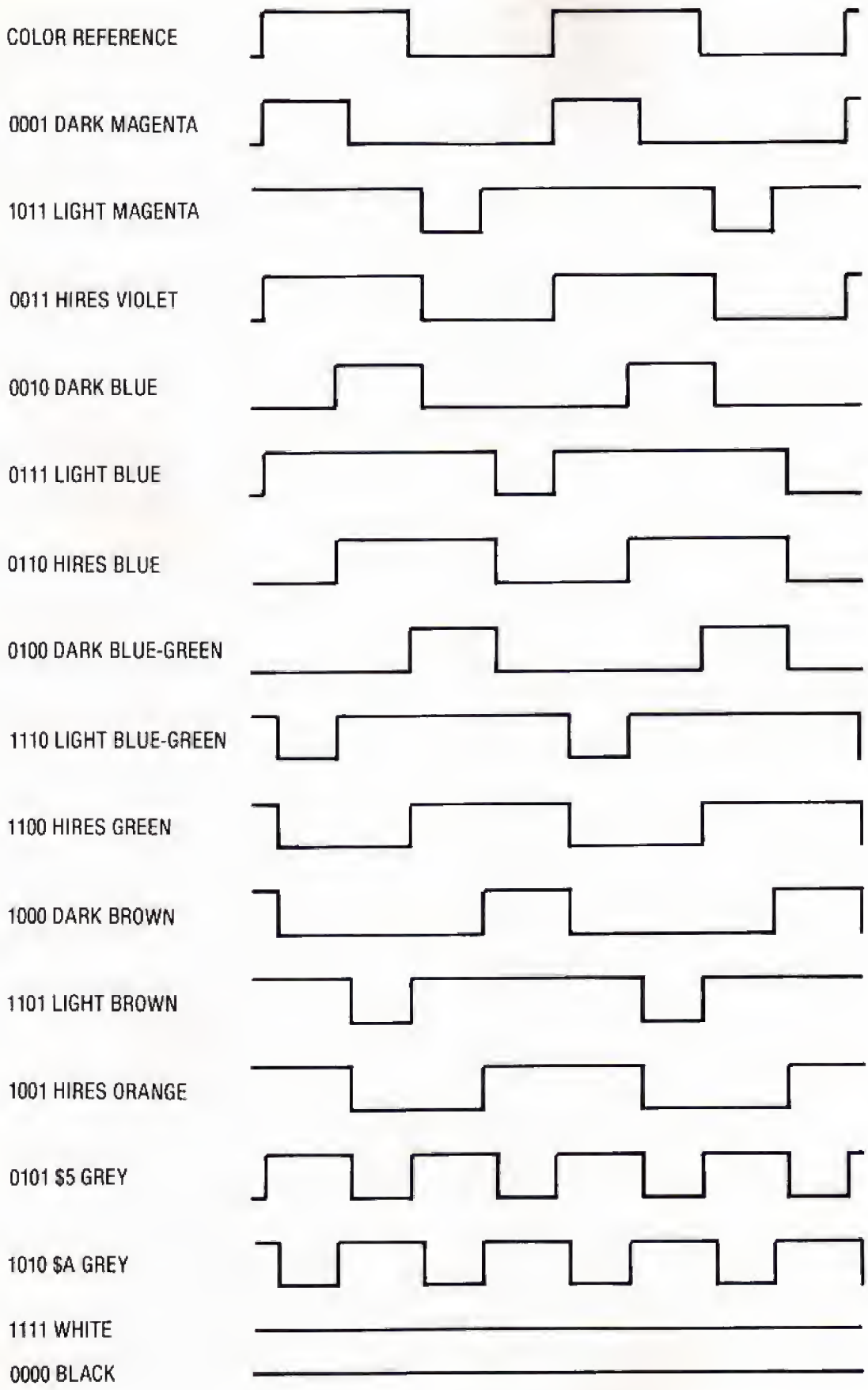


Figure 8.11 LORES Patterns at B10, Pin 5.

Figure 8.11 shows the PICTURE signals, compared to COLOR REFERENCE, which are produced by the various LORES patterns. White levels are a little longer than equivalent black levels because high to low propagation on the picture flip-flop takes longer than low to high propagation. The colors are shown in an order in which the picture pulse shifts right as the colors progress from top to bottom. A very interesting point becomes evident when looking at this figure. There are four color tone pairs: dark and light magenta, dark and light blue, dark and light blue green, and dark and light brown. For example, the dark magenta pulse is surrounded equally on both sides by the light magenta pulse, and the horizontal center of both pulses is at the same point on the COLOR REFERENCE. As a result, they produce the same color tone, but a series of wide pulses is brighter and whiter than the series of narrow pulses with the same color tone.

Now color 0001 is usually referred to as magenta, and color 1011 is usually referred to as pink. This book has been calling pink "light magenta" to make the point that color 1011 looks like color 0001 with a lot of whiteness in it. Anyone who wants to is encouraged to call pink "pink."

Figure 8.12 is a photograph of the LORES colors based on their circular nature. This display was generated by drawing the LORES display in LORES, drawing the color numbers in HIRES, and switching between LORES and HIRES in a 17030 cycle loop. This sort of screen mode switching produces a pronounced flicker which is very annoying when viewed. The flicker goes away in the photograph, however, which was taken at 1/30th of a second exposure. In this display, the different color tones are in different sectors of a circle (if you can picture a rectangular circle) and brightness is represented radially in the circle with dark at the center and white at the outside. Black, gray, and white cover all sectors of the circle, because they have no coloring. Black is the darkest color. Then comes 0001, 0010, 0100, and 1000. The grays, 0101 and 1010, are the same brightness as the HIRES equivalents, 0011, 0110, 1100, and 1001. Next brightest are 0111, 1110, 1101, and 1011. Brightest of all is white. Looking at the colors in this way should give you insights when you are trying to produce pleasing LORES displays.

MIXED MODE SWITCHING

A final topic to consider in the video generator is MIXED mode scanning. This is the reason we have to live with those RAS' delayed terms, so we'll have a closer look.

Figure 8.13 is a timing diagram of the last display cycle of line 159 in MIXED mode. At the left side of this figure you can see the horizontal section of the video scanner switch from 1111111 to 0000000. This is the beginning of HBL and the beginning of the long cycle, which has two LD194 pulses instead of the normal one. At the same time the horizontal section of the scanner goes to zeroes, the vertical section goes to 010100000 making the term V4 • V2 true. This identifies TEXT time, but you can't immediately switch to TEXT because the final displayed GRAPHICS pattern is not yet output. For that matter you can't start blanking yet either.

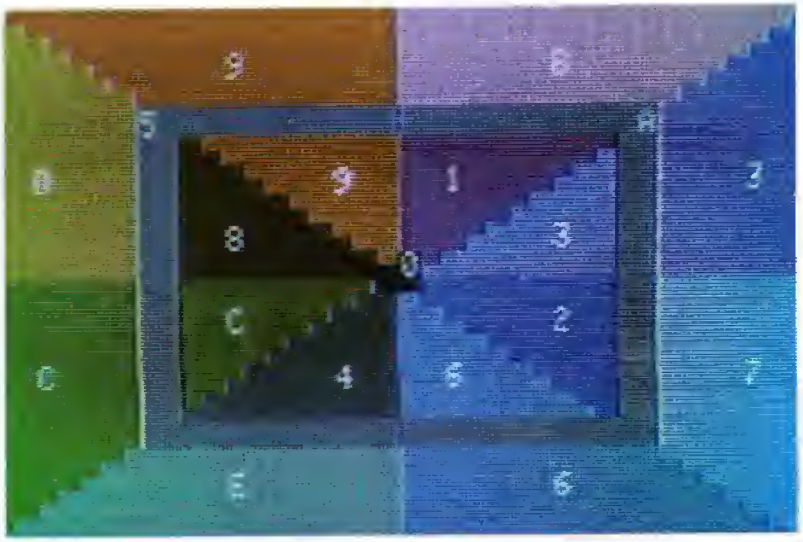


Figure 8.12 LORES Colors.

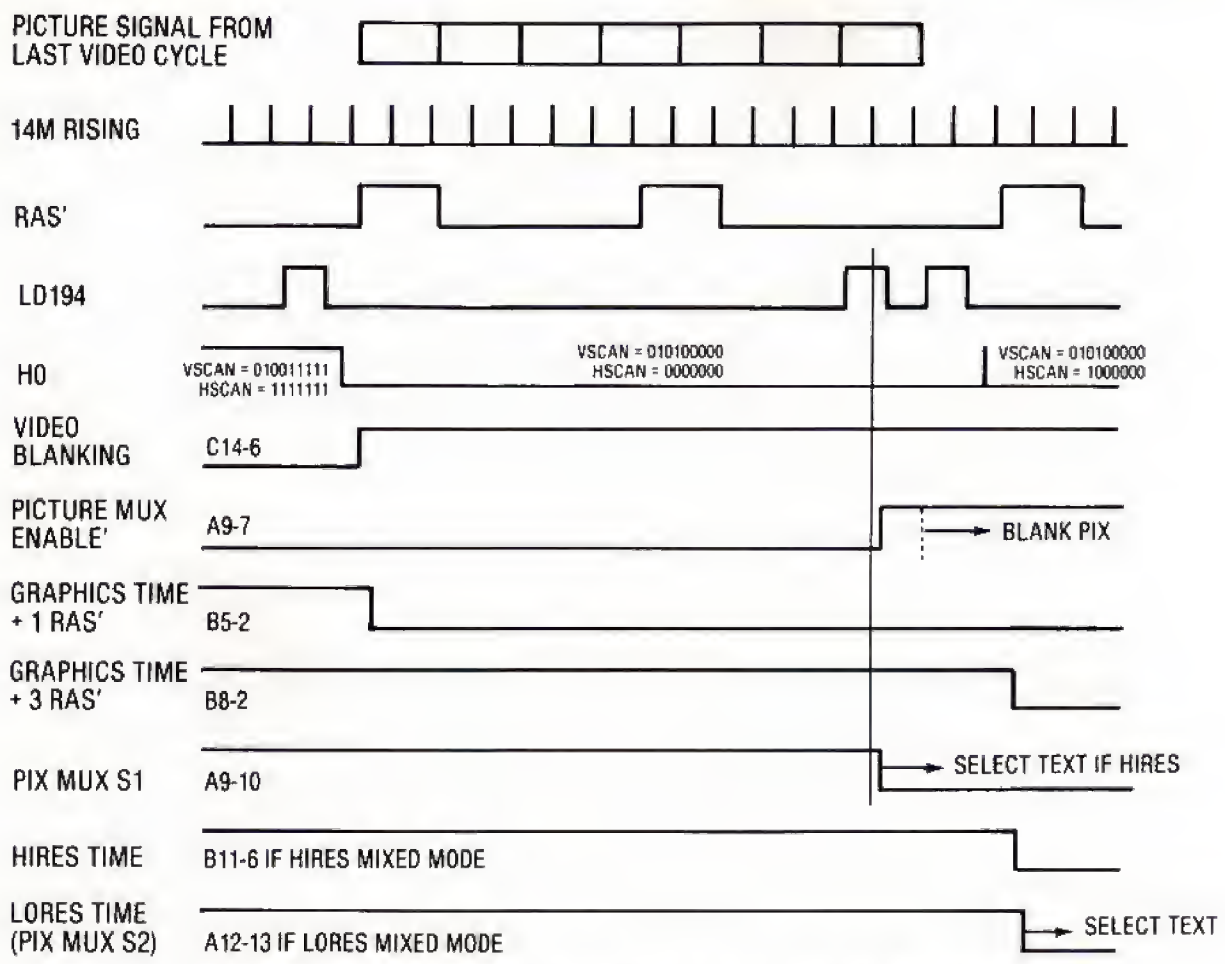


Figure 8.13 Switching from GRAPHICS to TEXT in MIXED Mode.

What happens is that the GRAPHICS TIME signal is shifted through three flip-flops which are clocked by RAS' rising. This creates the terms GRAPHICS TIME + 1 RAS' and GRAPHICS TIME + 3 RAS'. These are the signals which control the video generator configuration. Here is an order of events after V4 • V2 goes high:

1. RAS' rises, bringing GRAPHICS TIME plus one RAS' low.
2. 14M rises during LD194, signifying the end of the video cycle. This loads meaningless data to the GRAPHICS and TEXT shifters that will not be output to the screen. It latches the picture MUX ENABLE' signal high (since VIDEO BLANKING is now high). This forces the picture MUX output low. Picture MUX select S1 also goes low at this time; in HIRES because GRAPHICS TIME plus one RAS' is low and in LORES because VC is low. In HIRES MIXED mode, TEXT is now selected at the picture MUX.
3. 14M rises again, clocking the low from the picture MUX output to the picture flip-flop. This is the instant that HBL forces the PICTURE signal into the black, and it exactly coincides with the end of the last video pattern. If, however, the last video pattern is delayed HIRES, the blanking forces the PICTURE signal to the black prematurely, cutting the last dot in half. This happens at the end of any HIRES scan, not just line 159.
4. RAS' rises, bringing GRAPHICS TIME plus three RAS' low. This is followed closely by HIRES TIME falling in HIRES mode, and by LORES TIME falling in LORES mode. HIRES TIME is used in the RAM address multiplexor to cause HIRES memory addressing or TEXT/LORES memory addressing. Bringing it low just after RAS' rises switches TEXT/LORES memory addressing in time so that RAS' falling will correctly clock the next TEXT memory ROW address to RAM. LORES TIME' is the select input to the LS257 configuration control chip (A8), and LORES TIME is the S2 select input to the picture MUX. Bringing LORES TIME low here selects TEXT at the picture MUX in LORES MIXED mode.

The switch back to graphics is made at the same point in the horizontal scan, and the delayed timing similarly prevents display of the last video pattern of line 261. The timing is similar when switching to and from text via the $C050/$C051 toggle, but switching does not necessarily occur during a blanking gate.

SOFTWARE APPLICATION

ASPECT RATIO IN THE APPLE DISPLAY

The **aspect ratio** of an NTSC standard television is 4 to 3. This is the ratio between the horizontal size of the display and the vertical size of the display. Since the ratio of displayed to undisplayed scanning is different in the Apple than in an NTSC standard broadcast, the aspect ratio of the Apple displayed window is not 4 to 3. However, we can utilize knowledge of television scanning and Apple scanning to compute the aspect ratio of Apple graphics displays. Programming computations can use this information to make squares square and circles circular.

The 4 to 3 aspect ratio is the ratio of displayed horizontal and vertical travel of the beam. The television horizontal display gate is about 53.6 microseconds compared to 10 microseconds blanking, so 53.6/63.6 or 84.3% of the horizontal cycle is displayed. Vertically 242.5 out of 262.5 lines or 92.4% of the horizontal scans are displayed. Let HD be the distance a beam would travel at horizontal trace speed during a horizontal cycle period. Let VD be the distance a beam would travel at vertical trace speed in a vertical cycle period. Then:

$$(.843\ HD)/(.924\ VD) = 4/3$$

and

$$HD/VD = 1.46$$

Now in Apple scanning, 560 out of 912 14M periods are displayed in a horizontal cycle (remember the long cycle), so 61.4% of the horizontal cycle period is displayed. Vertically 192/262 or 73.3% of the vertical cycle is displayed. The aspect ratio of the Apple display window is therefore:

$$(.614HD)/(.733VD) = .838\ (HD/VD) = 1.225$$

This means the display window is 1.225 times as wide as it is high.

Let's say there are 1000 units vertically and 1225 units horizontally in the display. The distance vertically between HIRES dot centers would be 1000/192 = 5.21 units. In 280 point graphics, the distance horizontally between dot centers would be 1225/280 = 4.37 units. The dots are therefore further apart vertically than they are horizontally. More precisely, the ratio of the distance between horizontal dots to the distance between vertical dots is 4.37/5.21 = .84. This will vary with television alignment, but it is very practical to use .84 and its inverse, 1.19, as linearity compensation figures in programs. Here is an example:

```
10 HGR : HCOLOR = 3 : HPLOT 0,0
   TO 0,159 TO (159*1.19+.5),159
   TO (159*1.19+.5),0 TO 0,0 :
   REM ADD .5 TO ROUND OFF.
```

This Applesoft program draws a square, because the number of horizontal units is equal to the number of vertical units times the ratio 1.19.

In 140 point graphics computations, the horizontal to vertical distance ratio would be doubled to 1.68. In 560 point graphics computations, the ratio would be cut in half to .42. LORES blocks are 7 HIRES dots wide by 4 HIRES dots high, so the horizontal distance between centers of blocks is (7/4)(.84)=1.47 times the vertical distance between centers of blocks. The horizontal distance between the centers of text characters is (7/8)(.84)=.735 times the vertical distance. The 5x7 text character itself is (4/6)(.84)=.56 times as wide as it is high (measuring between the centers of the corner dots). All of these ratios are summarized in Table 8.2.

Table 8.2 Size/Distance Ratios on the Apple Screen.

PLOTTING MODE	HORIZONTAL/VERTICAL	VERTICAL/HORIZONTAL
DISPLAY WINDOW SIZE	1.225	.816
HIRES 140 Point Distance	1.680	.595
HIRES 280 Point Distance	.840	1.191
HIRES 560 Point Distance	.420	2.381
LORES Distance	1.47	.68
Text Distance	.735	1.361
Text Size	.560	1.786

HARDWARE APPLICATION

ELIMINATING COLORED SHADOWS FROM TEXT

In Revision 1, a Color Burst Killer circuit was added which turns off the COLOR BURST in TEXT Mode. This is nice, because white letters are much easier to read than violet and green letters. The problem with this circuit is that it does not completely eliminate the COLOR BURST but reduces it to a very small level. Some televisions lock up on this residual COLOR BURST in an odd way that adds very unpleasant color shadows to text. In a recent mod to the Apple II Plus, this problem was eliminated by shutting off the COLOR BURST with a TTL gate before it gets to the sometimes ineffective Color Burst Killer, Q6. This Application Note offers some suggestions for eliminating color shadows from older Apples on which the COLOR BURST is not quite dead in TEXT mode.

There is a color killer adjustment on the back of many television sets, especially older ones. This adjustment sets the sensitivity of the television circuit which detects the presence of the color burst on the back porch of the horizontal blanking gate. This adjustment is labeled "color killer" if it's there, and it may be just a hole in the back through which you can stick a plastic screwdriver to engage the slot in a potentiometer.

DO NOT REMOVE THE BACK OF THE TELEVISION SET UNLESS YOU ARE A QUALIFIED ELECTRONIC TECHNICIAN. THE VOLTAGES ARE LETHAL.

The television color killer adjustment is very simple to make. You simply adjust it so color broadcasts have color in them but monochrome broadcasts don't. It is so simple that our Asian television suppliers have figured out you don't really need a color killer adjustment in the back, so most new televisions don't have it. If it is there, feel free to tweak on it and try to eliminate your color shadows that way. Just make sure that you still get color in GRAPHICS mode and on broadcasts if you also use the TV for normal reception.

If no color killer adjustment is accessible on your television, there is a second option which works well on my Apple. Increase the size of R6, the COLOR BURST summing resistor. The size of this resistor determines the amplitude of the COLOR BURST when it is added to the PICTURE signal. By increasing the resistor, you decrease the size of the COLOR BURST and the amount of residual signal that gets through the Color Burst Killer. The trick is to select a resistor that eliminates color shadows from the screen but doesn't prevent color graphics generation. This depends on the television rather than the computer. With my composite video monitor, increasing R6 from 2.7 Kilohms to 4.7 Kilohms worked nicely.

Disconnect the power cord before doing any work on the motherboard. R6 is right next to the game I/O socket and is labeled 2.7K. You can remove it from the top of the motherboard with a soldering iron and needlenose pliers. After R6 removal, desolder the holes with a solder sucker. Try a 4.7K resistor without solder to see if it solves the problem in a satisfactory way. If 4.7 Kilohms doesn't work, experiment. When you have the right resistor size, solder it in, and you're done.*

*Please read the NOTE OF CAUTION at the beginning of the book before performing any modifications to your hardware.

HARDWARE APPLICATION

PROGRAMMING SCREEN CHARACTER SETS IN EPROM

In Revision 7 and later Apples, ones with 2316 TEXT ROMs, you can install your own screen character sets. You may design your own upper case/lower case set or use an existing upper case/lower case set. You may also design your own INVERSE set and FLASHING set which will be output anytime a program outputs inverse or flashing text, but they don't have to be inverted or flashing characters. This Application Note contains some suggestions for burning screen character EPROMS for Apples that will accept them. Please refer to Figure 8.7 during these discussions.

There are three groups of characters to consider when building character sets: symbols and numbers, upper case alphabet, and lower case alphabet. These will be referred to here as symbols, upper case, and lower case. Character patterns for each group require 256 bytes of storage so a complete character set requires 768 bytes. You should place your primary character set in the top 768 bytes of the 2716 EPROM with symbols at addresses $500-$5FF, upper case at $600-$6FF, and lower case at $700-$7FF. This will result in word processing and similar programs being able to output lower case to your screen. These 768 bytes will be driven out by Apple character codes in the $A0-$FF range.

Where can you get a full ASCII set of text patterns? There are numerous sources for character set patterns. One source is the *DOS TOOL KIT* distributed by Apple. Among other things, this valuable disk contains 21 character sets and *ANIMATRIX*, a program which implements computer aided design of other character sets. These HRCG (HIRES Character Generator) sets are meant to draw text on the HIRES screen, but they may be adapted for your TEXT EPROM. The main difference between a HIRES character pattern and a TEXT ROM pattern is that they are reversed. The Apple's 2316 TEXT ROM is connected to the TEXT shifter so that the most significant bit is shifted out first. HIRES patterns are shifted out least significant bit first, so HIRES patterns are in reverse order.

A second adaptation that must be made to HRCG character sets is to offset the different effect of bit 7 in the TEXT ROM and in a HIRES pattern. Bit 7 of your EPROM fonts will control flashing in the lower 1024 bytes of the EPROM. In the upper 1024 bytes, bit 7 will have no effect, but its a good idea to leave it reset any time you are not specifically programming a character you want to flash. Bit seven set in a HIRES pattern causes the pattern to be delayed half a dot position. In the *DOS TOOL KIT* sets, DL7 is occasionally set to improve the smoothness of a character. These characters would look a little cockeyed without the delay, so they need to be modified before using them in your TEXT EPROM. The way to do this would be to load the set into *ANIMATRIX* where the few delayed patterns can be easily spotted and the character modified for symmetry with no pattern delays. Needless to say, if you use *ANIMATRIX* to design your EPROM text patterns from scratch, don't use the delay feature.

Several other character generating programs are available, one very nice one at no cost. The August, 1980 edition of *KILOBAUD Microcomputing* magazine contains the listing of a character generating aid for the Apple and includes a full ASCII set. The article is "Graphics Character Generator" by Robin B. Moore. This character generator does not use delayed patterns, so there would be no problem adopting the character set for your TEXT EPROM.

If you have selected a HIRES character set for your primary ASCII patterns, you must move it to your EPROM programming buffer in reverse order. Here is a little program sequence which can be used in a transfer loop:

```
        LDA (FONT),Y ;Get byte from
                     ;source font
        LDX #7
MOVELP  LSR A        ;Transfer D6-D0 in
                     ;reverse
        ROL WORK     ;shift pattern to
                     ;intermediate loc'n
        DEX
        BNE MOVELP
        LDA WORK
        AND #$7F     ;Reset FLASH
                     ;control bit
        STA (BUFR),Y
```

The 256 bytes of your EPROM just below the primary set will not be utilized by most commercial programs. This area of ASCII ($80-$9F) is the CONTROL codes. You may place any sort of alphabetic or symbolic set in this area of your EPROM and output these symbols to the screen by storing codes $80-$9F in Text memory. This will also let you know when any program is storing control codes in

text memory. You will store the CONTROL patterns at $400-$4FF of your EPROM.

The lower 1024 bytes are the INVERSE area ($0-$1FF) and the FLASHING area ($200-$3FF). The ASCII codes which will drive out these patterns will be in the $00-$7F range. If you leave O7 (EPROM Output 7) reset in these patterns, the resulting characters will be inversions of your patterns. If you set O7, the resulting characters will flash. You may force normal characters (white on black) in these addresses by storing inverted patterns and leaving O7 reset. In both the INVERSE area and the FLASHING area, there is space available for a symbolic set and an upper or lower case set.

Notice that the upper half of the codes are divided into CONTROL, symbolic, upper case, and lower case, but the lower half isn't. The upper half is divided like true ASCII. The lower half is tailored to Apple features, giving the user a symbolic/uppercase INVERSE set and a symbolic/uppercase FLASHING set. Since Applesoft INVERSE and FLASH commands and the monitor "I" command support this division, you would be well advised to maintain it in your EPROM.

An idea for your INVERSE set is to make it normal rather than inverse, and let the alphabet be lower case. This will give you a convenient way of outputting lower case to the screen from Applesoft. If you do this, any alphabetic screen output from Applesoft with INVERSE active will be normal lower case. You may want to use one of the different *TOOL KIT* sets for this purpose, so you will know when the INVERSE set is being used.

There are a few symbols associated with the upper case and lower case alphabetic sections. If you store lower case patterns in an upper case area like the INVERSE area, it is best to install upper case associated symbols there. Of the 32 characters in an alphabetic section, the first one and last five are symbols. Therefore, when installing lower case patterns in the INVERSE Area, reserve addresses $0-$7 and $D8-$FF of your EPROM for patterns from an upper case set.

To transfer the symbolic section of a *TOOL KIT* set to the INVERSE Area, transfer the first third of the *TOOL KIT* set to $100—$1FF of the EPROM buffer. The alphabetic part of the INVERSE area is $0-$FF of the EPROM. Lower case for this area can be gotten from the last third of a *TOOL KIT* set. Upper case for this area can be gotten from the middle third of a *TOOL KIT* set. O7 in each byte of the EPROM should be reset for no flashing, set for flashing. To make the set normal instead of inverse you must store the inversion of the *TOOL KIT* patterns at the EPROM buffer, this in addition to reversing order of the ones and zeroes as in the earlier programming example. In the earlier example, O7 was reset by ANDing the pattern with #$7F. You can invert the pattern and leave O7 reset by following the AND instruction with an EOR:

```
AND   #$7F
EOR   #$7F
STA   (BUFR), Y
```

Programming the FLASHING area of your EPROM is exactly like programming the INVERSE area: O7 high causes flashing; the lower half ($200-$2FF) is the alphabet; the upper half ($300-$3FF) is symbols. There is a restriction you should place on your FLASHING set. The FLASHING set must be very easily distinguished from any other set, because the Apple cursor is generated by

Table 8.3 The Division of Screen Text Patterns in a 2048 x 8 ROM.

APPLESOFT COMMAND	CATEGORY	ASCII	EPROM ADDRESS
INVERSE	ALPHABET	$00-$1F	$000-$0FF
	SYMBOLS	$20-$3F	$100-$1FF
FLASH	ALPHABET	$40-$5F	$200-$2FF
	SYMBOLS	$60-$7F	$300-$3FF
NORMAL	CONTROL	$80-$9F	$400-$4FF
	SYMBOLS	$A0-$BF	$500-$5FF
	UPPER ALPHA	$C0-$DF	$600-$6FF
	LOWER ALPHA	$E0-$FF	$700-$7FF

flashing the text character under the cursor. You may make your FLASHING set distinguishable by having it flash, inverting it, putting a mark on it (like a dot in the upper left corner of every character), or by using an odd HRCG set in the FLASHING area. As an example, you could use the symbolic and upper case "OUTLINE" set from the *DOS TOOL KIT* in the FLASHING area. Your CURSOR position would then be identified by a text character changing to the interesting OUTLINE form. You would want to let the symbols be displayed inverted or flashing because the symbols and numbers in the OUTLINE set are not distinct enough from a normal set.

What's wrong with letting FLASHING characters flash and INVERSE characters be inverted? Nothing—it's just that customizing your Apple is fun and rewarding. Burning your special screen character EPROM is an easy way to do this, and the capability is a nice feature of newer Apples.

A final word of advice is to use 350 nanosecond EPROM for your TEXT pattern ROM rather than the more commonly available 450 nanosecond variety. The Apple design only gives a 390 nanosecond address setup time before the ROM data must be valid. 450 nanosecond EPROM will probably work, but if your screen text starts looking odd on a hot summer day, you'll know what's causing it.

TECHNICAL NOTE

DETAILS OF TELEVISION PROCESSING OF APPLE VIDEO

A rigorous examination of the television processing of the Apple signal involves technical details beyond the scope of *Understanding the Apple II*. Brief descriptions of some of these technical details are presented here for those readers who wish to study television processing of Apple II video in depth.

A square wave is the sum of the odd harmonics of a sine wave of the same frequency. For example, a 3 MHz square wave can be produced by summing the following sine waves:

3 MHz at amplitude A
9 MHz at amplitude A/3
15 MHz at amplitude A/5
.
.
.

The more harmonics added, the more perfect the square wave. This sinusoidal make-up of a square wave is significant because tuned circuits such as those found in a television receiver respond to the sinusoidal components of signals. A square wave will be processed as the sum of its sinusoidal components.

Generally, Apple PICTURE signals, which produce color displays, are 3.58 MHz square waves. These square waves modulate a television carrier frequency in the user supplied modulator, creating a radio frequency with a square modulation envelope. Sinusoidally, the square wave intelligence is carried by the following series of frequencies:

carrier
carrier + 3.58 MHz
carrier + (3.58 MHz) x 3
carrier + (3.58 MHz) x 5
.
.
.

The IF strip of the television will pass the sine wave carrier and those sine wave frequencies above the carrier, up to carrier + 4.2 MHz. Only the carrier and carrier + 3.58 MHz of the above distribution are within this range. As a result, the 3.58 MHz square envelope is converted to a 3.58 MHz sinusoidal envelope, and the output of the second detector in the television is a 3.58 MHz sine wave. This sine wave is passed by the chrominance amplifier to the synchronous demodulator, where it is processed with the reconstructed color reference sine and cosine waves to produce color signals. It is also processed by the luminance amplifier to produce the luminance signal.

Many televisions have a 3.58 MHz trap in the luminance path which reduces color interference with the luminance signal. The effect of this trap is to remove the 3.58 MHz variation, and pass a gray luminance level which lasts for the duration of the 3.58 MHz presence. A similar effect is felt on the 7 MHz modulation envelope produced by LORES 5 and 10 colors. The 7 MHz + carrier frequency is out of the band pass of the IF strip, so the 7 MHz variations are removed and replaced by a gray level. These solid gray levels do not degrade the Apple luminance signal, but enhance it. We cannot see 3.58 MHz variations in picture brightness at normal viewing distance. We just see solid blocks of brightness. Conversion of 7 MHz and 3.58 MHz signals to solid gray levels does not, therefore, degrade the picture we perceive.

A very interesting special case among Apple PICTURE signals is that created by turning alternating groups of three HIRES dots on and off. Conventional Apple wisdom is that this will create a horizontal dashed line with white coloration of the dashes because they are three adjacent dots. However, the picture signal produced by this pattern is a square wave of 3.58 MHz/3. This square wave has significant 3.58 MHz sinusoidal content, since 3.58 MHz is the third harmonic of the fundamental square wave frequency. This produces a 3.58 MHz sine wave at the output of the chrominance amplifier about one third the amplitude of the signal produced by a 3.58 MHz PICTURE signal. The result is a washed out coloring of the 3.58 MHz/3 PICTURE signal, not nearly as intense as the coloring of 3.58 MHz PICTURE signals. The chrominance amplifier frequency band is from 3.1 MHz to 4.1 MHz, so any PICTURE signal from 3.1 MHz/3 to 4.1 MHz/3 should have some coloration.

A second television phenomenon is less predictable. Many televisions have a coupling transformer or inductor/capacitor combination at the input to the chrominance amplifier. I have found that this input circuit has a marked tendency to ring at 3.58 MHz when the PICTURE signal switches from white to black or black to white. This ringing produces an output from the chrominance amplifier of

about the same amplitude as that produced by a 3.58 MHz/3 PICTURE signal. One result is edge coloring of white screen displays. The 3.58 MHz ringing should vary greatly from television to television, and may be reponsible for many of the off-color edges found in Apple displays.

In a normal television broadcast signal, the luminance signal energy is concentrated at multiples of the horizontal frequency removed from the picture carrier. This is because the luminance signal itself has a very high content of harmonics of the line scanning frequency. When the luminance signal modulates the carrier, the energy is largely distributed in groups centered at the following frequencies:

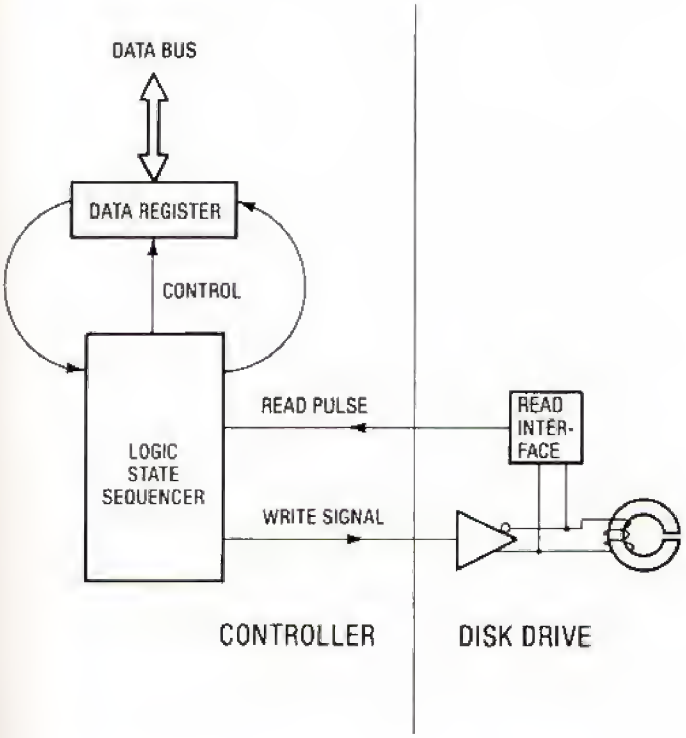
carrier
carrier + horizontal frequency
carrier + 2(horizontal frequency)
.
.
.

The color signals have a similarly high content of horizontal frequency harmonics. When the 3.58 MHz color subcarrier is modulated by a color signal, the energy is largely distributed at

3.58 MHz (suppressed)
3.58 MHz + horizontal frequency
3.58 MHz + 2(horizontal frequency)
.
.
.

The 3.58 MHz color subcarrier is used because carrier + 3.58 MHz resides between the "carrier + 227(horizontal frequency)" and the "carrier + 228(horizontal freq)" energy concentrations of the luminance signal. The "carrier + n(horizontal freq)" luminance distribution is thus interlaced with the "carrier + 227.5(horizontal freq) + n(horizontal freq)" distribution. This reduces interference between the chrominance and luminance signals.

The Apple PICTURE signal is high in horizontal frequency harmonic content, just like the normal television luminance signal and color signals. Therefore, the energy of the modulated carrier should be distributed primarily at "carrier + n(horizontal freq)" intervals. The process of modulating a 3.58 MHz subcarrier with a color signal does not take place, so the "carrier + 3.58 MHz + n(horizontal freq)" energy distribution should not exist to the same extent as it does in a normal television chrominance signal. However, the 3.58 MHz PICTURE signal with horizontal frequency harmonic periodicity should create some elements of the "carrier + 3.58 MHz + n(horizontal freq)" distribution when the carrier is modulated. It is, therefore, interesting to note that the 3.58 MHz COLOR REFERENCE signal of the Apple is 228 times the horizontal frequency, not 227.5 times the horizontal frequency as in a normal television signal. This means that the "carrier + n(horizontal freq)" energy distribution is superimposed, rather than interlaced, with the "carrier + 3.58 MHz + n(horizontal freq)" distribution. My conclusion is that the energy of a carrier wave modulated by Apple video is largely distributed at multiples of the horizontal frequency removed from the carrier frequency, with no interlaced distribution induced by the chrominance element.



chapter 9

The Disk Controller

The coming of age of inexpensive floppy disk drives has been a very important factor in the current proliferation of personal computers. By the same token, the timely introduction of the Apple **Disk II** and its associated **Disk Operating System** was a very important factor in the growth of Apple's share of the market.

There has been a real personality change in the Apple II since it was introduced. It was originally a neat, cassette based machine whose creative and versatile design made it an initial success. The original hardware and firmware are the rock upon which the Apple empire has been built. Assets from the original success put the company in a position to keep the Apple competitive with improvements such as Applesoft, RAM and ROM cards, and the Disk II. Combined with substantial software and hardware support from other vendors, these improvements made the Apple II into what it is today—one of the world's most popular personal computers.

The long term success of the Apple II could not have come about without the development of the Disk II. For simple storage of data and programs, disk I/O is merely faster and more convenient than cassette I/O. But for important computer uses such as word processing, data base management, business accounting, and file handling, the disk drive or its equivalent is mandatory. There can be no doubt that for most owners the disk drive is the most important peripheral in the Apple II computer.

Since the original Apple II was strictly cassette based, the interface to the Disk II had to be built on a peripheral card. The extent of motherboard support of the Disk II is an empty card slot and the Autostart ROM. The Autostart ROM has no disk handling routines but only looks for disk handling routines in the peripheral slots and jumps to them at power up. The **Bootstrap** program and the circuits that interface the computer with the drive are on the **disk controller**, which is a peripheral card usually installed in Slot 6. The disk controller is connected by a 20 wire ribbon cable to the disk **drive**, which contains more electronic circuits as well as the drive mechanisms. It is primarily the controller circuits and the DOS which determine the features of disk operation that are unique to the Apple, and it is the

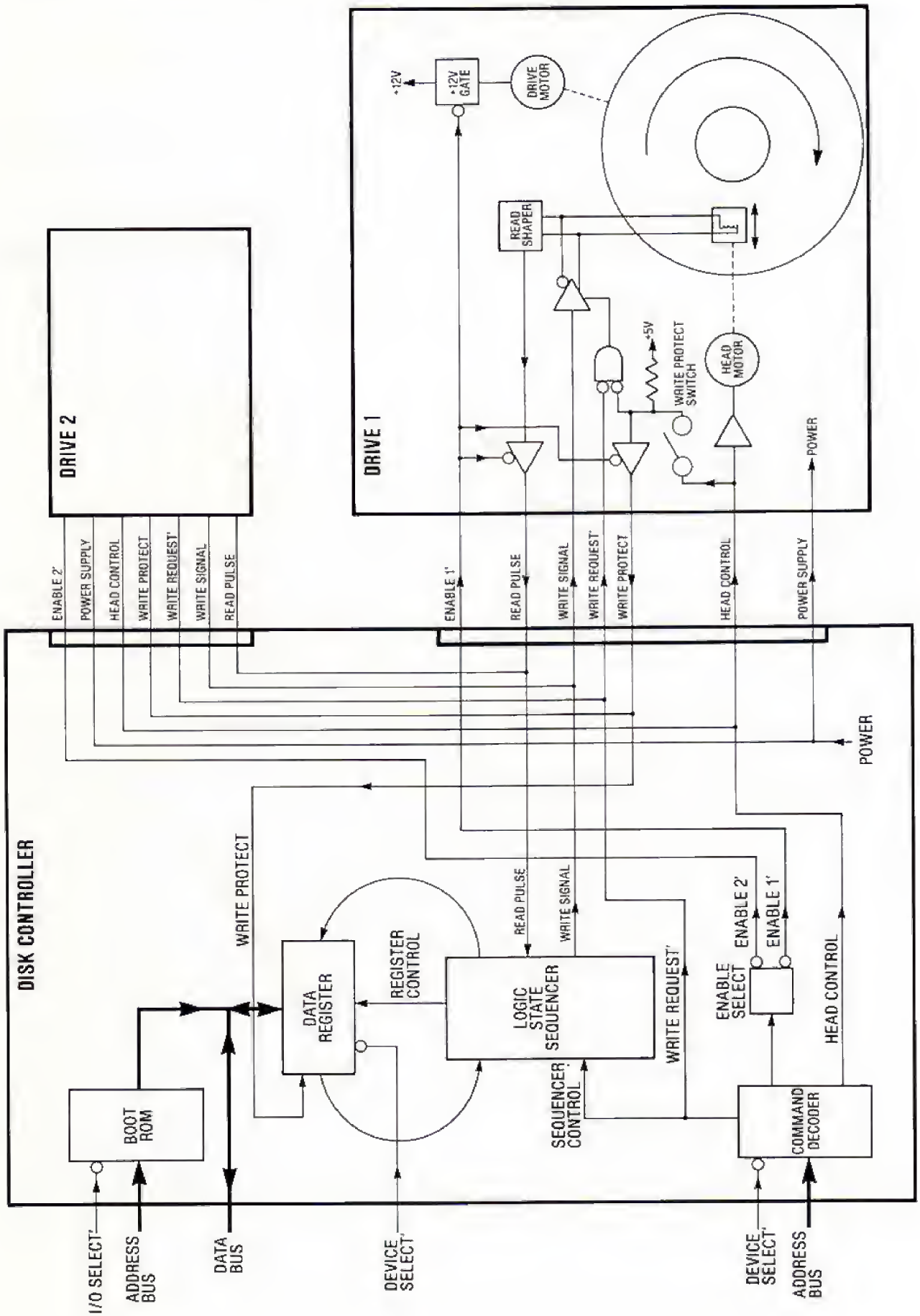


Figure 9.1 Functional Diagram of the Disk Interface.

controller circuits which are the main topic of this chapter. General features of the disk drive are also discussed, but no attempt is made here to document the DOS beyond the **RWTS** (Read or Write a Track and Sector) subroutine.*

DISK II OVERVIEW

Near the end of 1977, Apple Computer's decision making group was still very small. Mike Markkula, the chairman of the board, presented the group with a list of products needed to be developed for the Apple. At the top of the list was a floppy disk drive for the Apple II. Within a very short period of time, Apple developed the Disk II and released it along with its operating system, DOS 3.

The disk drive chosen by Apple was a Shugart SA400. Apple designed their controller card (mounted in a peripheral slot), analog card (mounted in the disk drive), and data formats around this standard drive. More recently, a subsidiary of Apple Computer Inc. makes the Disk II drives, but these drives are nearly identical to the old Shugart drives.

Other companies make drives and controllers which will work in the Apple II. If you are interested, a review of various 5 1/4" disk drives available for the Apple was presented in Jeffrey Mazur's "Hardtalk" column in the September, 1982 edition of *Softalk* magazine. Other compatible drives and controllers will be similar, but the discussions in this chapter detail the operation of the Disk II controller and, to a lesser extent, the Disk II drive.

Floppy disks are **magnetic media**, just like audio and video tapes. Reading and writing is performed by rotating the disk while a stationary read/write head presses against it. Disk speed is 300 RPM which translates to 48.4 inches per second on the inner track. This is a great deal faster than audio cassette tape speed so the rate of data transfer is greater than that achieved in a cassette storage system.

In between read or write periods, the head may be positioned radially so that different **tracks** can be written to or read from. The head is positioned precisely by a **stepper motor** under 6502 program control. There are thirty-five tracks on the Disk II, but some second source drives have forty or more tracks which can be utilized by modifying the DOS. There is no hardware sensor that can be used to determine which track the head is on. The DOS absolutely determines where the head is by running it against the outer stop at initialization. From that point it closely monitors head location, always saving the current position in RAM. Also, when reading data from or writing data to a formatted disk, the DOS always verifies head position by reading the stored track number from the disk and comparing it to the track number it is attempting to access.

Figure 9.1 is a functional diagram of Apple disk I/O. Data transfer between the MPU and the controller is eight bit parallel via the data bus. Control by the MPU is via the address bus, of course.* Data transfer between the controller and the drives is serial, and control of the drives is via multiple control lines serving various functions. The controller is primarily a digital data processing device, while the analog circuit card in the disk drive primarily performs the functions of amplification, shaping, and gating. Control of the disk is software intensive, meaning very little is done automatically by hardware.

Hard sectored disks are disks with little holes in them which divide the disk into a number of sectors. Disk drives supporting hard sector formats have a sensor in them which signals the host computer when a hole is passing by and allows a program to determine where the disk is in its circular trip. The Disk II does not have this feature, and the Apple DOS doesn't require it. Instead, the Apple uses a **soft sectored** format in which positional information is stored on the disk in uniquely identifiable **address fields**. These address fields are the "holes" which divide the disk into sectors and identify rotational position. The address fields contain an address field identifier and a volume-track-sector address from which programs can locate specific address fields. Behind each address field, there is **a data field** with space for 256 bytes of data. An address field and the data field that follows it make up a **sector** of disk information.**

*Operation of the DOS is well documented in the book, *Beneath Apple DOS*, by Don Worth and Pieter Lechner, Quality Software, 1982.

*In this chapter there is much discussion of the role of the MPU in disk I/O. The reader will do well to remember that while the MPU has certain capabilities of manipulating the Disk II hardware, these capabilities can only be utilized under program control. In other words, a 6502 program must supervise the role of the MPU in disk I/O.

**The Apple Disk II and DOS will work with hard sectored disks, but the holes in the disk will be ignored.

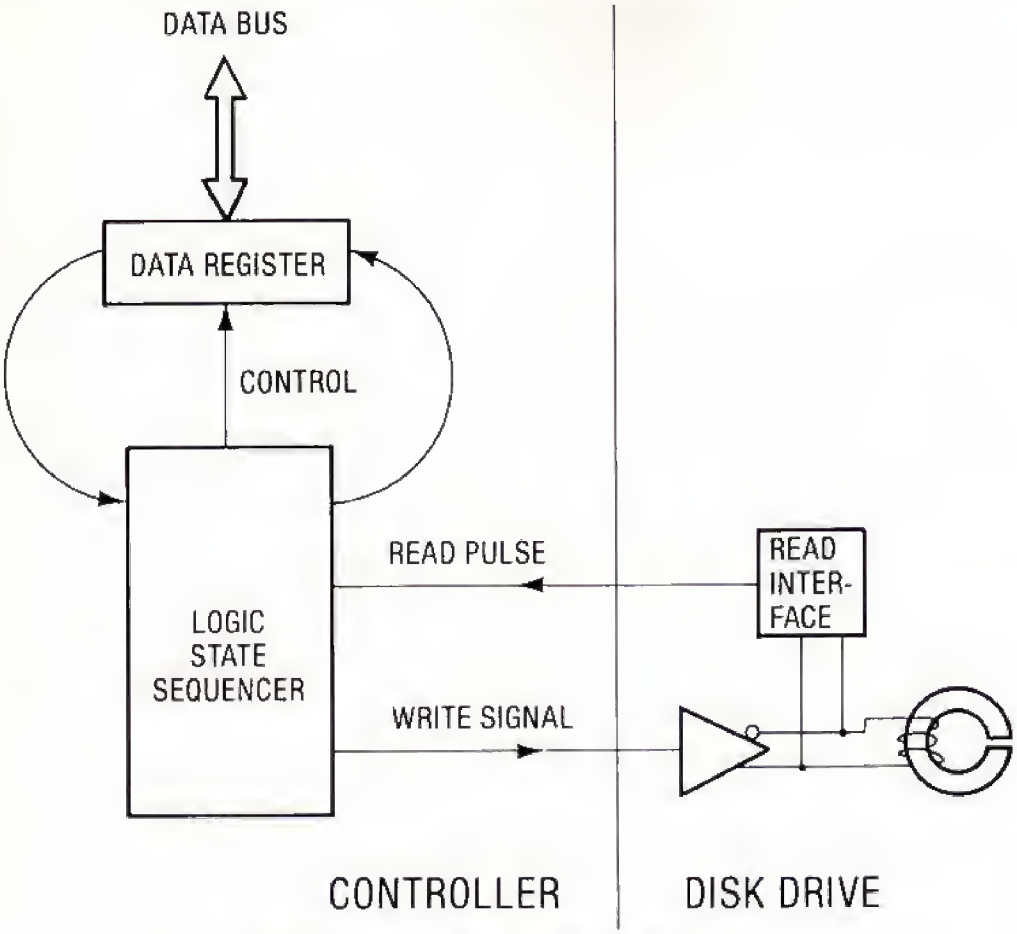


Figure 9.2 Data Transfer in Disk I/O.

The DOS writes 16 sectors (address fields followed by null data fields) in its FORMAT routine. The **16 sector format** is not unconditionally dictated by hardware. It is just a very reasonable number of sectors to have, considering the facts that the 6502 addressing modes are best suited for manipulation of data blocks up to 256 bytes in size, that 256 bytes is a workable size for data blocks in disk I/O, and that the Disk II is capable of storing 16 sectors of 256 bytes on a track in the DOS 3.3 format. As an example, the hardware will let you store data in eight sectors of 512 bytes each. Sixteen sectors of 256 bytes, however, is the only format supported by DOS 3.3 and its RWTS subroutine, and the only reasons to deviate from it are for copy protection or to have fun (using the word "fun" in a very broad sense).

The nuts and bolts details of disk I/O in most Apples are handled by the **RWTS** (Read or Write a Track and Sector Subroutine) of the DOS. The majority of the DOS is concerned with such tasks as command interpretation and execution, file management, track and sector mapping, and cataloging. Anytime it actually reads or writes disk information, however, it uses RWTS. There are four types of call to RWTS: FORMAT, READ, WRITE, and SEEK (head position only). **FORMAT** writes identifying information for 16 sectors on all thirty-five tracks. **READ** positions the head and reads a specified track/sector, and **WRITE** positions the head and writes to a specified track/sector. **SEEK** moves the head to a specified track.

Ignoring elaborate copy protection methods, the normal method of **reading** a sector is to position the head and poll the disk input port until the desired identifying leader, or **address field**, is found. Then the following data is read into an area of RAM the size of the sectors being used. Data is written one sector at a time. The DOS **writing** method is to find the pertinent sector on a formatted disk as when reading, but to overwrite the following data area after the desired sector address field has been found.

Figure 9.2 shows the paths data takes during disk I/O. Data is transferred between the MPU and the **data register** on the controller, one byte at a time over the data bus.* Data is shifted serially between

*The data register is referred to in some writings as the data latch.

the controller and the disk drive under control of the **logic state sequencer** on the controller. The logic state sequencer is a ROM and some flip-flops wired up to act like a little 2 MHz computer. It has a stored program which it sequences through while executing commands that control the data register. Some writings refer to the sequencer as the "state machine," but logic state sequencer more accurately describes its functions. Via address bus commands, a 6502 program can configure the sequencer to shift out write data, shift in read data, or shift in the state of the write protect switch in the disk drive.

In the **write process**, a 6502 program causes the MPU to store a byte of data in the data register of the controller. The logic state sequencer shifts this information, monitoring each bit individually. Every time the sequencer sees a one, it toggles the WRITE signal. This changes the direction of magnetic field in the **read/write head**. As the disk passes across the head during write operation, the surface of the disk near the head is magnetized, and the direction of field in the head determines the direction of the magnetic field on the disk. When the writing stops, the data remains on the disk in the form of transitions or lack of transitions in the magnetic field. Disk I/O is identical to cassette I/O in this regard. Serial data is stored in the form of magnetic field reversals on the medium. A ONE is a field reversal. A ZERO is the lack of a field reversal.

Reading is the reverse of writing as far as the read/write head is concerned. As the disk rotates, magnetic field reversals on the moving disk surface cause voltages to be induced in the head. The voltage pulses are sensed by a special purpose floppy disk read interface chip which puts out a nice square read pulse for every magnetic field reversal on the disk. The sequencer monitors these read pulses and shifts ONEs and ZEROs into the data register based on the presence or absence of the read pulse at the normal write interval.

The sequencer syncs up on the read data if it was written properly. This means that it will shift data into the data register until a complete byte is shifted in, then it holds that complete byte long enough for a 6502 program to detect it by polling the data register. The program recognizes that the data register holds a valid byte when the most significant bit of the register is set. When a valid byte is detected, the program must quickly process or store the byte and start checking the data register for the next one.

It can be seen from this overview that the key to understanding how data is transferred to and from the disk is the logic state sequencer and how it is manipulated by the 6502 program. Later, we will analyze the sequencer in great detail, but first we need to lay the groundwork by looking at the hardware environment.

THE DISK II DRIVE

Figure 9.3 is a functional diagram of the Disk II drive. The intention here is not to explain all details of floppy disk drive operation, but only to establish the basis of control of the drive from the computer. In addition to Figure 9.3, reference to Figure 9.1 should help clarify the points of discussion. Even though only the Apple Disk II is spoken of, most of the discussion will also be valid for substitute drives. As Figure 9.1 indicates, all connections to the drive are routed to the controller.

Power Supply

The drive takes its power supply voltages from the Apple's main power supply. +12V, -12V, and -5V are all utilized, but only +12V is used for motor drive. Since +12V is used both to position the head and rotate the disk, the load on +12V is significant, especially at disk start-up. The drive has a high capacity +12V input filter to assist the Apple's power supply in supplying motor start-up current.

The DRIVE ENABLE' Input

The ENABLE' input is low at drive 1 or drive 2, but not both, when a drive is being accessed. At the enabled drive, the drive motor is on, the IN USE indicator glows, head positioning is enabled, and the read pulse output and the state of the write protect switch are enabled to the controller. Speed of disk rotation is regulated by a motor speed control board in the back of the drive. This speed is adjustable via a potentiometer on the speed regulator board.

The Head Positioning Mechanism

The head assembly is positioned precisely, via a stepper motor. The stepper motor, which rotates, is linked mechanically to the head assembly, which travels linearly. A 6502 program positions the head assembly by directly controlling the **four phased inputs** to the stepper motor.

Figure 9.4 is a functional diagram of a four phase stepping motor, which can provide a basis for understanding the positioning of the read/write head. The rotor is a cogged ferrous drum whose cogs may be attracted by one of four electromagnets. The electromagnets are activated sequentially under program control. There may be any number of cogs on the rotor, but only one of them is next to one of the four electromagnets at any time.

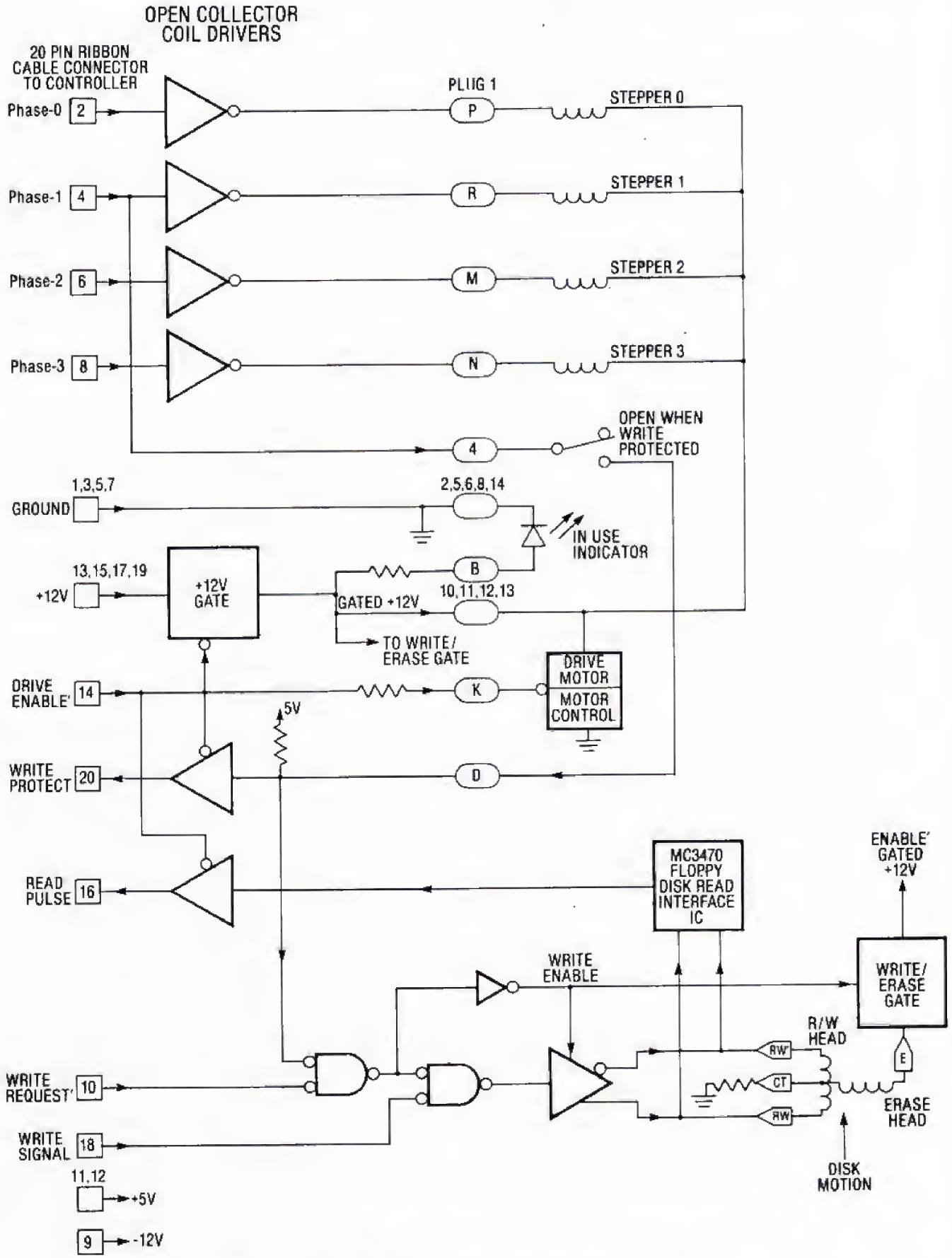


Figure 9.3 Simplified Functional Diagram of Disk II Drive.

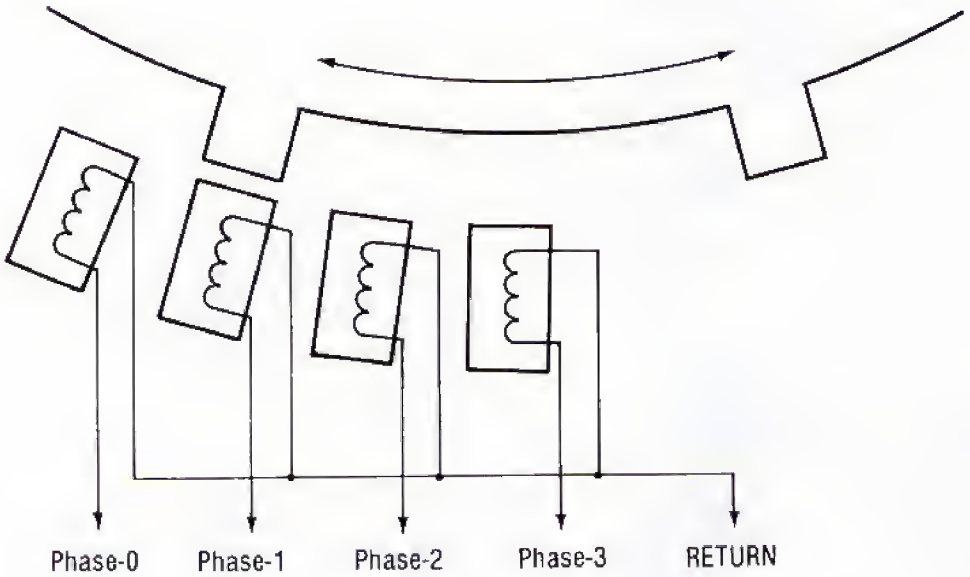


Figure 9.4 A Stepping Motor.

To rotate the rotor one step to the right in Figure 9.4, energize the phase-2 magnet, then deenergize the phase-1 magnet. Magnetic attraction will then align the cog with the phase-2 magnet. To go three steps left from this new position, perform this progression: phase-1 on, phase-2 off, wait, phase-0 on, phase-1 off, wait, phase-3 on, phase-0 off, wait. The **wait** is for the **response time** of the motor, which is slow compared to the MPU. Summarizing, to step left, sequence through the phases in descending order. To step right, sequence in ascending order.

The Apple disk drive uses a four phase stepper motor for head positioning. The controller has provisions for bringing the control voltage for each phase high or low individually. In the drive, these voltages will turn on or cut off current to the electromagnets in the stepper motor. A good deal of the software overhead is required to position the head and remember its location (See **PROGRAMMING EXAMPLES FROM RWTS** later in this chapter). Two phases must be stepped through to position the head one track, and the Disk II and DOS support thirty-five tracks.

Writing to the Disk

There are two write related inputs from the controller to the drive, WRITE REQUEST' and the WRITE signal. **WRITE REQUEST'** causes the drive to be configured for writing unless the disk is write protected. Configuring the drive for writing consists of allowing the **WRITE signal** to control the current in the coil of the read/write head, and of applying a direct current to a second head referred to here as the **erase** head. The WRITE SIGNAL control of the current in the read/write head is such that the high/low state of the WRITE signal determines the direction of the magnetic field set up in the read/write head.

Current in the read/write head tends to produce magnetic fields on the disk parallel to disk motion. Current in the erase head tends to produce magnetic fields on the disk perpendicular to disk motion. The actual strength and direction of the fields produced are a result of the vector sum of the WRITE signal field and the erase field (see Figure 9.5). The presence of the erase field means that there is always some field in the head assembly while writing, even in the middle of a WRITE signal field reversal. The absence of a field in the head assembly would allow previous field alignment on the disk to remain unchanged. Thus, the erase field causes the previously written data to be erased when the WRITE signal component of the field vector has an amplitude of zero.

As was mentioned in the overview, data bits are stored on the disk as field reversals, or the lack of field reversals, at a regular interval. This fact is not changed by the presence of the erase component in

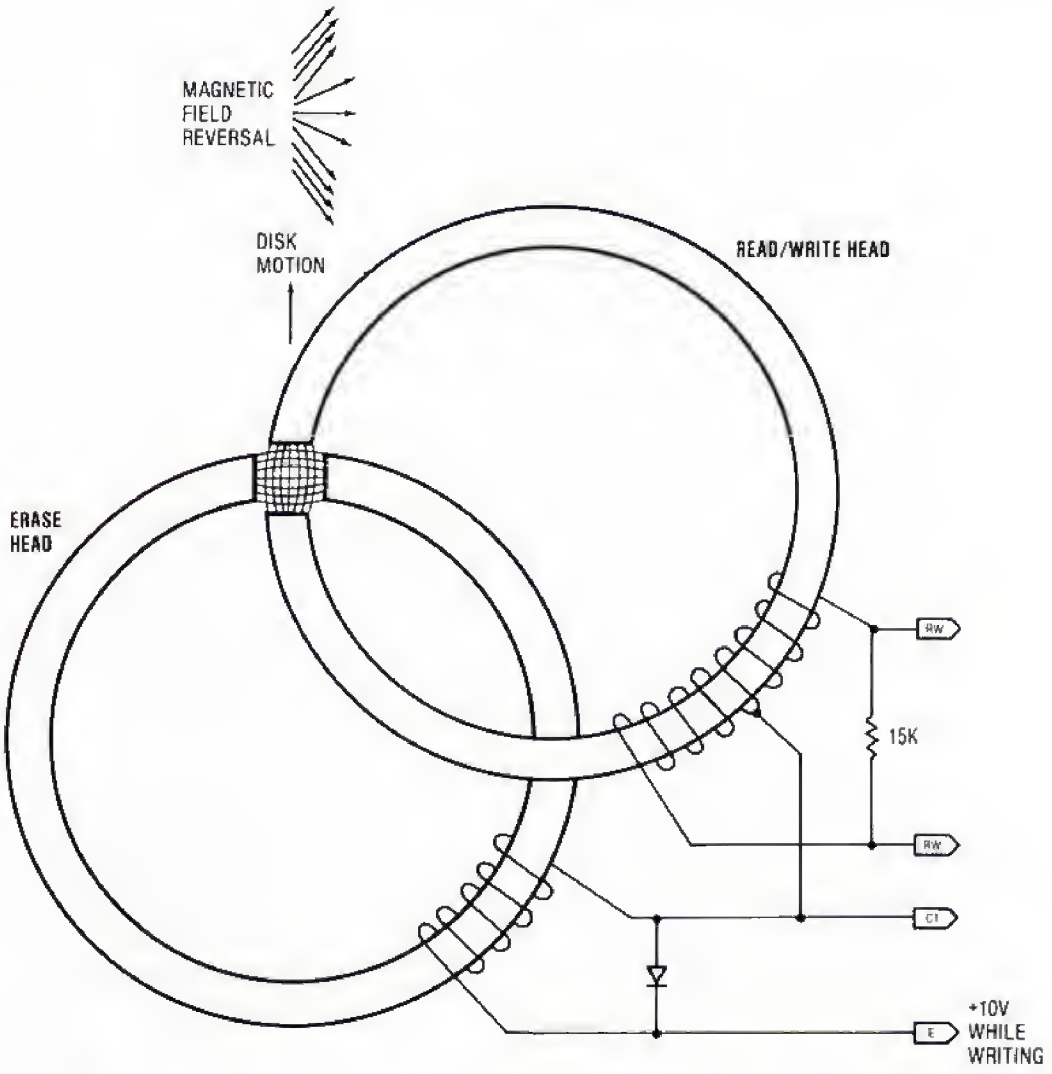


Figure 9.5 The Write Field is the Vector Sum of the WRITE and ERASE Fields.

the field. Just understand that the erase, or radial, component is constant, while the read/write, or tangential, component reverses depending on the WRITE signal.

The Write Protect Switch

There is a spring loaded switch in the drive which is open when a write protected disk is installed. A write protected disk is one with no notch on the left edge or one with the notch covered up. Having this switch open causes the WRITE PROTECT signal to be high, which isolates the WRITE signal from the read/write head even if WRITE REQUEST' is low. The WRITE PROTECT signal is also routed from the enabled drive to the controller so a 6502 program can determine the state of the write protect switch.

The WRITE PROTECT signal and the read pulse are the only two outputs from the drive to the controller (see Figures 9.1 and 9.3). Both are output through tri-state drivers which are gated by the drive ENABLE' input. This allows the two drives to share control of the read pulse and write protect lines.

An interesting feature of the write protect circuitry is that its activating voltage is the phase-1 voltage input to the head positioning motor (see PLUG 1, pin 4 in Figure 9.3). As a result, phase-1 must be turned off after head positioning or writing to the disk is impossible. This is probably a way of forcing the programmer turn off all phases (not just phase-1) after head positioning. This seems to imply that keeping a stepper motor input active causes an

undesirable effect: perhaps a tiny vibration or overheating of the motor. In any case, the RWTS head positioning routines leave all four phases off after head positioning. The boot routine on the controller card, however, leaves phase-0 energized after sending the head to track 0.

The Read Pulse

When writing is not enabled, passing a field reversal on the surface of the disk over the read/write head induces a voltage in the coil. This induced voltage will alternate in polarity for every field reversal. The induced voltage is sensed by a special purpose chip which is designed for this function. The special purpose chip, a Motorola MC3470, outputs a positive one microsecond pulse for every field reversal (see Figure 9.6). This **read pulse** is routed from the enabled drive to the controller.

Now we come to the biggest problem with reading a disk. The signal coming off the read/write head is a dirty little voltage. The shape and size of this **read pick-up signal** will vary with disk speed, temperature, humidity, head alignment, disk warpage, and Murphy's Law in the read environment as opposed to the same factors in the write environment. The MC3470 has to clean up the imperfect read pick-up signal and pass it to the controller. The basic features of the MC3470 clean up job include amplification, shaping, filtering, and noise rejection. The MC3470 detects positive and negative voltage peaks on the read pick-up signal, then waits about two microseconds to verify that a second opposite polarity peak has not occurred. The purpose of this is to prevent narrow noise pulses from generating invalid read pulses. After the two microsecond mask period, a one microsecond read pulse is output. Even with this sophisticated interface, the controller must be able to reliably monitor a read pulse whose timing interval will vary significantly. In addition, it must be able to interpret either the presence or the absence of the read pulse at the distorted interval.

The two microsecond delay period between peak detection and read pulse output turned out to be too short for the new format when Apple came out with DOS 3.3. Apple solved this problem by replacing fixed resistor R21 with a potentiometer on the analog card in the disk drive. Technicians align the potentiometer for an optimum delay, which works out to be approximately three microseconds. Old cards causing unreliable data transfer because of the two microsecond delay are replaced by Apple as if the card were still under warranty. Your dealer will still charge you for labor and alignment in event of such replacement.

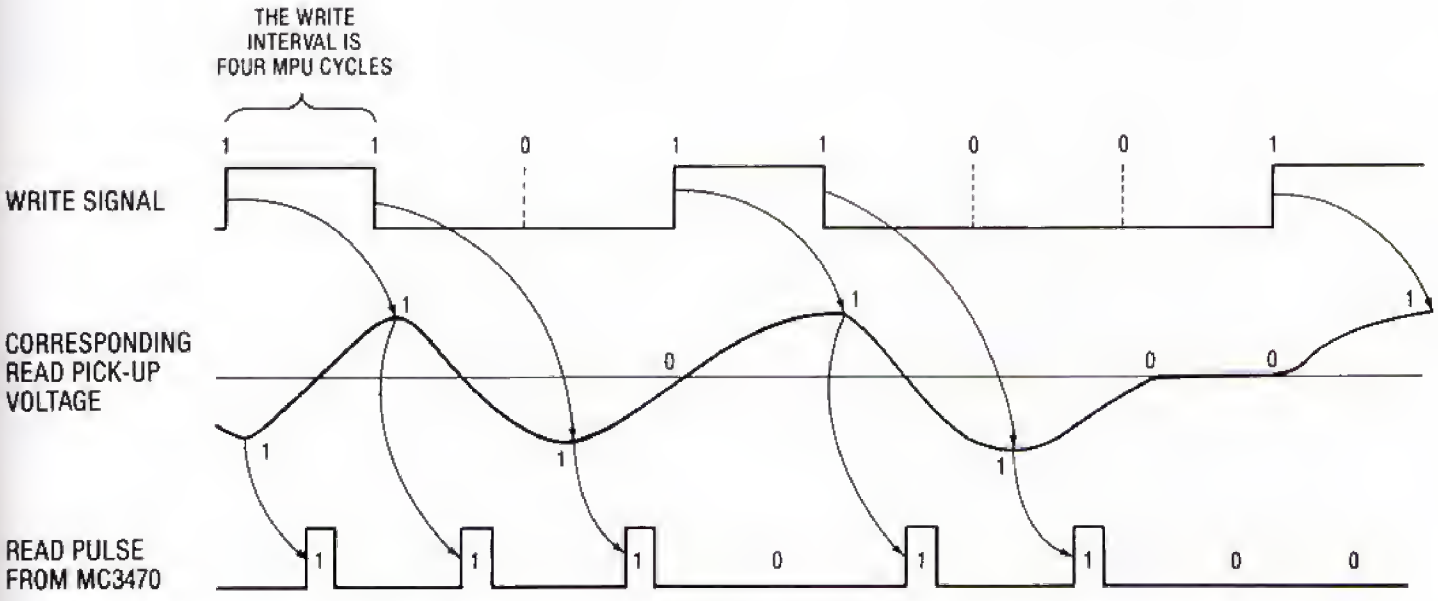


Figure 9.6 When Reading, the Lack of a Read Pulse at a Regular Interval Represents a ZERO.

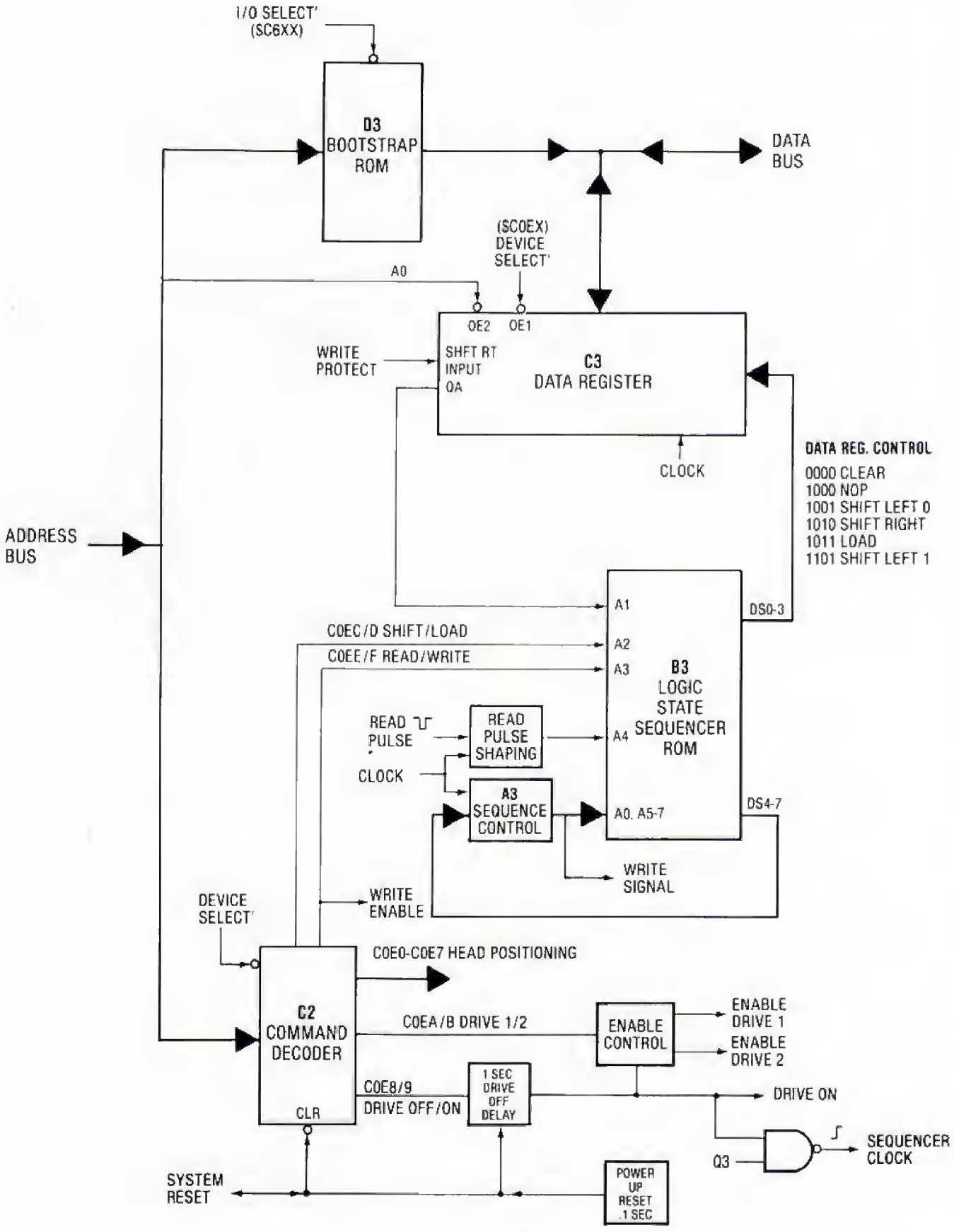


Figure 9.7 Block Diagram of the Disk II Controller.

The combination of the read/write head and the MC3470 responds very well to field reversals on the moving disk as long as there is not too much space between them. However, if there is too much space between field reversals on the disk, the MC3470 starts putting out false read pulses. This means that you can't utilize copy protect schemes that call for isolated field reversals on the disk separated by large intervals of constant field direction. In other words, the MC3470 will reliably produce read pulses while data of normal density is moving past the read/write head, but reduction of this density will cause spurious read pulses to be generated. The write interval in the Apple is four MPU cycles, and we will see that in the DOS formats the maximum time between field reversals on a disk will be three write intervals.

THE DISK II CONTROLLER

The Disk II controller contains that part of disk I/O electronics which needs to be positioned close to the motherboard. This includes a Bootstrap ROM, a data register, a logic state sequencer, and a command decoder. Figure 9.7 shows the functional flow of the controller, and Figure 9.8 is a full schematic.

The Bootstrap ROM

The **Bootstrap ROM** is referred to as the P5 ROM by Apple. It contains a 256 byte program which begins the bootstrap procedure that reads the DOS from a disk, puts it into RAM, and initializes it. This 256 byte program is the only 6502 program which resides on the controller, and it may be accessed any time at CnXX where n is the controller slot number. This program has just enough code to get the contents of track 0, sector 0 into RAM. Then program control is passed to that part of RAM so that the bootstrap procedure can be continued. The Bootstrap ROM is connected to the address bus and data bus naturally, and its output enable is the I/O SELECT' input to the slot. This response to the I/O SELECT' input causes the DOS to be booted when PR#n or IN#n is executed from BASIC.*

The Autostart power-up RESET routine uses the contents of the Bootstrap ROM to detect the presence of a Disk II controller in the Apple. It does this by starting with slot 7 and working downward, checking each slot for the presence of $20, $00, $03, and $3C at locations $Cn01, $Cn03, $Cn05, and $Cn07. When it finds this combination, it starts executing at $Cn00 on that slot, thus booting the DOS.

The Command Decoder

The overseeing 6502 program manages the controller via address bus commands in the DEVICE SELECT' range of the controller's slot ($C080,X through $C08F,X with slot number times $10 in the X-register). This range is divided up into eight on/off switches by an LS259 on the controller identically to the way the $C05X range on the motherboard is divided up (see the F14 chip in Figure 7.2). In other words, there are eight on/off soft switches by which a program can manage disk I/O. Like the motherboard screen mode switches and annunciators, the disk switches are set OFF by even addresses and ON by odd addresses. Table 9.1 is a brief listing of the functions of these address decoded commands, but more detailed explanations follow.

*See Chapter 7, **PERIPHERAL SLOT CONNECTIONS** and **THE APPLE I/O SYSTEM: KSW AND CSW**.

Table 9.1 Disk II Controller Commands.

SWITCH	OFF FUNCTION	ON FUNCTION
Q0	$C080,X - PHASE 0 OFF	$C081,X - PHASE 0 ON
Q1	$C082,X - PHASE 1 OFF	$C083,X - PHASE 1 ON
Q2	$C084,X - PHASE 2 OFF	$C085,X - PHASE 2 ON
Q3	$C086,X - PHASE 3 OFF	$C087,X - PHASE 3 ON
Q4	$C088,X - DRIVES OFF	$C089,X - SELECTED DRIVE ON
Q5	$C08A,X - SELECT DRIVE 1	$C08B,X - SELECT DRIVE 2
Q6	$C08C,X - SHIFT WHILE WRITING/ READ DATA	$C08D,X - LOAD WHILE WRITING/ READ WRITE PROTECT
Q7	$C08E,X - READ	$C08F,X - WRITE

1. RESET' forces all switches off.
2. Access to even addresses causes the data register contents to be transferred to the data bus.

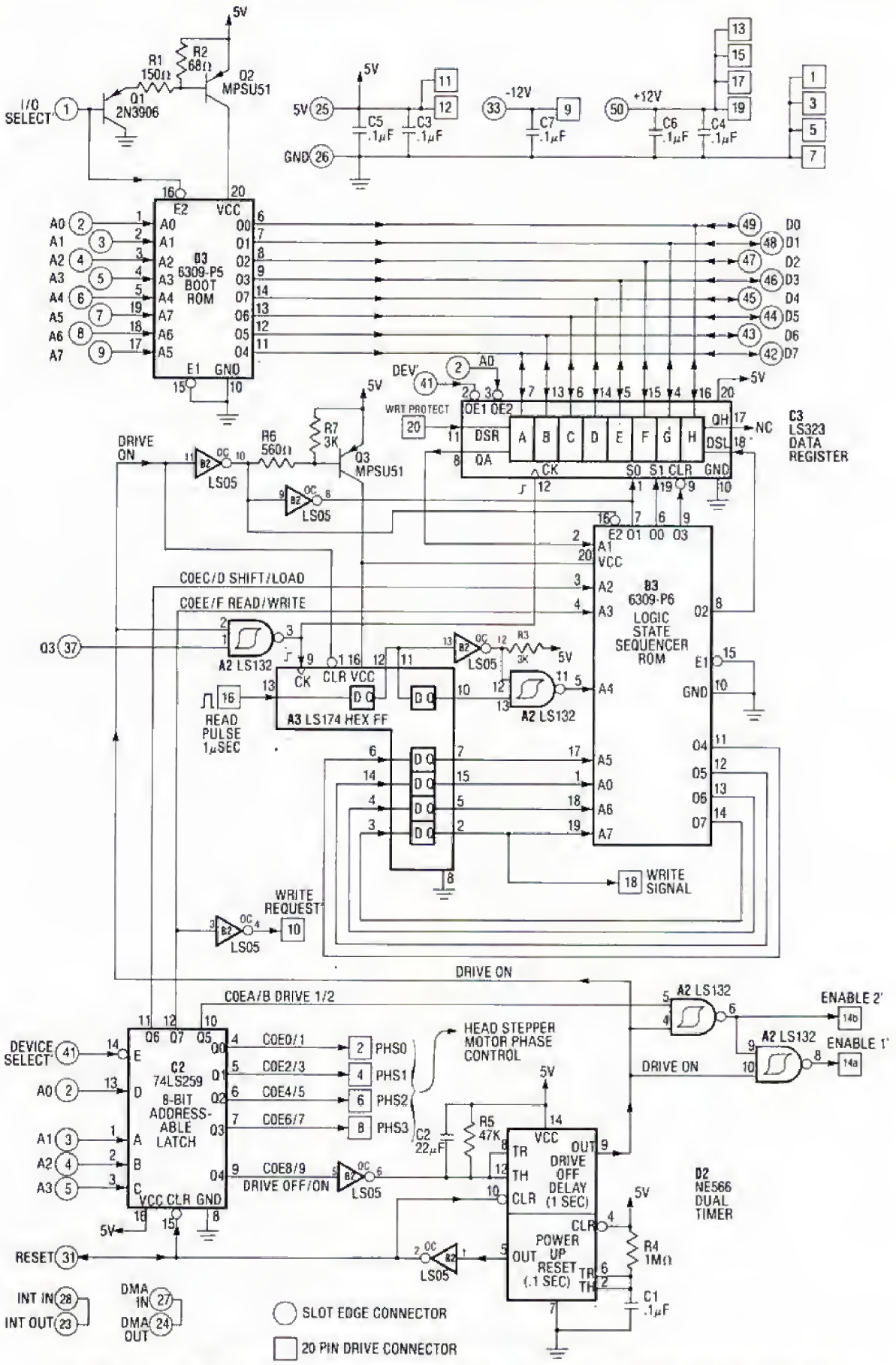


Figure 9.8 Schematic: The Disk II Controller (Address References Assume Slot 6).

Drive Off/On and Drive Select

The $C088,X/$C089,X switch disables the drives or enables a drive, while the $C08A,X/$C08B,X switch selects drive 1 or drive 2 for enabling. Here is an illustrative, but otherwise useless program sequence:

```
LDX  #$60       ;SLOT 6
CMP  $C08B,X    ;SELECT DRIVE 2
CMP  $C089,X    ;DRIVE 2 ON
CMP  $C08A,X    ;DRIVE 2 OFF
                ;DRIVE 1 ON
CMP  $C088,X    ;DRIVES OFF
```

Turning a drive on ($C089,X) performs the following at the controller:

1. Applies +5V power to the sequencer ROM and the sequencer flip-flops at A3.
2. Applies Q3' to the clockpulse inputs of the data register and the sequencer control flip-flops. The sequencing and data transfer clock is Q3 falling. (DRIVES OFF forces the control flip-flops to clear.)
3. Enables the outputs of the sequencer ROM.
4. Enables sequencer control of the data register. (DRIVES OFF forces the data register to hold its present state.)
5. Causes the ENABLE 1' or the ENABLE 2' signal to go low depending on which drive is selected by the drive 1/drive 2 switch. At whichever drive is enabled, this turns on the drive motor and IN USE indicator, and it enables head positioning, writing, and control of the read pulse and write protect inputs to the controller.

The drive off/on signal is routed through one half of an NE556 timer. The effect of this timer is to delay drive turn-off until one second after a reference to $C088,X. This gives the drive apparent momentum, keeping it running after it is turned off. The result is that the drive never turns off between closely spaced accesses, and in these instances, access time is reduced because there is no delay while the disk comes up to speed. This is why the computer is ready to process after a "CATALOG" well before the IN USE light goes out on the drive. The one second turn off delay does not apply to turning off the drive via a RESET. Pressing RESET causes the delay timer to clear and turns off the drive almost immediately.

The second half of the NE556 timer is used to generate a .1 second **power-up RESET**. This is overridden by the motherboard .3 second power-up RESET on Revision 1 and later boards, but on Revision 0 boards, the controller generated RESET is necessary to achieve the autostart capability.

Head Positioning Commands

Address bus commands $C080,X through $C087,X are translated directly into four phase-off/phase-on stepper motor controls. These control voltages are routed to the drives where they are amplified and applied to the head positioning motor. Ascending references to $C080,X through $C087,X cause the head assembly on the enabled drive to move toward the inner track (track 34). Descending references cause movement toward track 0. The controlling program must wait approximately 20 milliseconds for motor response after turning a phase on. The actual motor response will vary with momentum, and the RWTS head positioning routine reduces positioning time by varying the waiting period with expected momentum.

The motor must be stepped **two phases per track**, so there are really 70 head positions. RWTS writes at the phase-0 aligned and the phase-2 aligned positions, but copy protected disks may have data written on the **half-tracks**, the phase-1 aligned and phase-3 aligned positions. The head assembly has a **mechanical stop**—an electrical stop in some alternate drives—at track 0. One method to absolutely ascertain the head position is to step outward eighty phases as the bootstrap program does. The head will run against the track 0 stop, and you will be at track 0. The head/stepper motor linkage is aligned so that the motor will be **phase-0 aligned at track 0**, so from track 0 it is known that stepping inward must be begun by turning phase-1 on by a reference to $C083,X. From this point, the head position can be stored in RAM, and the phase alignment can be determined from the head position. After head positioning, the four phases should all be turned off, because, if phase-1 is left on, the drive will behave as if a write protected disk is installed.

Read/Write

$C08E,X/$C08F,X is the controller's READ/WRITE soft switch. It is an addressing input to the sequencer and divides the sequencer into its two most significant parts, the **READ sequence** and the **WRITE sequence**. It also is inverted to become the WRITE REQUEST' signal to the drives. Thus, the READ/WRITE switch configures the controller

for reading or writing via sequencer addressing, and it configures the enabled drive for writing via the WRITE REQUEST' line unless a write protected disk is installed.

SHIFT/LOAD

SHIFT/LOAD is a fairly inadequate label for the $C08C,X/$C08D,X soft switch, but any label would be. It is chosen because, during writing, $C08C,X causes shifting of the data register on every eighth sequencer clock, and $C08D,X causes loading of the data register from the data bus on every eighth sequencer clock. In reality, the SHIFT/LOAD switch performs several functions which defy summarization in a short label.

SHIFT/LOAD is an addressing input to the sequencer and it divides both the READ sequence and the WRITE sequence into two parts. As mentioned above, it is a programmable SHIFT/LOAD control for the data register. When the READ/-WRITE switch is low, the SHIFT/LOAD function is changed to READ/CHECK WRITE PROTECT. These sequencer control functions are summarized in Table 9.2. The operation should become clearer when we study the sequencer listings.

$C08C,X and $C08D,X are also the normal input and output port addresses used by RWTS for transfer of disk data. In reality any even address could be used to load data from the data register to the MPU, although $C088 (DRIVES OFF) and $C08A (SELECT DRIVE 1) would be inappropriate for this purpose. Use of $C08D,X as the output address goes hand in hand with the fact that it causes loading of the data register from the data bus.

The Logic State Sequencer and Data Register

As mentioned before, the **logic state sequencer** is a ROM wired up to behave like a little 2 MHz computer. It is this powerful sequencer that enables the controller to perform such complex control with so few chips. It uses a 256 byte ROM with four of its data outputs (O4-O7) connected through flip-flops to four of its address inputs (A5, A0, A6, and A7). The flip-flops are clocked by **Q3 falling** while a drive is enabled.*

There are eight address inputs to the sequencer ROM, so let's refer to the four flip-flop latched end around inputs as the **sequencing inputs** and the other four address bits the **partitioning inputs**. Assume for a moment that the four partitioning inputs are fixed. The overall ROM address will then change every time Q3 falls, and the contents of bits O4-O7 of any addressed location will determine the address after the next clockpulse. These four data bits thus contain flow information, and they are programmed so the flow will proceed in an orderly manner from clock to clock. There are 16 states of the four sequencing address inputs and 16 states of the four partitioning address inputs. The sequencer ROM is thus divided into 16 partitions of 16 sequencer states each.

The four partitioning inputs are:

A1 - QA, the MSB of the data register
A2 - SHIFT/LOAD, the $C08C,X/$C08D,X switch
A3 - READ/WRITE, the $C08E,X/$C08F,X switch
A4 - The read pulse from the disk drive

The A2 and A3 inputs allow the 6502 programmer to configure the sequencer for loading while writing, shifting while writing, reading data from the disk, or checking the write protect switch. The A1 and A4 inputs allow the sequencer flow to deviate

*The sequencer clock is actually the rising edge of Q3', developed by gating Q3 through a 74LS132 NAND gate (see Figure 9.8). Due to propagation delay, the clock is effectively Q3 falling plus approximately 15 nanoseconds.

Table 9.2 Functions of the $C08C,X/$C08D,X and $C08E/$C08F Switch.

SHIFT/LOAD	READ/WRITE	Sequencer Function
$C08C,X	$C08F,X	Data register shift every eighth sequencer clock while writing.
$C08D,X	$C08F,X	Data register load every eighth sequencer clock while writing.
$C08C,X	$C08E,X	Enable READ sequencing.
$C08D,X	$C08E,X	Check state of write protect switch and initialize sequencer for writing.

depending on the presence or absence of a read pulse and depending on the state of shifted data in the data register.

The **data register** is a very versatile 8-bit IC that can shift left, shift right, load parallel or store parallel based on its control inputs. The remaining four outputs of the sequencer ROM are connected to inputs of the data register, which completes our picture of the sequencer. Four of the ROM data bits are programmed to control sequencing and the other four data bits are programmed to control the data register. There are only six distinct commands which the sequencer can cause the data register to perform, but there are 16 possible states of the command bits. This is because there are redundant states which command the data register to do the same thing. Table 9.3 shows the 16-bit states in hexadecimal and binary with an asterisk next to the six states used by Apple to perform commands in their 3.2 and 3.3 ROMs.

In addition to the sequencer control of the data register, any reference to even addresses in the DEVICE SELECT' range of the controller slot will cause the contents of the data register to be placed on the data bus. For this reason, programs should not cause the MPU to write to even addresses or the MPU bidirectional driver will compete with the data register for control of the data bus. It is important to note that while a 6502 program can check the state of the data register at any time, a program can store data to the data register only when the sequencer is performing a load. As a result, the write operation involves getting the 6502 program in sync with the sequencer and keeping it there by performing write operations in exact timing loops. If the data register is to accept data from the MPU, the 6502 program must store the data to the data register at **exact multiples of the bit writing internal.** This interval is eight cycles of the sequencer or **four cycles of the MPU.**

It is important to make a distinction here between what **can** be done and what normally **is** done. A program can store data at any multiple of the bit writing interval—8, 12, 16, 20, etc. MPU cycles—and the data register will accept it. The DOS, however, only stores data to the data register at 32, 36, and 40 cycle intervals. There are very practical reasons for this which will become apparent as these discussions progress.

Both the sequencer control flip-flops and the data register are clocked by Q3 falling when a drive is enabled. This means the sequencer operates at approximately 2 MHz, twice the frequency of the 6502. Additionally, the read pulse from the enabled drive is synchronized to the Q3 falling clock, quantized in pulse width to one Q3 period, and inverted in the process to form a negative pulse. It is this synchronized, negative read pulse which is applied to

Table 9.3 Logic State Sequencer Command.

07-06-05-04 HEX	BINARY	MNEMONIC	FUNCTION
***0**	**0000**	**CLR**	CLEAR DATA REGISTER
1	0001	CLR	
2	0010	CLR	
3	0011	CLR	
4	0100	CLR	
5	0101	CLR	
6	0110	CLR	
7	0111	CLR	
***8**	**1000**	**NOP**	NO OPERATION
***9**	**1001**	**SL0**	SHIFT ZERO LEFT INTO DATA REGISTER
***A**	**1010**	**SR**	SHIFT WRITE PROTECT SIGNAL RIGHT INTO DATA REGISTER
***B**	**1011**	**LD**	LOAD DATA REGISTER FROM DATA BUS
C	1100	NOP	
***D**	**1101**	**SL1**	SHIFT ONE LEFT INTO DATA REGISTER
E	1110	SR	
F	1111	LD	

*These are the codes actually used by Apple.

A4 of the sequencer ROM, and the read pulse is therefore monitored in the same 500 nanosecond time frame as the other addressing inputs to the ROM. In a further attempt to stabilize the shaky read pulse, a Schmidt trigger type NAND gate is used in the read pulse quantizing circuit (see Figure 9.8). Use of this type of gate increases noise immunity and ensures smooth transitions of the A4 input to the sequencer ROM.

During both reading and writing, the data register is **shifted left** while the most significant bit, QA, is monitored. In writing, the state of QA is monitored and the WRITE SIGNAL is toggled at the bit writing interval when QA is set. In reading, ONEs and ZEROs are shifted into the data register depending on the presence or absence of a read pulse at the bit writing interval. When QA becomes set, the sequencer holds the data register long enough for a 6502 program to detect the valid byte with a seven MPU cycle polling loop. Then the data register is cleared and the next byte is shifted in from the disk.

Decoding the Contents of the Sequencer ROM

In the past, the contents of the logic state sequencer ROM have been a mystery in the world of Apple users. The basic reason for this is that no one who understood the contents ever bothered to publish any information about them. A primary aim of this chapter is to fill this gap in Apple literature by providing formatted listings of the DOS 3.2 and DOS 3.3 sequencer ROM contents and discussing operational aspects of disk I/O which are determined by the contents.

This section shows how to make the contents of the sequencer ROM accessible to the MPU and provides a program which makes formatted listings of those contents. The program will accurately list the contents of any ROM designed to operate in the B3 socket of the Disk II controller. It is not necessary for the reader to go through the exercise of making listings of the DOS 3.2 and DOS 3.3 sequences since these listings are furnished in Figures 9.10 and 9.11. He may, however, find it interesting and educational to make his own listings.

The MPU cannot normally read the contents of the sequencer ROM because the ROM is not connected to the data bus. You can change this situation by removing the bootstrap ROM from the D3 socket and moving the sequencer ROM from the B3 socket to the D3 socket. The contents of the sequencer ROM can then be read by the MPU by addressing $C6XX (assumes Slot 6). Of course, your disk drive won't work with the controller configured this way, but you can save the data to cassette tape and then transfer it to a disk file when your controller is back to normal.

This data can be listed to a printer using the monitor, but it is not particularly readable in this form. I have written a BASIC program to format the data into a readable listing. Figures 9.9 through 9.11 contain listings of the Applesoft BASIC formatting program, the DOS 3.2 sequencer, and the DOS 3.3 sequencer respectively. The program was written in BASIC rather than assembly language so it could be more easily understood by readers who choose to study it. Also, readers should find it very easy to manipulate the sequencer listing because it is formatted as a 16 by 16 BASIC subscripted string variable. The program takes a little over a minute to run. It will list any sequencer ROM designed to run in the Disk II controller as long as its source file is read from the D3 socket of the controller. The source file needs to be resident on diskette as a binary file named "SEQROM." However, it would be easy to change line 50 of the program so the source file was obtained elsewhere. Before running the program, enable your 80 column printer or 80 column video card.

The manipulations of the program are based on the following features of controller wiring:

1. Address bus A5 is connected to the A7 input to the D3 socket. Address bus A7 is connected to the A5 input to the D3 socket.
2. Data bus D4 through D7 is connected to outputs O7 through O4 on the D3 socket in reverse order.
3. A natural significance order of addressing inputs to the sequencer ROM is WRITE—READ PULSE—SHIFT/LOAD—QA—O7—O6—O5—O4. This does not correspond to the way they are connected to A7 through A0 on the sequencer ROM.
4. The read pulse applied to A4 of the sequencer ROM is a negative pulse.

These wiring connections were not designed to confuse us, although they serve the purpose. They were designed to minimize wire cross over on the mechanical layout. A design engineer can swap the address and data pin assignments on a ROM until he finds his own version of peace of mind, then he compensates for the scrambled pin assignments when

he writes the ROM program.* The sequencer ROM formatting program must account for this by reading each byte of data, unscrambling address bits to find where that data is in the sequence, and reversing data bits before storing them in a 16 by 16 string variable matrix from which listings can easily be made.

The result is the listings of Figures 9.10 and 9.11.** The WRITE and READ sequences are separated from each other, and the listing is otherwise divided into columns of 16 sequencer states. This is a natural division since 16 different values can be represented by the four sequencing data bits. The far left hand column is the sequencer state, and the other columns hold the contents of the ROM for that sequencer state. **The left digit of each number is the next sequencer state, and the right digit is the command.** Next to each number is a mnemonic for the command digit.

As a quick example of interpreting this listing, assume that the sequencer is at the top of the fifth column of the WRITE sequence in Figure 9.10. This means the sequencer is at State 0 and the partitioning address bits are WRITE—NO READ PULSE—SHIFT—QA'. The data being driven out of the sequencer ROM is 18. **The 1 is the next sequencer state** which will occur when Q3 falls. **The 8 is the command which will be executed** when Q3 falls, a NOP. A quick scan through the WRITE listing will show that it contains mostly NOPs and that it normally flows from one sequencer state to the next.

The WRITE Sequence

The WRITE sequence and WRITE PROTECT sequence are the same in the 3.2 and 3.3 controllers, so this section pertains to both. Please refer to the 3.3 listing during these discussions, using the 3.2 listing for comparison.

*Since writing this paragraph, I have learned that Steve Wozniak designed the Disk II controller. Working with Randy Wigginton, Wozniak completed this complicated and innovative design in the space of one week (the final week of 1977). Steve is very proud of the mechanical layout and at one time redid the layout for the sole purpose of reducing the number of feedthrough holes from three to two. I had always assumed that much of Apple's disk interface technology had been borrowed from Shugart, but I couldn't have been further from the truth. The format and circuitry represent a notable creative effort by the Apple group. Besides the controller design, Wozniak wrote RWTS. Randy Wigginton wrote the rest of the original DOS, and Rod Holt designed the analog board of the disk drive.

**The formatted listings presented here are my own representations which resulted from lengthy investigation. The mnemonics, address labels, and layout won't, therefore, be the same as those used by Apple Computer engineers. Who knows what they use? I submit, though, that my representation provides adequate illustration in the absence of any labels and formats published by Apple.

We begin our analysis by making two simplifying observations. First, if WRITE is selected at the READ/WRITE switch ($C08F,X), the left four columns (READ PULSE) are identical to the right four columns (NO READ PULSE). This means that read data has no effect on the WRITE sequence and it can be ignored for purposes of writing. This is a necessity because meaningless read pulses will normally be present while writing. Second, all entries of the READ—LOAD sequence states are 0A. This means that READ—LOAD sets the sequencer state to 0 and idly shifts the state of the write protect switch **right** into the data register where it can by checked by a 6502 program. Thus we have the basis for checking if a disk is write protected, namely:

```
LDA $C08D,X ;CHECK WRITE PROTECT
            ;IF READ.
LDA $C08E,X ;READ.
BMI ERROR   ;BRANCH TO WRITE PROTECT
            ;ROUTINE IF QA SET.
```

These are the program steps used by RWTS to check for write protection before writing a sector. A routine such as this must be performed every time before writing. It not only checks if a disk is write protected, it also clears QA if the disk is not write protected and sets the sequencer to State 0. This initializes the sequencer for writing, and it is the only way an MPU program can establish the state of the sequencer.

In the above program steps, the data register was checked via a LDA $C08E,X. This sets READ/WRITE to READ, shifts the state of the write protect switch into QA of the data register, and places this shifted value on the data bus so the MPU can load it to its accumulator. Now, in reality, any time you would run this program, the READ/WRITE switch will already be set to READ, because we are preparing to write. If READ/WRITE were set to WRITE and a disk with no write protect were rotating, the logic state sequencer would be merrily stomping the flux out of all the data on the track. When you write to the disk, the program waits until the right moment, switches READ/WRITE to WRITE, then stores a few hundred bytes of data to the data register in precision loops, then switches back to READ. So, since the READ/WRITE switch was already in the READ above, the "LDA $C08D,X" is not intended to switch READ/WRITE to READ. It is intended solely to load the contents of the data register from the only appropriate even controller address $C08E,X.

```
10  REM
12  REM              LIST LOGIC STATE SEQUENCER ROM
13  REM
14  REM                        BY JIM SATHER
15  REM                           2/22/83
16  REM
30  REM  THIS PROGRAM FORMATS AND LISTS THE PROGRAM CONTAINED IN THE
32  REM  DISK II LOGIC STATE SEQUENCER.  BEFORE RUNNING THIS PROGRAM,
33  REM  YOU MUST CREATE A BINARY SOURCE FILE ON A DISK AND
34  REM  NAME IT "SEQROM".  THE SOURCE FILE IS CREATED BY PLACING
36  REM  THE B3 ROM IN THE D3 SLOT OF THE DISK CONTROLLER.  THE
38  REM  SOURCE FILE IS WRITTEN TO CASSETTE FROM $C600-$C6FF.  WHEN THE
40  REM  CONTROLLER IS RESTORED, THE CASSETTE FILE IS TRANSFERRED TO DISK.
42  REM
50  PRINT  CHR$ (4);"BLOAD SEQROM,A7936"
100  DIM LST(15,15): DIM ASWAP(7): DIM DFIX(15): DIM HEX$(15): DIM CMND$(15)
110  FOR X = 0 TO 7: READ ASWAP(X): NEXT
120  FOR X = 0 TO 15: READ DFIX(X): NEXT
130  FOR X = 0 TO 15: READ HEX$(X): NEXT
140  FOR X = 0 TO 15: READ CMND$(X): NEXT
150  GR : COLOR= 6: REM SOMETHING TO WATCH WHILE WAITING
200  REM
201  REM  FORMAT DATA INTO 16 X 16 MATRIX
202  REM
210  FOR COL = 0 TO 15: FOR ROW = 0 TO 15:WRK = ROW + 16 * COL
300  REM
301  REM  GET BINARY FORM OF WRK
302  REM
305  FOR X = 0 TO 7:A(X) = 0
310 WRK% = WRK / 2:WRK = WRK / 2: IF (WRK - WRK%) THEN A(X) = 1
320 WRK = WRK%: NEXT X
400  REM
401  REM  RECONSTRUCT DECIMAL WRK WITH ADDRESS BITS SWAPPED:
402  REM  A7-A6-A5-A4-A3-A2-A1-A0 ---> A0-A2-A3-A6-A7-A5-A4-A1
403  REM  THE DATA WILL THEN BE ADDRESSED BY THE WORD:
404  REM  WRITE-READ PULSE-SHIFT/LOAD-QA-07-06-05-04.
405  REM
408 POWEROF2 = 1
410  FOR X = 0 TO 7: IF A(ASWAP(X)) THEN WRK% = WRK% + POWEROF2
420 POWEROF2 = POWEROF2 * 2: NEXT
500  REM
502  REM  SWAP BITS D7-D4 OF DATA BEFORE SAVING, BECAUSE THESE
503  REM  BITS ARE SWAPPED ON THE D3 ROM.
504  REM
507 DTA =  PEEK (WRK% + 7936):HI =  INT (DTA / 16)
510 LST(COL,ROW) = 16 * (DFIX(HI) + DTA / 16 - HI)
520  PLOT ROW,COL: NEXT ROW: NEXT COL: TEXT
600  REM
601  REM
602  REM  ADDRESS SWAP TABLE
610  DATA   1,4,5,7,6,3,2,0
620  REM
630  REM  DATA SWAP TABLE
640  DATA  0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15
650  REM
660  REM  DECIMAL TO HEX CONVERSION TABLE
670  DATA  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
680  REM
690  REM  SEQUENCER COMMAND TABLE
695  DATA  -CLR,-CLR,-CLR,-CLR,-CLR,-CLR,-CLR,-CLR
700  DATA  -NOP,-SL0,-SR ,-LD ,-NOP,-SL1,-SR ,-LD
```

Figure 9.9 BASIC Listing: Program to List State Sequencer ROM. (1 of 2)

```
800  REM
801  REM
802  REM  AT THIS POINT IN THE PROGRAM, THE SEQUENCER LISTING
810  REM  RESIDES IN 16 COLUMNS OF 16 SEQUENCES.  THESE COLUMNS
820  REM  ARE ARRANGED SO EACH COLUMN CONTAINS THE COMPLETE SEQUENCE
830  REM  FOR A STATE OF THE ADDRESS WORD:
840  REM  WRITE-READ PULSE-SHIFT/LOAD-QA.
900  REM
901  REM  *********      LIST WRITE SEQUENCE
902  REM
910  GOSUB 2000: GOSUB 2100: GOSUB 2200: GOSUB 2300: REM  PRINT WRITE HEADING.
920  FOR ROW = 0 TO 15: PRINT HEX$(ROW);"-";
930  FOR COL = 8 TO 15
940 HI% = LST(COL,ROW) / 16:LO% = (LST(COL,ROW) / 16 - HI%) * 16
950  PRINT "   ";HEX$(HI%);HEX$(LO%);CMND$(LO%);
960  NEXT COL: PRINT : NEXT ROW
999  GET B$: PRINT : PRINT
1000  REM
1001  REM  *********      LIST READ SEQUENCE
1002  REM
1010  REM  THE READ PROGRAM IS EASIER TO UNDERSTAND WHEN THE
1020  REM  LOAD PORTION IS SEPARATED FROM THE SHIFT PORTION AND
1024  REM  WHEN QA' (QA LOW) IS SEPERATED FROM QA.  THEREFORE
1030  REM  THE COLUMN ORDER IS CHANGED IN THE READ LISTING.
1040  REM
1045  REM  COLUMN ORDER DATA FOR READ LISTING
1050  DATA  0,4,1,5,2,6,3,7
1055  IF FLAG THEN 1070
1060  FOR X = 0 TO 7: READ CSWAP(X): NEXT :FLAG = 1
1070  REM
1080  GOSUB 2400: GOSUB 2500: GOSUB 2600: GOSUB 2700: REM PRINT READ HEADING.
1090  FOR ROW = 0 TO 15: PRINT HEX$(ROW);"-";
1100  FOR COL = 0 TO 7
1110 HI% = LST(CSWAP(COL),ROW) / 16:LO% = (LST(CSWAP(COL),ROW) / 16 - HI%) * 16
1120  PRINT "   ";HEX$(HI%);HEX$(LO%);CMND$(LO%);
1130  NEXT COL: PRINT : NEXT ROW: PRINT
1160  GET B$: PRINT : PRINT
1170  PRINT "CODE MNEMONIC  FUNCTION"
1180  PRINT " 0     CLR     CLEAR DATA REGISTER"
1190  PRINT " 8     NOP     NO OPERATION"
1200  PRINT " 9     SL0     SHIFT ZERO LEFT INTO DATA REGISTER"
1210  PRINT " A     SR      SHIFT WRITE PROTECT SIGNAL RIGHT INTO DATA REGISTER"
1220  PRINT " B     LD      LOAD DATA REGISTER FROM DATA BUS"
1230  PRINT " D     SL1     SHIFT ONE LEFT INTO DATA REGISTER"
1240  END
1990  REM
1992  REM
2000  PRINT : PRINT ; SPC( 37);"WRITE": PRINT : RETURN
2100  PRINT "    *-----------READ PULSE-------------*-----------NO READ PULSE----------*"
2110  RETURN
2200  PRINT "    *-----SHIFT------*------LOAD-------*------SHIFT------*------LOAD------*"
2210  RETURN
2300  PRINT "SEQ *--QA'--*---QA---*--QA'---*---QA---*--QA'---*---QA---*--QA'---*---QA--*"
2310  PRINT : RETURN
2400  PRINT ; SPC( 37);"READ": PRINT : RETURN
2500  PRINT "    *--------------SHIFT---------------*---------------LOAD--------------*"
2510  RETURN
2600  PRINT "    *------QA'--------*--------QA---------*--------QA'--------*--------QA-------*"
2610  RETURN
2700  PRINT "SEQ *--RP---*-NO RP--*---RP---*-NO RP--*---RP---*-NO RP--*---RP---*-NO RP-*"
2710  PRINT : RETURN
```

Figure 9.9 BASIC Listing: Program to List State Sequencer ROM. (2 of 2)

WRITE

SEQ	READ PULSE SHIFT QA'	READ PULSE SHIFT QA	READ PULSE LOAD QA'	READ PULSE LOAD QA	NO READ PULSE SHIFT QA'	NO READ PULSE SHIFT QA	NO READ PULSE LOAD QA'	NO READ PULSE LOAD QA
0-	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP
1-	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP
2-	39-SL0	39-SL0	3B-LD	3B-LD	39-SL0	39-SL0	3B-LD	3B-LD
3-	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP
4-	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP
5-	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP
6-	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP
7-	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP
8-	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP
9-	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP
A-	B9-SL0	B9-SL0	BB-LD	BB-LD	B9-SL0	B9-SL0	BB-LD	BB-LD
B-	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP
C-	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP
D-	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP
E-	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP
F-	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP

READ

SEQ	SHIFT QA' RP	SHIFT QA' NO RP	SHIFT QA RP	SHIFT QA NO RP	LOAD QA' RP	LOAD QA' NO RP	LOAD QA RP	LOAD QA NO RP
0-	D8-NOP	18-NOP	18-NOP	08-NOP	0A-SR	0A-SR	0A-SR	0A-SR
1-	D8-NOP	28-NOP	28-NOP	28-NOP	0A-SR	0A-SR	0A-SR	0A-SR
2-	D8-NOP	38-NOP	38-NOP	38-NOP	0A-SR	0A-SR	0A-SR	0A-SR
3-	D8-NOP	48-NOP	D8-NOP	48-NOP	0A-SR	0A-SR	0A-SR	0A-SR
4-	D8-NOP	58-NOP	D8-NOP	58-NOP	0A-SR	0A-SR	0A-SR	0A-SR
5-	D8-NOP	68-NOP	D8-NOP	68-NOP	0A-SR	0A-SR	0A-SR	0A-SR
6-	D8-NOP	78-NOP	D8-NOP	78-NOP	0A-SR	0A-SR	0A-SR	0A-SR
7-	D8-NOP	88-NOP	D8-NOP	88-NOP	0A-SR	0A-SR	0A-SR	0A-SR
8-	D8-NOP	98-NOP	D8-NOP	98-NOP	0A-SR	0A-SR	0A-SR	0A-SR
9-	D8-NOP	09-SL0	D8-NOP	A8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
A-	CD-SL1	BD-SL1	D8-NOP	B8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
B-	D9-SL0	39-SL0	D8-NOP	C8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
C-	D9-SL0	D9-SL0	D8-NOP	A0-CLR	0A-SR	0A-SR	0A-SR	0A-SR
D-	1D-SL1	0D-SL1	E8-NOP	E8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
E-	FD-SL1	FD-SL1	F8-NOP	F8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
F-	DD-SL1	4D-SL1	E0-CLR	E0-CLR	0A-SR	0A-SR	0A-SR	0A-SR

CODE	MNEMONIC	FUNCTION
0	CLR	CLEAR DATA REGISTER
8	NOP	NO OPERATION
9	SL0	SHIFT ZERO LEFT INTO DATA REGISTER
A	SR	SHIFT WRITE PROTECT SIGNAL RIGHT INTO DATA REGISTER
B	LD	LOAD DATA REGISTER FROM DATA BUS
D	SL1	SHIFT ONE LEFT INTO DATA REGISTER

Figure 9.10 The DOS 3.2 Logic State Sequencer.

WRITE

SEQ	READ PULSE / SHIFT / QA'	READ PULSE / SHIFT / QA	READ PULSE / LOAD / QA'	READ PULSE / LOAD / QA	NO READ PULSE / SHIFT / QA'	NO READ PULSE / SHIFT / QA	NO READ PULSE / LOAD / QA'	NO READ PULSE / LOAD / QA
0-	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP	18-NOP
1-	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP	28-NOP
2-	39-SL0	39-SL0	3B-LD	3B-LD	39-SL0	39-SL0	3B-LD	3B-LD
3-	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP	48-NOP
4-	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP	58-NOP
5-	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP	68-NOP
6-	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP	78-NOP
7-	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP
8-	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP	98-NOP
9-	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP	A8-NOP
A-	B9-SL0	B9-SL0	BB-LD	BB-LD	B9-SL0	B9-SL0	BB-LD	BB-LD
B-	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP	C8-NOP
C-	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP	D8-NOP
D-	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP	E8-NOP
E-	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP	F8-NOP
F-	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP	88-NOP	08-NOP

READ

SEQ	SHIFT / QA' / RP	SHIFT / QA' / NO RP	SHIFT / QA / RP	SHIFT / QA / NO RP	LOAD / QA' / RP	LOAD / QA' / NO RP	LOAD / QA / RP	LOAD / QA / NO RP
0-	18-NOP	18-NOP	18-NOP	18-NOP	0A-SR	0A-SR	0A-SR	0A-SR
1-	2D-SL1	2D-SL1	38-NOP	38-NOP	0A-SR	0A-SR	0A-SR	0A-SR
2-	D8-NOP	38-NOP	08-NOP	28-NOP	0A-SR	0A-SR	0A-SR	0A-SR
3-	D8-NOP	48-NOP	48-NOP	48-NOP	0A-SR	0A-SR	0A-SR	0A-SR
4-	D8-NOP	58-NOP	D8-NOP	58-NOP	0A-SR	0A-SR	0A-SR	0A-SR
5-	D8-NOP	68-NOP	D8-NOP	68-NOP	0A-SR	0A-SR	0A-SR	0A-SR
6-	D8-NOP	78-NOP	D8-NOP	78-NOP	0A-SR	0A-SR	0A-SR	0A-SR
7-	D8-NOP	88-NOP	D8-NOP	88-NOP	0A-SR	0A-SR	0A-SR	0A-SR
8-	D8-NOP	98-NOP	D8-NOP	98-NOP	0A-SR	0A-SR	0A-SR	0A-SR
9-	D8-NOP	29-SL0	D8-NOP	A8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
A-	CD-SL1	BD-SL1	D8-NOP	B8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
B-	D9-SL0	59-SL0	D8-NOP	C8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
C-	D9-SL0	D9-SL0	D8-NOP	A0-CLR	0A-SR	0A-SR	0A-SR	0A-SR
D-	D8-NOP	08-NOP	E8-NOP	E8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
E-	FD-SL1	FD-SL1	F8-NOP	F8-NOP	0A-SR	0A-SR	0A-SR	0A-SR
F-	DD-SL1	4D-SL1	E0-CLR	E0-CLR	0A-SR	0A-SR	0A-SR	0A-SR

CODE	MNEMONIC	FUNCTION
0	CLR	CLEAR DATA REGISTER
8	NOP	NO OPERATION
9	SL0	SHIFT ZERO LEFT INTO DATA REGISTER
A	SR	SHIFT WRITE PROTECT SIGNAL RIGHT INTO DATA REGISTER
B	LD	LOAD DATA REGISTER FROM DATA BUS
D	SL1	SHIFT ONE LEFT INTO DATA REGISTER

Figure 9.11 The DOS 3.3 Logic State Sequencer.

If for some reason, the RWTS write protect check was entered with the READ/WRITE switch in WRITE, the write protect switch would still be read correctly. This is some pretty fast addressing, shifting, storing, and loading, the timing for which is illustrated in Figure 9.12. This figure shows the last PHASE 0 period of an MPU access to the READ/WRITE or SHIFT/LOAD switch. The whole operation depends on the very brief access time (70 nanoseconds) of the 6309 PROM that Apple uses for the sequencer ROM. This enables DEVICE SELECT' to fall, the accessed switch to change states, and data out of the sequencer ROM to become valid in plenty of time for the data register to be newly configured when the controller clock rises. The controller clock is Q3 falling or, more accurately, Q3 inverted rising with a typical propagation delay of 15 nanoseconds experienced in the inversion. The data register then responds to its clock well before PHASE 0 falls, allowing the MPU to load the new state of the data register. All this means that the following general rule applies: when the MPU reads the data register, the sequencer responds to any new configuration, performs a resulting operation on the data register, and then gates the contents of the data register to the data bus for reading by the MPU.

Now assume a 6502 program has checked the write protect switch and found it can now write to the disk. The program can then write a byte of data to the disk with these steps:

```
LDA   DATA1
STA   $C08F,X   ;WRITE
CMP   $C08C,X   ;SHIFT
```

Figure 9.13 shows the timing of what happens here. The "STA $C08F,X" without indexing across a page boundary actually puts $C08F,X on the address bus for the last two MPU cycles of this four cycle instruction. The first $C08F,X cycle is a read access and the second is a write access with the MPU controlling the data bus during the greater part of PHASE 0. The first $C08F,X cycle causes READ/WRITE to switch to WRITE about 90 nanoseconds after PHASE 0 rises. Within 70 nanoseconds of this event, the sequencer ROM's data outputs have responded to the new address input. The sequencer state is still 0. That won't change until the sequencing flip-flops sense a clock edge. The sequencer is therefore waiting for a clock edge, sitting at State 0—WRITE—LOAD—QA'. There may or may not be a read pulse, but we don't care. Assume there is no read pulse.

Now look at the WRITE sequence in Figure 9.11 (finally). We are at the top of column seven where a 18-NOP is found. At the next sequencer clock nothing is done to the data register, but the sequencer moves up to State 1. State 1 contains a 28-NOP, which causes sequencing to State 2 at the next clock. State 2 contains a 3B-LD, which causes sequencing to State 3 at the next clock as well as the loading of the data register from the data bus.

Now look again at Figure 9.13. The sequencer clock edge in State 2 occurs while the MPU is placing valid data from the "STA $C08F,X" instruction on the data bus, and it therefore has succeeded in storing data to the data register. Note that a "STA $C0EF" (assumes Slot 6) would not have worked here because MPU data would have been on the data bus at State 0 instead of State 2. **Clearly, Apple designed the WRITE sequence to support the features of the 6502 "STA ABSOLUTE,X" instruction with no page crossing.**

Now look back at Figure 9.11. Assume that when the MPU stored data in the data register, it set QA. In fact, if the data was written by RWTS, it's a sure thing—**QA is set.** We will see that all data that RWTS stores to the disk has its MSB set. This means that instead of sequencing to State 3 in column 7, the sequencer goes to State 3 of column 8. It makes no difference here, but it will when we reach State 7. Until that point is reached nothing happens. At State 7, if QA is set, sequencing up the states continues, but if QA is reset, the sequencer loops back to State 0. Right now, QA is set, so the next state is State 8.

When State 8 is entered from State 7, the WRITE signal switches from low to high. Why? Because the WRITE signal is connected to the A7 input of the sequencer ROM. The sequencing address bits are A7—A6—A0—A5, so A7 is low in states 0-7 and high in states 8-F (see Figure 9.8). Now think about the decision that was made at State 7 in this light. **If QA was reset, the WRITE signal was left alone. If QA was set, the WRITE signal was toggled.**

Continuing on in column 8, State A is reached before the next event. Remember that "STA $C08F,X" was followed by "CMP $C08C,X." Just as we arrive at State A, this second instruction switches SHIFT/LOAD to SHIFT, causing us to arrive at State A in column 6 instead of 8. This is just in time to cause shifting instead of loading at State A, because there is a SL0 in column 6 compared to the LD in column 8. The "CMP $C08C,X" is barely short enough to meet the timing requirements for switching SHIFT/LOAD to SHIFT.

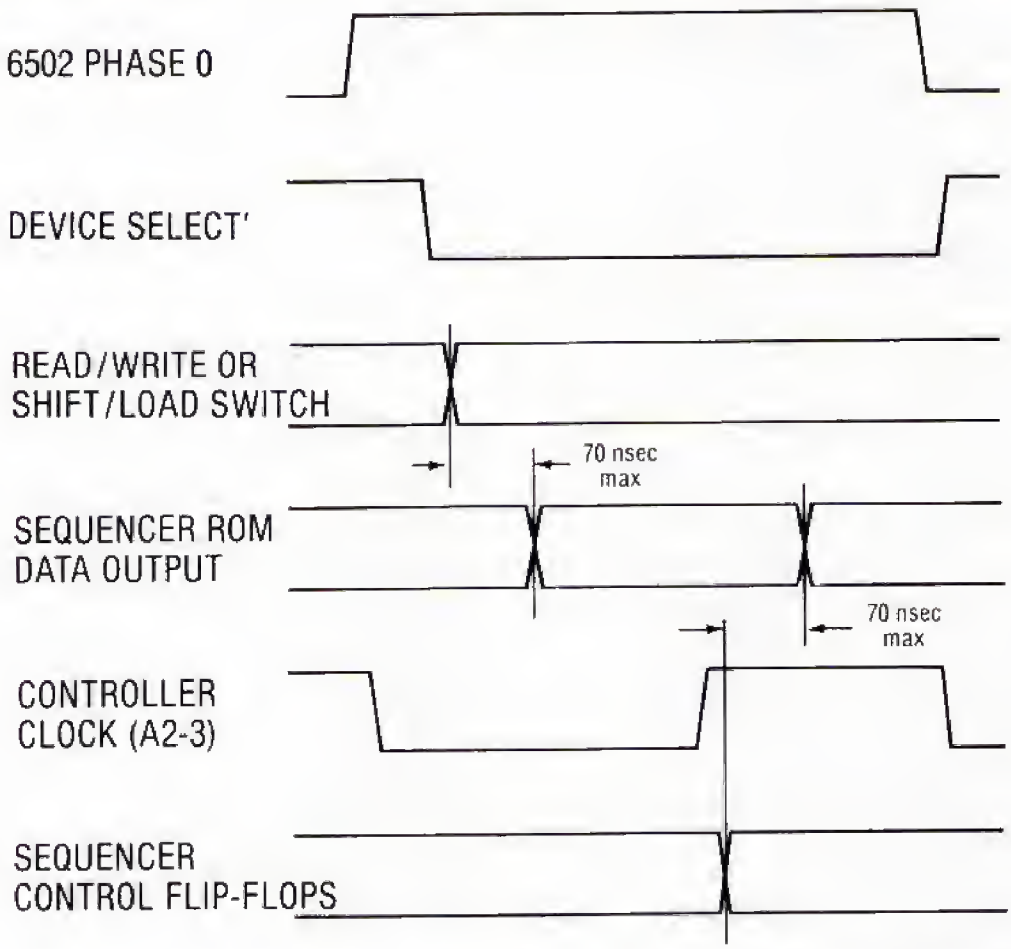


Figure 9.12 Timing Example: Sequencer Control While Changing the READ/WRITE or SHIFT/LOAD Switches.

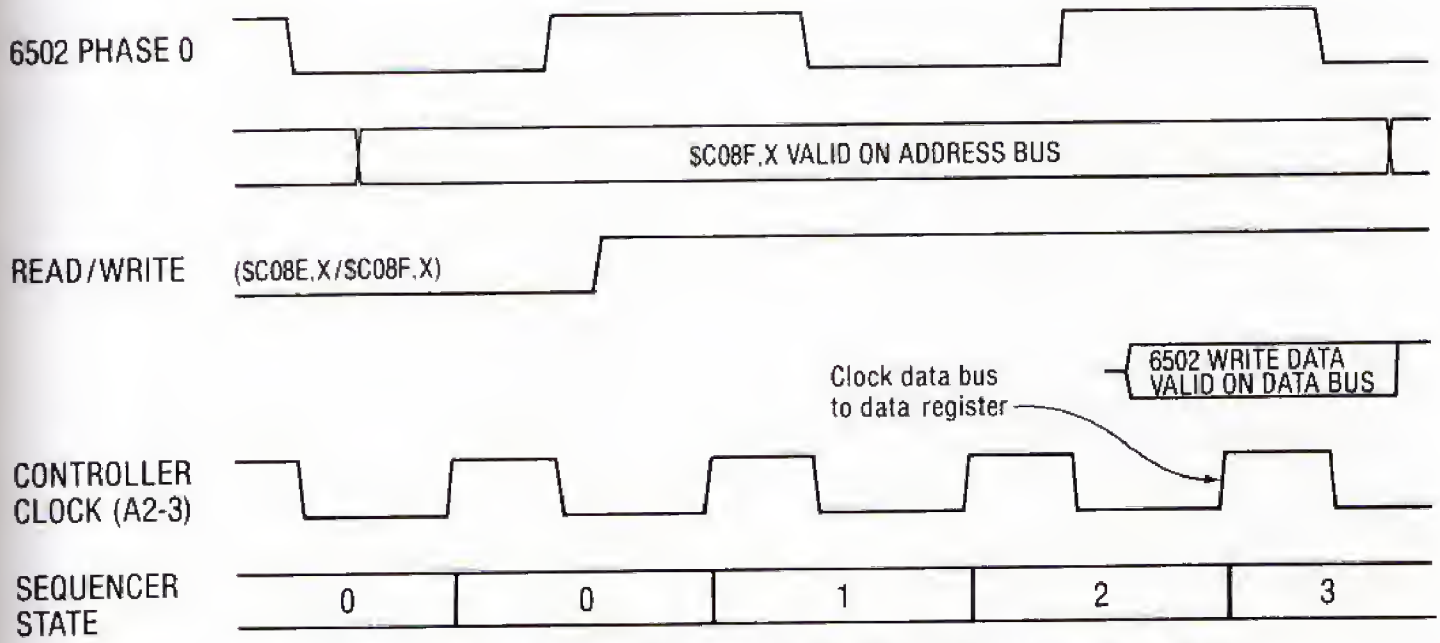


Figure 9.13 Timing Example: Switching to Write After Checking Write Protect.

The SL0 causes the next bit to be shifted into QA, while QH, the LSB of the data register, is filled with a ZERO. SL0 is functionally similar to the 6502 instruction, ASL. From this point, we sequence up to State F in column 5 or 6 depending on whether QA was set or reset by the SL0. From State F, the sequencer will loop back to State 0 if QA is set, toggling the WRITE signal, or it will loop back to State 8, leaving the WRITE signal high, if QA is reset.

Now let's step back and look at what's happening. **Writing to the disk is a load and shift process,** a little like HIRES pattern outputs but much slower. Also, the MPU takes a very active role in the loading and shifting of disk write data. There are two 8-state loops in the WRITE sequence. After initializing the WRITE sequence, data is stored in the data register at a critical point in the A7′ loop. As quickly thereafter as a 6502 can do it, the sequencer is configured to shift left at the critical point instead of loading. Then the MPU goes about its business while the sequencer continues looping, shifting the data register. **If the sequencer senses QA high, flow vectors to the opposite loop, toggling the WRITE signal.**

Figure 9.14 is a flowchart of the WRITE sequence, which may help you interpret the listing. The flowchart is reminiscent of the schematic of a flip-flop. This is not surprising because the WRITE sequence has functional similarities to a toggling flip-flop.

So, the sequencer outputs eight bits of data. What then? Well, if the program controlling the MPU did nothing more, after the eighth bit was shifted out the data register would be all zeros, and no more field reversals would be written on the disk. This constant field condition cannot be read by the drives but results in sporadic read pulses. In other words, the MPU needs to stay involved. Normally, the MPU program will wait until the last bit has been shifted to the disk, then switch SHIFT/LOAD to LOAD with a STA instruction at the precise moment that the data register will accept it. This moment occurs once every sequencer write loop, once every eight sequencer clocks, or once every four MPU cycles. It takes 32 MPU cycles to output eight bits of information, so **to continuously shift out information in 8-bit groups, the MPU must store data at $C08D,X every 32 cycles, then immediately enable shifting with a read access to $C08C,X.** This example writes two bytes to the disk:

```
LDA $C08D,X     ;LOAD
LDA $C08E,X     ;READ
BMI ERROR       ;WRITE PROTECT
                ;ERROR
LDA DATA1
STA $C08F,X     ;WRITE DATA1
CMP $C08C,X     ;SHIFT (4CP)
PHA             ;(3CP)
PLA             ;(4CP)
PHA             ;(3CP)
PLA             ;(4CP)
BIT $0          ;(3CP)
NOP             ;(2CP)
LDA DATA2       ;(4CP)
STA $C08D,X     ;LOAD (5CP)
                ;TOTAL = 32CP
CMP $C08C,X     ;SHIFT
```

The above example illustrates the principle of writing continuous bytes of data: initiate the WRITE sequence, then store data every 32 cycles. This will normally be accomplished in 32 cycle loops. After storing the last byte to be written, wait 32 cycles, switching READ/WRITE to READ on the 32nd cycle. You don't have to write in 32 cycle loops. You can store 6-bit data words in 24 cycle loops if you can figure out some purpose for it. You can store data in any multiple of 4 cycles and the data register will accept it. RWTS writes read syncing leaders by storing $FF to the data register in 36 cycle loops in DOS 3.2 and 40 cycle loops in DOS 3.3.* This creates a series of 11111110 or 1111111100 strings which, we will see, syncs up the sequencer for reading following data.

Disk Data Formats

There is an inherent problem with storing data on floppy disks. As their name suggests, floppies are less than rigid. This and other factors contribute to the following fact of life: just because some Apple wrote data to a disk at a four cycle bit interval doesn't mean that read pulses are going to come back from the disk at the same interval. In the first place, the Apple has a built-in clockpulse jitter with every 65th MPU cycle longer than the rest. This elongates some write intervals by 140 nanoseconds, and it doesn't help. But normal problems with the floppy medium and drive inconsistencies are more significant in causing read pulse variations.

Realizing that the read pulse varies a lot, consider how this variation affects trying to detect the absence of a pulse. In reading the absence of a pulse, the READ sequence must wait a certain amount of time past the last actual pulse, then decide "there was definitely no pulse there; that sure was a ZERO." In the absence of a clockpulse coming off the disk, the previous read pulse, or ONE, becomes the time reference for the absence of a read pulse, or a ZERO.

*In some literature, the read syncing leaders are referred to as series of autosync bytes or self-sync bytes.

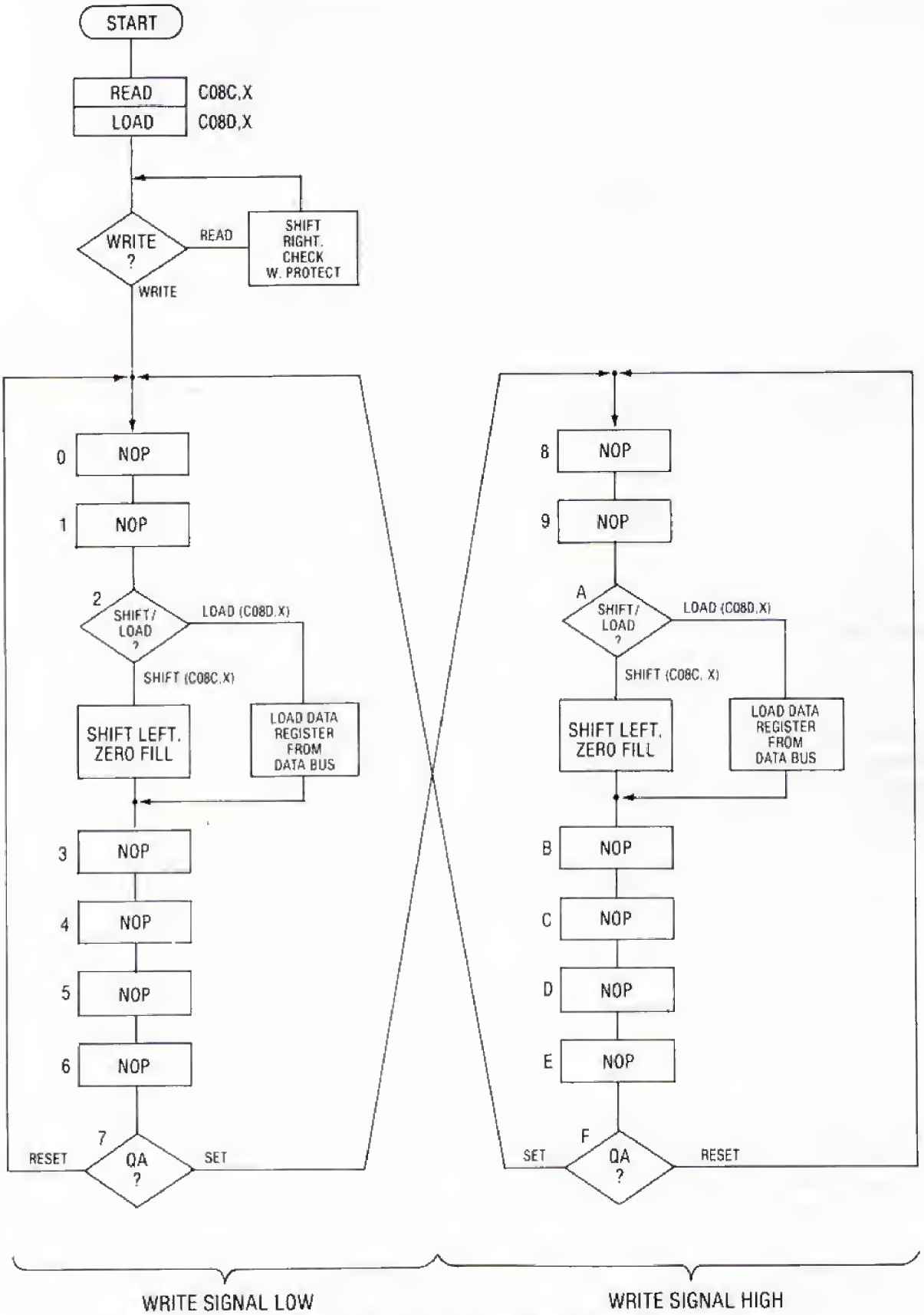


Figure 9.14 Flowchart of the Write Sequence.

What if you have two ZEROs in a row? Well the last ONE bit is the time reference for the second and all succeeding ZEROs in a string. Suppose the reading drive rotates at 280 RPM but the disk was written using a drive that rotated at 320 RPM. You cannot read a long string of ZEROs under these conditions, because while the write interval was eight controller cycles, the read interval could be ten cycles. Furthermore, in the absence of field reversals on the disk for too long, the read interface chip in the drive begins to generate invalid read pulses. You simply cannot read a long string of ZEROs, because your time reference is unstable and your read interface can't do it anyway.

How may ZEROs in a row can be read reliably? Two. **In DOS 3.3, RWTS never writes more than two ZEROs in a row, and RWTS 3.2 never writes more than one ZERO in a row.** Of course, normal data bytes very often have more than two adjacent zero bits. These normal data bytes cannot be stored directly to the disk but must be translated to disk compatible 8-bit words. It takes more than one LOAD-SHIFT cycle to output a data byte, because 256 possible numbers can be represented in a normal byte, but not nearly as many can be represented if you place restrictions on the number of adjacent ZEROs.

There is a second restriction on data which the Apple stores to the disk. **The MSB is always set.** The MSB is used by both the READ sequencer and 6502 programs to define when a byte of read data is in the data register. The MSB is therefore not data but the **BYTE FLAG.*** It serves as a data gap which allows the READ sequencer to hold the previous byte in the data register for a long enough time that a 6502 program can detect the presence of a complete byte in a seven cycle polling loop:

```
POLLIT  LDA $C08C,X
        BPL POLLIT
```

The Apple Disk II was first released with DOS 3 and the associated controller. Then the DOS was upgraded to 3.1, then 3.2, then 3.2.1 with no change in written data format. This format will be referred to here as the 3.2 format. The 3.2 data restrictions are MSB set and no two adjacent bits reset. All sector data was written using the following 32 values:

```
WRITE TABLE FROM DOS 3.2.1

BC9A - AB AD AE AF B5 B6
BCA0 - B7 BA BB BD BE BF D6 D7
BCA8 - DA DB DD DE DF EA EB ED
BCB0 - EE EF F5 F6 F7 FA FB FD
BCB8 - FE FF
```

The values D5 and AA are also valid, and they are used as the first two identifying values which precede every address field and data field. A 5-bit word can contain 32 values, so the 3.2 writing process involves taking a 256-byte data block and translating it into 410 5-bit words. The 410 5-bit words do not directly index the write table to find the byte to be written. Rather, the first word written directly indexes the write table. The second index is an exclusive-OR between the first and second words. Each following index is an exclusive-OR between the current and previous word. At the end, a 411th word is written. The last word of the coded buffer directly indexes the write table for this word. When this process is reversed in the read operation, each byte has to be correctly read for the following bytes to be read correctly. The 411th word read serves as a check sum and must be equal to the 410th decoded word or the read will be deemed unsuccessful and revert to a try again loop. This check sum procedure is an effective method of verifying the validity of disk data transfer.

Apple increased their disk data density in the DOS 3.3 upgrade by easing the restriction on adjacent ZEROs. The 3.2 controller can read this data, but it's a struggle and the possibility of reading errors is great. Apple improved the read reliability by changing the sequencer ROM. They also made the questionable move of changing the Bootstrap ROM and bootstrap procedure, the most notable result of which is that a 3.3 controller will not boot a 3.2 disk. This incompatibility is due solely to program based conventions. The 3.3 controller is fully capable of reading anything written on 3.2 disks.

The DOS 3.3 restrictions on written data are MSB set, no more than one pair of adjacent ZEROs, and at least one pair of adjacent ONEs in bits 6 through 0. D5 and AA are still used only as field identifiers, and they don't meet the pair of adjacent ONEs requirements. This is notable because it helps distinguish D5 and AA from the other valid written

*Steve Wozniak devised the Apple II disk formats. In a speech given in Anaheim, California, on April 17, 1983, he said that his idea for flagging groups of data by having every eighth bit set came directly from the "stop bit" used in RS232 data transmission.

words. The restriction requiring a pair of adjacent ONEs rules out 95, A5, A9, and CA. Besides D5 and AA, the DOS 3.3 written values are:

```
WRITE TABLE FROM DOS 3.3

BA29 - 96 97 9A 9B 9D 9E 9F
BA3Ø - A6 A7 AB AC AD AE AF B2
BA38 - B3 B4 B5 B6 B7 B9 BA BB
BA4Ø - BC BD BE BF CB CD CE CF
BA48 - D3 D6 D7 D9 DA DB DC DD
BA5Ø - DE DF E5 E6 E7 E9 EA EB
BA58 - EC ED EE EF F2 F3 F4 F5
BA6Ø - F6 F7 F9 FA FB FC FD FE
BA68 - FF
```

There are 64 values in the write table, so you can guess that six bits are written per LOAD-SHIFT cycle, and a 256-byte block is written in 342 LOAD-SHIFT cycles. With the size of the data field in a sector thus reduced, Apple was able to increase the number of sectors per track from 13 to 16.

The whole gist of this discussion about data formats is that you can write any sort of bit stream you desire, but you must write something that can be read by the logic state sequencer. The sequencer was designed to read a certain data format, and it's all it can do to read this floppy data reliably. Copy protect artists must study the READ sequence very thoroughly to discover ways to write bit streams which can be synced by the READ sequence with some secret manipulation by a 6502 program.

We will see that the READ sequence will properly read streams of data written from the 3.2 and 3.3 write tables in 32 cycle loops. It does take an indefinite period of time for the sequencer to sync up when it first encounters a random stream of data. However, random data streams in the DOS format are always preceded by read syncing leaders which force the READ sequence into sync very quickly. These leaders consist of a series of 111111110 or 1111111100 data streams. They are written by storing $FF in the data register in 36 cycle loops (111111110) or 40 cycle loops (1111111100) before flowing directly into the 32 cycle data writing loops. Writing data at intervals greater than 32 cycles results in trailing ZEROs (see WRITE sequence, State 2:39-SL0). This causes the ZEROs behind the eight ONEs in the read syncing leaders.

When a string of read pulses from a read syncing leader is applied to the sequencer, bytes of data following the seventh FF36 or fourth FF40 will always be in sync.* RWTS 3.2 syncing leaders are series of FF36s. RWTS 3.2 uses more than seven, which works fine, but only seven are necessary. RWTS 3.3 uses FFs written in 40 cycle loops as read syncing leaders. A string of four FF40s followed by valid data will ensure that following data will be in sync. The use of FF40s allows RWTS 3.3 to sync in a shorter period of time, slightly increasing the space available for data.

There are two different times that RWTS writes to a disk. One time is when it formats the disk, writing sector information for 16 (3.3) or 13 (3.2) sectors on 35 tracks. The other time is when the data field for a sector on a track is written. This consists of positioning the head, then reading until the specified sector **address field** is found. After the desired address field has passed by, the **data field** is written from a 342 byte (410 if DOS 3.2) buffer.

The 3.2 and 3.3 sector formats are shown in Figure 9.15. Other than the three extra sectors, there are several basic differences. The address field identifiers are different, D5 AA B5 in 3.2 and D5 AA 96 in 3.3, causing bootstrap incompatibility. The read syncing leaders are different as was mentioned. DOS 3.2 overwrites each track with 9984 FF36s before writing the sectors, but DOS 3.3 doesn't. DOS 3.2 reserves space for data by writing 431 FF32s after each address field while formatting. DOS 3.3 reserves space by actually writing a data field, with unwritten gaps of about 50 MPU cycles on either side. The gaps are too short to enable accidental detection of a false data field identification string so they don't hurt anything. In either 3.2 or 3.3 format, the data space behind an address field is partially overwritten with a leader and data field when a "write sector" call to RWTS is made.

An interesting point about DOS formats is the DE AA EB series that follows every address field and data field. Apple has always had trouble writing the EB. In RWTS 3.2 they cut off the EB at the end of the data field by neglecting to wait 32 cycles before switching READ/WRITE to READ after storing EB in the data register. They changed that in RWTS 3.3, so the EB is actually written at the end of the **data field**. However, RWTS 3.3 cuts off the EB at the end of the **address field**. Those cut off EBs are not really written, so don't bother trying to read them. RWTS doesn't try either.

*FF36 and FF40 refer to $FF bytes written using 36 or 40 cycle loops.

DOS 3.2 FORMATTED SECTOR

SYNCING LEADER 16-80 FFs	ADDRESS FIELD	DATA SPACE 431 FFs	NEXT SECTOR
FF FF ⟶ FF FF 36 36 36 32	D5 AA B5 VOL VOL TRK TRK SCT SCT SUM SUM DE AA EB 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32	FF FF ⟶ FF 32 32 32	⟶

RWTS COMMAND-2 DATA	SYNCING LEADER 11 FFs	DATA FIELD			
⟶	FF FF ⟶ FF 36 36 36	D5 AA AD 32 32 32	410 WORDS CODED 5/8	SUM 32	DE AA EB 32 32 14

LAST EB OF 3.2 FIELD NOT COMPL WRITTEN.

DOS 3.3 FORMATTED SECTOR

SYNCING LEADER 5-40 FFs	ADDRESS FIELD	ZIP	DATA SPACE	ZIP
FF FF ⟶ FF FF 40 40 40 32	D5 AA 96 VOL VOL TRK TRK SCT SCT SUM SUM DE AA EB 32 32 32 32 32 32 32 32 32 32 32 32 32 32 16	GAP 50	NULL DATA FIELD	GAP 53

LAST EB OF 3.3 ADDRESS FIELD NOT COMPLETELY WRITTEN.

RWTS COMMAND-2 DATA	SYNCING LEADER 5 FFs	DATA FIELD			
⟶	FF FF FF FF FF 40 40 40 40 36	D5 AA AD 32 32 32	342 WORDS CODED 6/8	SUM 32	DE AA EB 32 32 32

Figure 9.15 Diskette Formatting.

The READ Sequence

There is an odd contrast in Apple Disk II I/O. The 6502 program works the tail off the MPU to write data, initializing the WRITE sequence, then storing coded data in precise timing loops. Yet, the WRITE sequence listing is really pretty simple. Reading is just the opposite. The MPU program sits back and lets the sequencer do all the work. The program merely polls the data register, waiting for the sequencer to lay a complete byte at its feet. **The READ sequence is not simple.** Our discussion will concentrate on the 3.3 sequencer, which is very nearly the same as the 3.2 sequencer.

We start by assuming that the sequencer is configured for reading ($C08E,X; $C08C,X) and that a valid data field is passing across the READ/WRITE head with every eighth bit set. Further assume that QA of the data register is not set, there is no read pulse present, and the sequencer is at State 2. You've got to start somewhere, and we are starting at State 2 of column 2 in the READ sequence listing of Figure 9.11.

At this location in the sequencer there is a 38-NOP. At State 3, there is a 48-NOP. At State 4, there is a 58-NOP. We are sequencing through the states, waiting for a read pulse. Assume a read pulse occurs at State 6, switching the sequencer to column 1. The read pulse will last for one sequencer clock because it is synchronized to the clock by a pair of flip-flops and a NAND gate. There is a D8-NOP at State 6 in column 1. In fact if you look at column 1, a read pulse at any of the states would have resulted in a D8-NOP.

When the read pulse goes away after the next clock, the sequencer goes to State D in column 2, a 08-NOP. This means move down to State 0 (18-NOP) and then up to State 1 (2D-SL1). This SHIFT LEFT ONE is a direct consequence of the read pulse. A read pulse occurred, so a ONE was shifted in. Assume the SL1 does not cause QA to become set, and don't get tired of assumptions. We now sequence to State 2 in column 2, right where we started, moving down the line, waiting for a read pulse.

This time let's say no read pulse occurs before we reach State 9. This is the point at which the sequencer decides it can't wait for a ONE any more—that was a ZERO bit. State 9 is a 29-SL0. A ZERO is shifted in. We'll say QA is still not set and we're back to State 2, waiting for a read pulse. This cycle will continue until QA becomes set after an SL0 or SL1. The sequencer is shifting in data based on the presence or absence of read pulses.

Now assume QA sets as the result of an SL0 or SL1. This breaks the loop, shifting flow to State 2 of column four, a 28-NOP. We are at State 2; the next state is State 2; we are going nowhere until a read pulse occurs. This is the **QA WAIT** location, outlined in both the 3.3 and 3.2 listings. **If the sequencer is in sync with the data stream, the fact that QA is set means that a valid eight bit word is now in the data register** just as it was when it was stored there to be written. We will assume for now that the sequencer is in sync with the data stream. This means that the next read pulse will be the MSB set pulse of the next word.

So we're sitting at QA WAIT waiting for the BYTE FLAG of the next eight bit group. The read pulse occurs. Do we clear the data register and do an SL1? No way, Jose. That's a valid byte sitting in the data register. We're going to hold that information as long as possible so that the 6502 program can figure out it's good stuff. The read pulse shifts the sequencer to State 2 of column 3 (08-NOP). Then the read pulse goes away and we sequence to column 4, State 0 (18-NOP), then State 1 (38-NOP), then State 3 (48-NOP), etc. We are sequencing now, waiting for the read pulse that means the second MSB is a ONE or the decision point that means the second MSB is a ZERO.

We'll say a read pulse occurs at State 8. The sequencer goes to column 3, State 8 (D8-NOP), column 4, State D (E8-NOP), State E (F8-NOP), State F (E0-CLR). **The data register is finally cleared.** It was held from the last read pulse or decision point of the previous word until past the BYTE FLAG pulse and second MSB pulse of this word. The sequencer then goes to column 2, State E (FD-SL1), then to column 2, State F (4D-SL1). **We shift ONE twice, once for the BYTE FLAG and once for the second MSB set**, then flow to State 4 of column 2 in the exact condition in which we started: QA reset, sequencing along, waiting for a read pulse.

Now we shift in six more bits of data which sets QA and puts us at QA WAIT. You should notice that QA is set precisely when a complete eight bit word, lead by the BYTE FLAG, is completely shifted in. We made an assumption earlier that we were in sync, but no further assumptions are required. **Once we are in sync with a continuous stream of MSB set data, we stay in sync.**

We are at QA WAIT, waiting for the BYTE FLAG pulse that starts the next word. Suppose somebody had written the word we just read using a 36 cycle loop instead of 32. There would be a ZERO following

the eight bits of data prior to the BYTE FLAG pulse. **You can't read ZEROs from QA WAIT.** There is no decision point here. The only thing the sequencer will respond to is a read pulse, so that ZERO passes right by and is not shifted to the data register.

Assume the next BYTE FLAG pulse occurs, this time followed by a ZERO. From QA WAIT the sequencer takes the same path it did previously, except no read pulse occurs. The decision point is reached at column 4, State C (A0-CLR) followed by column 2, State A (BD-SL1), State B (59-SL0), State 5 (68-NOP), etc. The sequencer cleared the data register, shifted a ONE, shifted a ZERO, then continued processing of the next six bits.

The whole idea of the QA WAIT is this: **the sequencer always knows the next bit is a ONE so it is not monitoring the next pulse as data; it is monitoring the next pulse as the BYTE FLAG.** It monitors the pulse that follows the BYTE FLAG as data, and after monitoring this second pulse, it clears the data register and shifts in a ONE,ZERO or a ONE,ONE. In the process, the valid data word is held in the data register for a long time with the MSB set, a condition which a 6502 program can easily detect.

How does the sequencer get in sync with a data stream? The QA WAIT will cause the sequencer to eventually sync on nearly any valid data stream it encounters. This is because it ignores ZEROs while sitting at QA WAIT. **What the READ sequence does is to give the MPU a look at the data stream in groups of eight bits. Every such group has a leading ONE. ZEROs between a group and the next ONE are lost.**

Suppose the sequencer encounters a stream of data which was written in 32 cycle loops with the MSB set on every 8-bit word. Being out of sync, the sequencer groups the first data into eight bits lead by a ONE in some random way. This is illustrated in the first entry of Table 9.4, 1XXXBXXX. The B represents the BYTE FLAG pulse. It is a normal read pulse, like that generated by any other ONE, but it is represented by B here for illustration. When the sequencer is in sync, the BYTE FLAG will be the first ONE of every group.

At the first entry of Table 9.4, the BYTE FLAG is in the fifth bit position from the left. If the bit following this group is a ONE, that ONE becomes the first bit of the next group, and the BYTE FLAG stays in the fifth bit position. If, however, the bit following this group is a ZERO, the ZERO is lost and the BYTE FLAG moves closer to the MSB of the next group. Eventually, several ZEROs will have been encountered between groups, and the BYTE FLAG will reach the MSB. **From that point, the sequencer will stay in sync because the bit following each group of eight always be a ONE.**

In data written by RWTS, the sequencer is never left to randomly sync on a data stream. All data is preceded by read syncing leaders which insure the sequencer is in sync when following data is encountered. A string of seven or more FFs written in 36 cycle loops or four or more FFs written in 40 cycle loops will insure synchronization. These nine and ten bit write cycles cause synchronization because they are longer than the eight bit groups. When encountered, these strings quickly are aligned into groups of eight ONEs followed by one or two ZEROs. Table 9.4 shows the worst case conditions for syncing to strings of FF36s and FF40s.

In the READ sequence examples we went through, many events could have occurred which we did not take into account. It would not be practical or useful to try to step through all possible events.

Table 9.4 Syncing the READ Sequence to Data.

SYNCING TO A RANDOM DATA STREAM	SYNCING TO FF36s	SYNCING TO FF40s
1XXXBXXX	10B11111	100B1111
1XXXBXXX0	110B1111	11100B11
1XXBXXXX	1110B111	1111100B
1XXBXXXX	11110B11	111111100
1XXBXXXX00	111110B1	B111111100
1BXXXXXX	1111110B	B111111100
1BXXXXXX0	11111110	
BXXXXXXX	B11111110	
BXXXXXXX	B11111110	

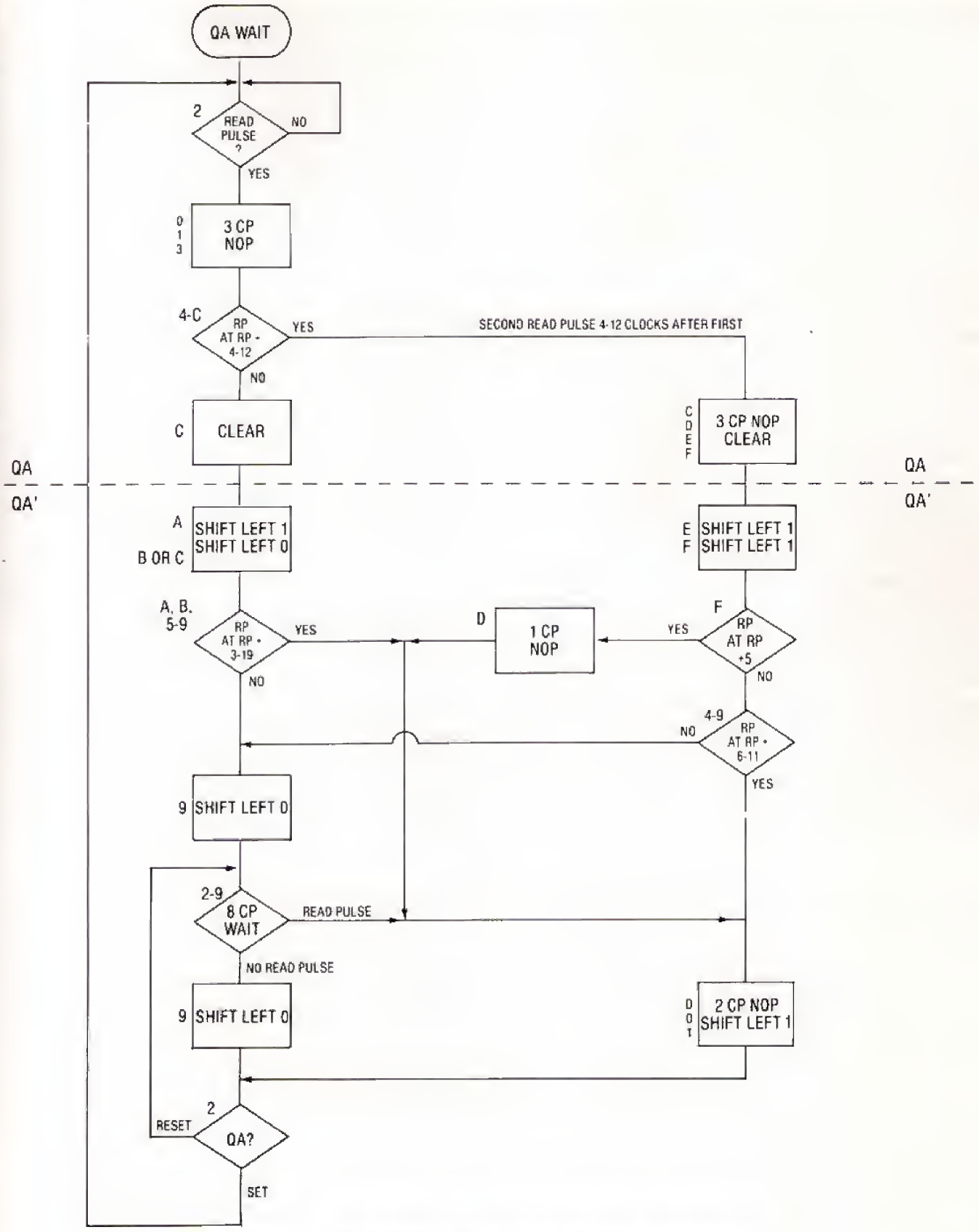


Figure 9.16 Simplified Flowchart of the Read Sequence.

Deeper analysis shows that the read sequence is designed to correctly interpret read pulses while tolerating an expected variation in the pulse interval. Figure 9.16 is my attempt to put the basic flow of the 3.3 READ sequence in perspective in a simple diagram. The pertinent sequencer states are listed next to each block to aid readers in correlating the flow chart to the sequencer listing.

In this flowchart, it was assumed that the read pulse interface circuits were successful in producing a read pulse which was actually one clock period in width. In the READ sequence, there are provisions to handle the rare event that a two clock read pulse occurs. This is an extra ounce of reliability which would clutter up the flowchart and obstruct understanding of normal flow. I have monitored the read pulse with an oscilloscope and have never seen any read pulses that were not properly synchronized to the sequencer clock.

Please direct your attention to the 8CP WAIT decision block near the bottom of Figure 9.16. This represents the time when the sequencer is sequencing up column 2, waiting for a read pulse. The 8CP WAIT indicates that a ZERO will be shifted if a read pulse hasn't occurred by the eleventh sequencer clock after a read pulse, and following ZEROs will be shifted every eight clocks after the first ZERO. The write interval is eight clocks, of course, so there is an allowable distortion of three clock pulses for the first pulse position after a read pulse. This is the main difference between the 3.2 sequence and the 3.3 sequence. After the first ZERO, following ZEROs are shifted every 10 clocks in the 3.2 sequence. This represents a skew away from the write interval while reading strings of ZEROs. It makes no difference with DOS 3.2 data formats, but it makes reading less reliable with DOS 3.3 formats.

The skew in the 3.2 READ sequence is shown in Figure 9.17. While the 3.3 sequencer always makes a shift 0 decision on the third clock period past an expected pulse, the 3.2 sequencer starts making the decision at the wrong point after the first ZERO. There are never more than two ZEROs in a row in DOS 3.3 data, but the sequencer will handle more than two if drive improvements make it a possibility. Note that the 3.2 sequencer will read 3.3 formatted data, especially if the reading drive is slightly slow. The 3.3 sequencer reads 3.3 data more reliably though.

If a read pulse occurs on the twelfth sequencer clock after the previous read pulse, it is smack in the middle of the two points where the sequencer expects a pulse. Is this an early pulse caused by a fast drive or a late pulse caused by a slow drive? The sequencer treats this pulse as an early pulse when it occurs, no doubt because it takes less room in the sequencer ROM to do so. As a result, the sequencer tolerates a fast **reading** drive or a slow **writing** drive better than the opposite condition when reading the data format it was designed to read.

For reference, I have tabulated the intervals which the sequencer can tolerate for various types of written data. Figure 9.18 shows this tabulation, and you can see that it shows what would happen with one data stream the DOS doesn't use. This is sequencer performance, not drive and disk performance. The use of a string of three ZEROs is not recommended, although a copy protect scheme

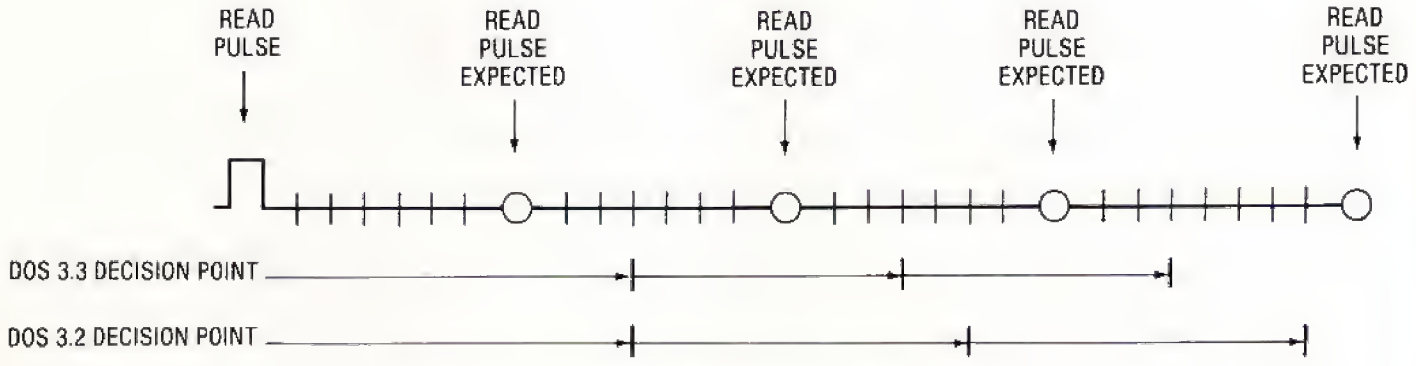
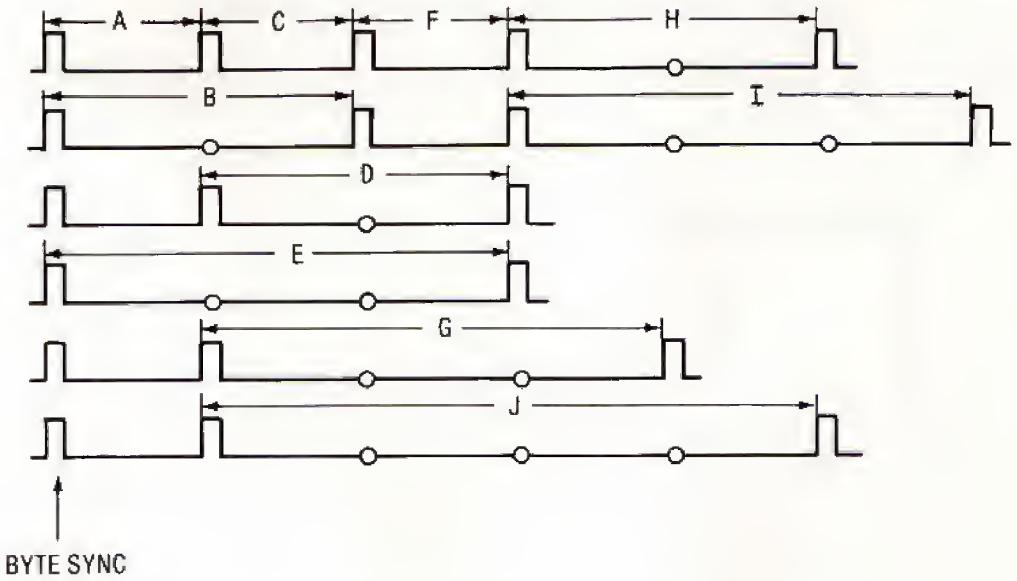


Figure 9.17 Decision Points for Reading ZEROS.



	DOS 3.2			DOS 3.3		
	MIN	AVG	MAX	MIN	AVG	MAX
A	3	8	12	4	8	12
B	13	16	21	13	16	19
C	5	8	11	5	8	11
D	12	16	21	12	16	19
E	22	24	31	20	24	27
F	2	8	11	4	8	11
G	22	24	31	20	24	27
H	12	16	21	12	16	19
I	22	24	31	20	24	27
J	32	32	41	28	32	35

ALL VALUES IN SEQUENCER CLOCK PERIODS

Figure 9.18 Read Performance of the Logic State Sequencer.

might use such a string. As an example of interpreting Figure 9.18, interval A is the case of a BYTE FLAG pulse followed by a second MSB pulse. The expected interval between these two pulses is eight sequencer clocks. The 3.2 sequencer will read the second pulse correctly if it occurs anywhere from 3 to 12 clocks after the BYTE FLAG pulse. The 3.3 sequencer will read the second pulse correctly if it occurs anywhere from 4 to 12 clocks after the BYTE FLAG pulse.

The amount of time a valid data word is held in the data register depends on the second MSB of the next word, normal pulse interval variations, and, in the 3.2 sequencer, the least significant bits of the word. The only bad thing that can happen is that the data will be valid for too short a period of time due to a fast reading drive, a slow writing drive, or both. The average data valid period for some data streams is tabulated in Table 9.5. Values are in sequencer clocks, so the number of MPU cycles is half as many. In the Table 9.5 DATA STREAM entries, "B" represents the BYTE FLAG pulse which follows the valid data. As always, variations in disk surface speed at the read/write head are most likely to cause errors in the presence of one or two consecutive ZEROs.

The average data valid period must be at least 14 sequencer clocks if the MPU is to detect it in a normal 6502 polling loop. Notice that the 3.2 sequencer would have trouble meeting this requirement on data which has two trailing ZEROs. This is one more reason that a 3.2 sequencer would have an easier time with 3.3 formatted data if the reading drive were a little slow.

Table 9.5 Data Valid Periods in Sequence Clocks.

DATA STREAM	3.2 AVERAGE VALID PERIOD	3.3 AVERAGE VALID PERIOD
XX1B1	18	16
X10B1	16	16
100B1	14	16
XX1B0	19	17
X10B0	17	17
100B0	15	17

The selection of D5 AA as the field identifier in DOS 3.2 and 3.3 was no accident. D5 and AA both consist of alternating strings of ONEs and ZEROs, a fact which gives them a more unique identity in an environment filled with strings of FFs and other valid DOS data. Even when it is not in sync, the sequencer should never produce the D5 AA combination from a valid data stream. This is because the 1101010110101010, 11010101010101010, and 110101010010101010 combinations do not exist in a valid data stream. Obviously, if D5 AA should not be accidentally read, then the D5 AA AD, D5 AA B5, and D5 AA 96 combinations should not be accidentally read either. Reliability is even greater because the AA AD combination itself should not be accidentally produced by DOS 3.2 data or DOS 3.3 data.

The AA B5 combination should not be accidentally produced by DOS 3.2 data, but it can be produced by an out of sync encounter with DOS 3.3 data. Specifically, the strings EA 96 AX or EA 96 BX can be grouped as AA B5 if the sequencer is out of sync:

```
X11.101010100.10110101.X
```

It is my speculation that this is the reason the address identifier was changed to D5 AA 96 in DOS 3.3, causing the DOS 3.3 controller to be unable to boot 3.2 disks. It is my further speculation that this is why DOS 3.3 data words all have at least one pair of adjacent ONEs in bits 0 through 6. This eliminates longer strings of ONEs alternating with ZEROs which might tend to be interpreted as field identifiers in an unstable read pulse environment. In particular, the data EA A5 9X or EA A5 AX or EA A5 BX could be grouped as AA 96 if the use of A5 were allowed:

```
X11.101010100.10010110.X
```

The switch from B5 to 96 as an address field identifier in DOS 3.3 may have been required to maintain the normal level of reliability in Apple Disk II I/O. I hope it was worth the cost of bootstrap incapatibility between DOS 3.2 and DOS 3.3. If so, my apologies to Apple for suggesting they could have done better to stick with D5 AA B5 as the address field identifier.

The Read Sequence as a Finite State Automaton

A reviewer of the rough draft of this book, engineer/programmer Jim Aalto, used the Figure 9.11 listing of the DOS 3.3 read sequence to construct his own illustrative tool for studying the read sequence. This tool is valuable enough that we decided to include it in the book (see Figure 9.19). Jim depicts the read sequence as a "finite state automaton." Like a flow chart, Figure 9.19 shows the logical paths the sequencer may take, but the flow is in step with the sequencer clocks pictured at the top. The average read pulse interval is also pictured, so sequencer performance with pulses arriving at various intervals is clearly illustrated. It is recommended that readers studying this figure attempt to relate it to the read sequence listing of Figure 9.11.

PROGRAMMING EXAMPLES FROM RWTS

There are several levels at which you can program disk I/O. The DOS is set up with a very versatile file handling capability which can be utilized from BASIC as shown in the **DOS Manual**. If one had a desire, he could also perform such direct control functions as turning drives on and off, selecting drive 1 or 2, positioning the head, checking for write protect, and checking to see if a drive is turned on from BASIC via PEEK instructions. As an example, the following Applesoft subroutine will tell you

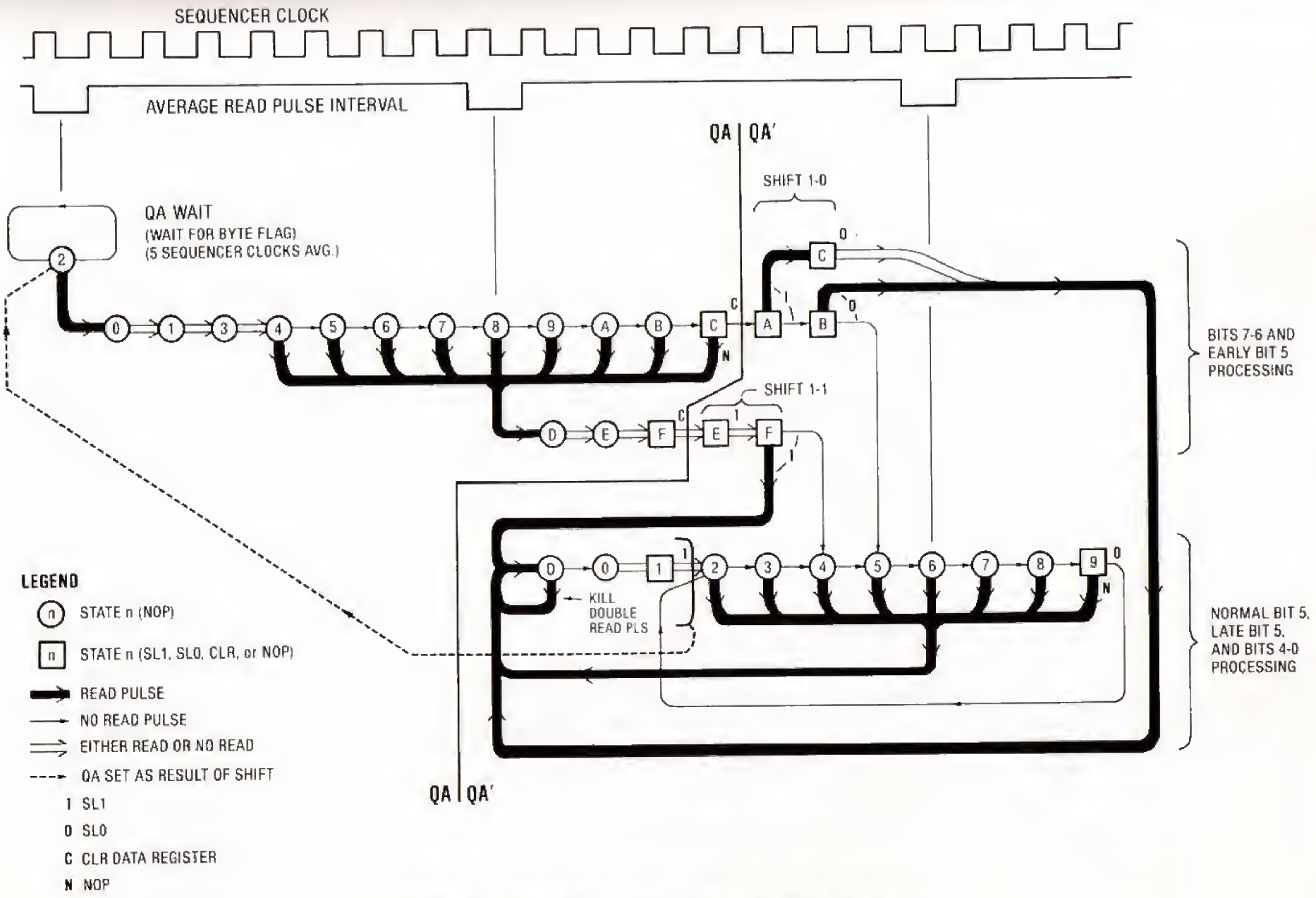


Figure 9.19 Jim Aalto's Finite State Automaton Diagram.

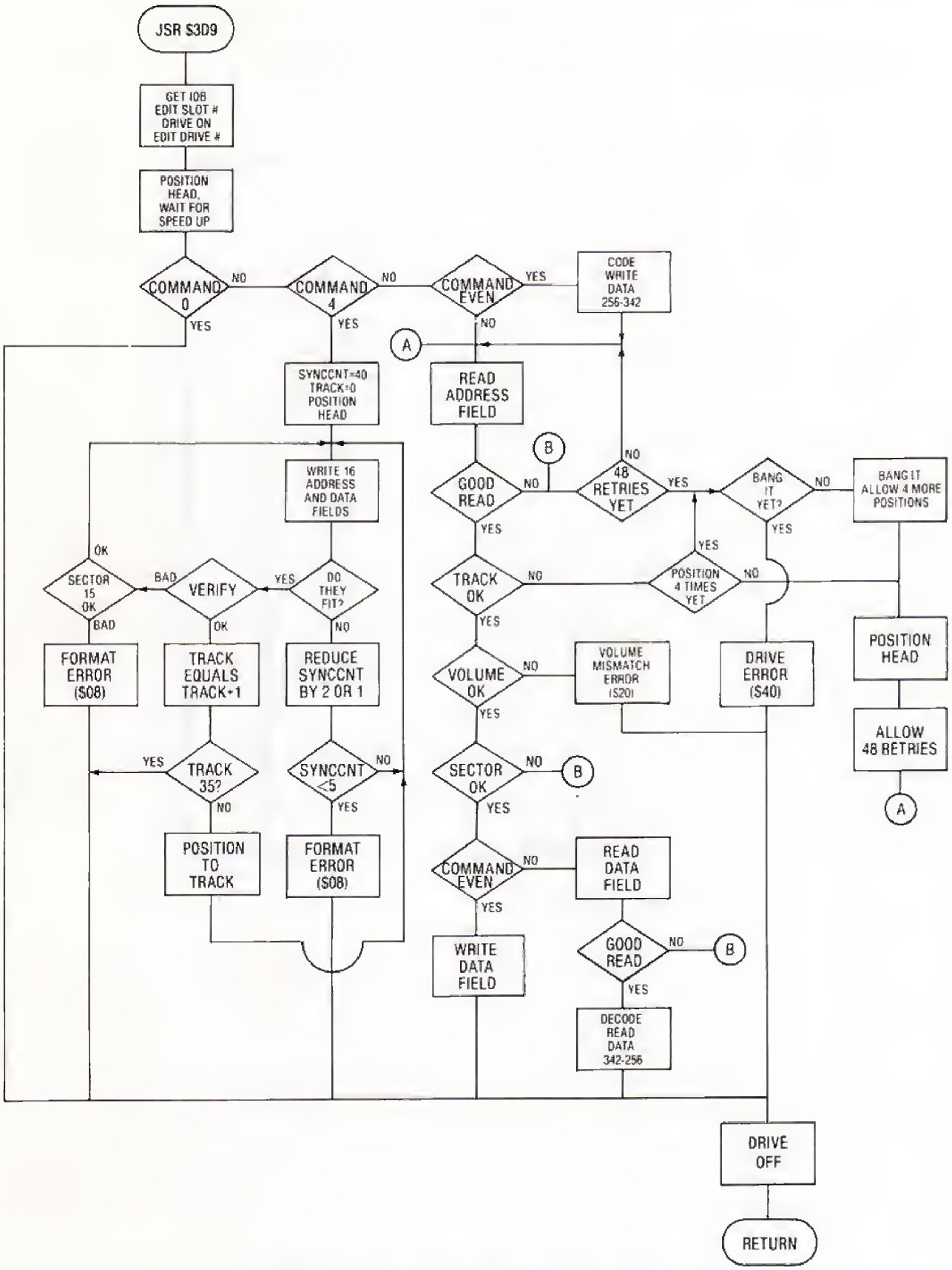


Figure 9.20 Flowchart of RWTS Routine.

if a disk in Slot 6, drive 1 is write protected:

```
10 SLOT6 = 49376 :  REM  $C0E0
20 DRIVE1 = PEEK (SLOT6 + 10)
30 REED = PEEK (SLOT6 + 14)
40 DRIVESTART = PEEK (SLOT6 + 9)
50 LODE = PEEK (SLOT6 + 13)
60 WPROTECT = PEEK (SLOT6 + 8)
70 REM  LINE 60 GETS DATA
   REGISTER AND TURNS DRIVE OFF
80 IF WPROTECT > 127
   THEN PRINT "WRITE PROTECTED"
90 IF WPROTECT < 128
   THEN PRINT "NOT WRITE
   PROTECTED"
100 RETURN
```

Note that the DOS does not have to be resident for this program to work. It bypasses the DOS and goes straight to the controller.

More sophisticated programs may make direct use of the DOS subroutines to perform special functions. You do not have to be an expert on the DOS to do this in your programs. The DOS manual shows you how to read a sector, write a sector, position the head to a track, or format a disk by making calls to RWTS. In *Beneath Apple DOS*, Don Worth and Pieter Lechner thoroughly explain the DOS and show you how to make direct calls to the DOS File Manager. Worth and Lechner also offer worthy perspectives of RWTS and RWTS data formats and provide programming examples which should be of interest to students of Apple II disk I/O.

Even more sophisticated programs may bypass RWTS or modify it, substituting different routines to manipulate the controller and transfer data to and from the disk in formats not utilized by the DOS. To learn how to do this, you should study this chapter to see how the controller works, and you should study RWTS to see how DOS formats are written and read. RWTS is located at $B800-$BFFF of the DOS (assuming 48K of RAM). Its entry point is $BD00, although it can be called by a JSR $3D9 in order to disable interrupts during RWTS processing and to ensure compatibility with future versions of DOS. Upon return from RWTS, 6502 status is interpreted as an error flag (carry set indicates that an error occcurred in RWTS). The following discussion points out some examples from RWTS which show how communication with the disk can be accomplished. Unless stated otherwise, these discussions refer to RWTS as it is in DOS 3.3. You are urged to make a listing of RWTS using your printer, so you can refer to it while reading this section. Make a listing of the Bootstrap ROM at $Cn00-$CnFF while you're at it. Reference to the RWTS general flowchart in Figure 9.20 should also help you keep your bearings.

Drive Turn-On

Drive turn-on is easy. Always configure for reading first (LDA $C08E,X), select the drive ($C08A,X/$C08B,X), and turn it on ($C089,X). Wait about a second after turn-on for the disk to get up to speed and then you can read and write. The RWTS turn-on procedure, which begins at $BD00, is a good deal more sophisticated than this. It takes into account all sorts of factors to get optimum performance in a general purpose routine.

First, RWTS checks to see if the slot being accessed is the same as the slot that was accessed last time RWTS was called ($BD13). If not ($BD19) it makes sure that a drive in the last slot accessed is not still rotating before proceeding. Remember that it takes one second for a drive to turn off after access to $C088,X. RWTS will not turn on two drives at once, presumably because of loading on the +12V power line.

It is possible to check whether a drive at a slot is on by configuring for reading data and monitoring the data register. If a drive is turned on, the data register will be changing and vice versa. This is the check used by RWTS:

```
         ORG  $BD22
         LDA  $C08E,X   ;READ
STILLON  LDY  #$08
         LDA  C08C,X    ;SHIFT
NOTSURE  CMP  C08C,X
         BNE  STILLON
         DEY
         BNE  NOTSURE
```

This routine loops until the drive at the previous slot turns off. It will hang in this loop until RESET is pressed if a call is made to RWTS that specifies a new slot and the last slot was never turned off. RWTS itself always finishes by turning off the accessed drive.

After processing the old slot, RWTS checks if the new slot has a rotating drive ($BD34). This will be the case if the one second turn-off delay hasn't elapsed. If the drive is already rotating, there is no need to wait for it to get up to speed. RWTS saves the rotating/not rotating status ($BD4E), then turns the drive on ($BD4F). This prevents a still rotating drive from turning off after its one second lease on life.

Next, RWTS checks to see if the specified drive is the same as the last call to RWTS ($BD6A). If not ($BD6E), it then assumes that if a drive was rotating earlier, it was the wrong one. Therefore, it sets the rotating/not rotating status to not rotating ($BD73). It also selects the drive via the $C08A,X/$C08B,X switch ($BD74).

Now if the selected drive was not previously rotating, RWTS waits 150 to 175 milliseconds ($BD85) then calls the head positioning routine. 150 milliseconds is not enough time that the drive is up to speed, but RWTS saves time by positioning the head while waiting for drive speed to stabilize. The 150 millisecond delay accomplishes two things. First, it avoids trying to position the head just after a drive has been turned on, which is a period of heavy current flow on the +12 Volt line. Second, if the opposite drive has just been disabled, the 12 Volts may not have yet bled off from the disabled drive.* This might accidentally cause positioning of the disabled drive if RWTS tried to step the enabled drive too soon.

Before DOS 3.2.1, this delay before positioning did not exist in RWTS. Apple added it in DOS 3.2.1, presumably to improve performance, but they botched it up. The way it is written in DOS 3.2.1, the delay before head positioning is dependent primarily on the random state of $46E6 when RWTS is called and to a lesser extent on $46E6,X. The error is in the JSR $BA7F which is stored at $BD7E. This should be changed to JSR $BA7B to make the waiting subroutine entry point correct. The error does not exist in DOS 3.3.

After the 150 millisecond delay in drives which were not previously rotating and almost immediately in previously rotating drives, the head is positioned to the selected track ($BD94). During the 150 millisecond delay and during head positioning, the motor-on time count is incremented at $46 and $47. This two byte counter counts the amount of time a drive has been rotating at the rate of one count per 100 microseconds. The preset count is part of the **Device Characteristics Table** used in a call to RWTS. The DOS uses a value of $D8EF which is equal to -$2711 or -10001 in decimal. **This converts to minus one second.**

After head positioning, the motor count will have partially counted up to $0000. If the drive was not previously rotating, RWTS will go no further than the count and wait loop at $BD9E until the motor count reaches $0000. This will complete the turn-on procedure, which takes one second plus small change for a not previously rotating drive and whatever time it takes to position the head for a previously rotating drive.

*I measured +12V bleed off time in my Disk II drive at 2 milliseconds.

Positioning the Head in RWTS

At bootstrap time, the Apple finds track 0 by banging the head assembly against the outer stop. The programming sequence which does this is at steps $Cn3B through $Cn50 of the 3.3 Bootstrap ROM (the P5A ROM). This routine shows a very economical way to step the head in terms of software overhead. Just wait 20 milliseconds for motor response before turning off a stepper phase. The bootstrap routine uses 80 on-off descending references to $C0E0-$C0E7 (assumes Slot 6) to drive the head 38 or 40 tracks outward depending on initial phase alignment. The timing is $C0E0, $C0E1, wait, $C0E0, $C0E7, wait, $C0E6, $C0E5, wait, $C0E4, $C0E3, wait,..., $C0E4, $C0E3, wait, $C0E2, $C0E1, wait. The wait period is 20 milliseconds. Note that phase-0 is left energized on the stepper motor after positioning. This is indicative of the fact that even numbered tracks are phase-0 aligned and odd numbered tracks are phase-2 aligned. It also contradicts every theory I can think of as to why the analog card was designed so that leaving phase-1 on forces write protection.

The head does not have to be banged against the stop to locate its position. The track number is written as part of the address field in front of every sector on a formatted disk. The head location can be determined at any time by simply reading an address field. Of course banging the head against the stop is the best way to absolutely determine head position, and there is no room in the bootstrap ROM for a routine that reads an address field and then tiptoes out to track 0.

The RWTS positioning routine is far more sophisticated, and there are two calls you can make. Both calls are made with slot number times $10 in the X-register. You can do a JSR $B9A0 with the destination track times two in the accumulator and the current track times two at $478. This will simply position the head using two phases per track. You can also do a JSR $BE5A with the destination track in the accumulator, a Device Characteristics Table set up, and some RAM locations correctly set up.

This will edit the RAM locations and do a JSR $B9A0. The RAM assignments are:

$3C, $3D	-	Device Characteristics Table location.
$35	-	MSB set if drive 1. MSB reset if drive 2.
$478 plus slot $B	-	Drive 1 last accessed track times 2.
$4F8 plus slot $B	-	Drive 2 last accessed track times 2.

The $B9A0 routine is the actual positioning routine for either type of call. It uses a technique of programming duration periods of the stepper motor controls to maximize acceleration in the first part of head travel then to reduce head velocity near the destination track to prevent overshoot and minimize settling time. For this purpose, the routine utilizes a **wait after phase-on table** at $BA11 and a **wait after phase-off table** at $BA1D. These amount to momentum tables for a typical head assembly. The values in the table can be multiplied by .1 milliseconds to give the wait time.

As an example, the Slot 6 phase control for stepping from track $10 to track $11 is as follows: $C0E3, wait $01, $C0E0, wait $70, $C0E5, wait $30, $C0E2, wait $2C, wait $100, $C0E4. This is phase-1 on, phase-0 off, phase-2 on, phase-1 off, phase-2 off. The above wait periods in decimal add up to .1 + 11.2 + 4.8 + 4.4 + 25.6 = 46.1 milliseconds which is the single track response time of the Disk II operating with RWTS, not including a millisecond or so of general computing time. The final wait of 25.6 milliseconds doesn't come from the wait tables but comes from looping through the 100 microsecond wait routine ($BA00) 256 times at the end of every head positioning sequence. This is the settling time of the Disk II head positioning assembly.

As mentioned previously, the 100 microsecond waiting loop that is used to generate delay periods also increments the motor-on counter. This is part of the scheme by which the head is positioned while the motor gets up to speed, killing two welfare bills with one Republican.

Formatting the Disk (Command 4)

Once the head is positioned and the disk is up to speed, RWTS looks at the command entry of the **IOB** (I/O Block) to see what it is supposed to do ($BDAB). Command 0 ($BDAF) causes an immediate exit with drives off and no error indicated. Command 4 ($BDB3) causes the disk to be formatted with 16 sectors written on every track. Other than Commands 0 and 4, even commands cause **writing** of a sector's data field, and odd Commands cause **reading** of a sector's data field, but only Commands 2 and 1 are normally used. Command 2 and 1 processing is the part that gives RWTS its name.

The FORMAT routine starts at $BEAF. It works by starting at track 0 ($BEBB), then formatting each track one by one. It starts by guessing there will be 40 FFs in the read syncing leaders which precede every sector ($BED0). It then writes the 16 sectors with 128 FF40s before sector 0 ($BF0D) and 40 FF40s before the other sectors. Sectors are written in order from 0 to F, but they are effectively interleaved because the sector specified in the IOB is not actually the one that is read during a Command 1 or 2 call to RWTS. Rather, **the specified sector indexes the Sector Interleave Table at $BFB8.**

Writing a sector while formatting consists of writing the address field ($BF17) which is preceded by a read syncing leader, then writing a data field ($BF1C), which is also preceded by a read syncing leader. The write coded data buffer contains all ZEROs ($BEBB), which means the data in the data field will be a string of $96s.

After writing the last sector on a track, the MPU waits for a number of cycles equal to about 200 plus 50 times the number of sync bytes (50 cycle loop at $BF3A). An attempt is then made to read the address field of sector 0. At 40 sync bytes, the sector 0 address field will probably be long gone, in which case the size of the address field syncing leader will be reduced by two ($BF52), then the tail end of sector F will be found ($BF71), and the sectors will be written again starting at the same point on the disk as before ($BF0D). This cycle continues until the sectors fit evenly on the disk. The sync count is reduced by twos until it reaches 16, then by ones until it reaches 5. If 16 sectors do not fit on the disk with a 5-byte leader, the disk speed is probably adjusted way too high and a formatting error (error code $08) is signalled ($BF60).

When the sectors fit well on the track, all the address fields ($BF62) and data fields ($BF67) are read and validated. As each sector is validated ($BF6A), an FF is stored in the correct spot in the **Sector Initialization Map** at $BFA8. Examining this map may give you hints about the cause of formatting errors. When all the sectors are validated the track number is checked ($BF98). If it is track 0 and the sync count is greater than 15, then two is subtracted from the sync count ($BFA2). Otherwise, the sync count is left alone for the next track.

Since the optimum sync count is found while formatting track 0 the other tracks take much less time to format.

After a track is completely formatted, flow returns to $BEDC. An address field ($BEEB) and data field ($BEF4) are read and the next track is stepped to, or RWTS is exited with the drive off if all tracks have been formatted. The drive will actually turn off about one second after RWTS is exited. Waiting until a data field is just past before switching tracks means that anytime the track number is incremented on a formatted disk, an address field will be ready to be read. Also, if all sectors are validated without incident, the adjacent track sectors would be interleaved so that you could start at track 0 and read a single sector on every track, stepping immediately inward after reading each sector, and the sector number read would be the same on every track. The track-to-track sector interleaving serves the purpose of minimizing access time when stepping inward.

Reading and Writing Sectors (Commands 1 and 2)

Reading and writing sectors are very similar operations in RWTS. Both operations cause drive selection and turn-on, head positioning, location of pertinent sector, reading or writing of a data field, and drive turn-off. Additionally, write data must be coded from 256 bytes to 342 6-bit words before locating the specified sector, and read data must be decoded from 342 6-bit words to 256 bytes after reading a data field.

If the RWTS command is not a 0 or 4, read/write processing begins at $BDB5. First, the command type is checked and saved ($BDB5). If a data field is to be written, the 256 bytes of data specified by the IOB (I/O Block) are coded into 342 6-bit words ($BDB9). After write data coding ($BDBC), read and write processing take the same path. A retry count is set to $30 indicating 48 attempts will be made to read the correct address field. The address field is read ($BDC4), then checked for correct track ($BDED), volume ($BE10), and sector ($BE26).

Unless volume 0 is specified, finding the wrong volume causes a return from RWTS with the drive off and a VOLUME MISMATCH ERROR indication (error code $20). Finding an incorrect track number causes up to four repositioning attempts (count preset at $BD09), followed by a major track recalculation ($BDCE), and up to four more repositioning attempts ($BDDC) before a DRIVE ERROR is indicated. A major track recalculation consists of banging the head against the track 0 stop, then repositioning to the specified track. Only one major track recalculation is allowed because the number of recalculation tries is set to one at the beginning of RWTS ($BD04). A major track recalculation is also performed if the correct sector cannot be located after reading 48 address fields. Another 48 attempts are made after recalculation before a DRIVE ERROR is indicated (error code $40).

Once the correct address field has been read, the command is checked again ($BE32). Read operations consist of reading the data field ($BE35), decoding the buffer 342 to 256 ($BE40), turning the motor off ($BE4D), and exiting. Write operations consist of writing the coded data to the data field ($BE51), turning the motor off ($BE4D), and exiting. It takes longer to begin writing than it does to begin reading, so the reading will start just before the read syncing leader is encountered if the data field was written by a Command 2 on the same drive.

Data fields written by Command 2 are not aligned with those written during formatting. Writing of the data field during formatting begins 50 cycles after writing of the address field ends. Writing of the data field during Command 2 begins 112 cycles after the DE AA is detected at the end of the address field. This is the equivalent point in time at which the 16 cycle EB is stored while writing the address field plus 0 to 6 cycles for MPU detection of AA. As a result, Command 2 should begin writing a data field about 49 cycles (112 + 3 - 16 - 50) after the NULL data field is written while formatting.

There are only 53 cycles between the end of the NULL data field and the beginning of the syncing leader of the next address field. If it takes six cycles for the MPU to see the AA at the end of the address field, the data field written by Command 2 will bump up against the syncing leader of the following address field. It seems likely that the first address field sync byte will occasionally be overwritten by Command 2. Furthermore, if the Command 2 drive is faster than the formatting drive, destruction of the first part of the address field leader seems a certainty. This should cause no problem unless the formatting drive was very fast, causing very short address field leaders.

Command 1 should still be able to read the NULL data field written by Command 4. Command 1 will cause the MPU to start looking at the data register while the data field syncing leader is still passing the read/write head. The data field leader is 192 cycles long so the 49 cycle misalignment should not cause the data field identifier to be missed.

The misalignment between the Command 2 and Command 4 data fields is caused by the long processing time used in verifying volume, track, and sector numbers during Commands 2 and 1. If they were concerned, Apple could easily and substantially reduce the misalignment by fetching volume and sector from the IOB and Sector Interleave Table before reading the address field instead of after.

The error detection circuitry in RWTS is very sophisticated, allowing as it does for the possible problems that might occur in data transfer. Not so sophisticated is the error indication found in the IOB after a return from RWTS. There are three types of error codes: VOLUME MISMATCH ($20), error during Command 2 or 1 ($40), and error during Command 4 ($08). With a little extra programming RWTS could give such indications as ADDRESS FIELD CHECK SUM ERROR, DATA FIELD CHECK SUM ERROR, CAN'T FIND ADDRESS FIELD IDENTIFIER, CAN'T FIND DATA FIELD INDENTIFIER, CAN'T FIND END OF ADDRESS FIELD, CAN'T FIND END OF DATA FIELD, CAN'T FIND TRACK, CAN'T FIND SECTOR, SYNC COUNT < 5, and so on. As it is, such DOS indications as DRIVE ERROR or I/O ERROR mean only that something went wrong in RWTS.

Write Routines

There are several routines related to writing in RWTS. One is the **WRITE ADDRESS FIELD** routine at $BC56. This routine writes the syncing leader and address field shown in Figure 9.15 and it is only called when a disk is being formatted. The input parameters are:

Y Reg - Number of FFs in syncing leader
$41 - Volume
$44 - Track
$3F - Sector
$AA - Contains the value $AA

The routine first checks for write protection ($BC57), then stores the first FF in the data register ($BC61), then continues to write FFs in a 40 cycle loop ($BC69-$BC77). The number of FFs is adjusted by the format routine so the 16 sectors fit on each track without a large gap between sector 0 and sector 15. The minimum number of FFs in the leader will be 5.

After the sync writing loop is exited, the series D5 AA 96 is stored directly to the data register at 32 cycle intervals. This is the address field identifier, and the values D5 and AA are not used in the storing of data. The D5 is placed in the data register 32 cycles after the last FF of the syncing leader, so there are no ZEROs following the last FF and it serves no read syncing purpose.

The volume ($BC88), track ($BC8D), and sector ($BC92) numbers are written next, followed by a checksum which is the exclusive-OR of the volume, track, and sector numbers. These four values cannot be stored directly to the disk, but RWTS writes them in a pair of 32 cycle loops following a simple coding scheme. First, the value to be stored is shifted left and ORed with AA. After storing this result to the disk in a 32 cycle loop, the unshifted value is ORed with AA and stored to the disk. The result is that four of the bits are encoded in each storage cycle, and only valid data words are stored. There are 16 possible storage words in this **4-4 encoded storage format**: AA, AB, AE, AF, BA, BB, BE, BF, EA, EB, EE, EF, FA, FB, FE, and FF. The use of AA here slightly degrades the integrity of the D5 AA field identifier, but the system works anyway.

The 4-4 CODE AND WRITE routine begins at $BCC4. This coding method offers less density than the 6-8 coding method, but it could be the basis for a low overhead read/write subroutine which would transfer 2500 byte blocks of data directly between RAM and a track on the disk. Such a low overhead subroutine would serve the purpose of many Apple users.

After the checksum is written, the WRITE ADDRESS routine finishes up by writing the values DE and AA, then part of an EB. The EB is truncated to a 1110 since the controller's READ/WRITE switch is switched to READ ($BCBD) on the 16th MPU clock after the EB is stored in the data register. Switching to READ here results in a 50 cycle gap between the address field and data field. The 50 cycle gap causes no harm, because it is not long enough to randomly produce a three word data field identifier (D5 AA 96 or D5 AA AD).

Another write related routine is a routine which codes a 256 byte data block into 342 6-bit words. This **"PRENIBBLE"** routine begins at $B800. The address of the 256 byte data block must be stored at $3E and $3F and the 6-bit words will be stored in a pair of coded buffers in the 00XXXXXX format. The six MSBs of the 256 data bytes are stored in a 256 word buffer beginning at $BB00, and the 2 LSBs of the 256 data bytes are grouped together in an 86 word buffer beginning at $BC00. The 256 word and 86 word buffers are the source file for the write data field routine at $B82A.

The **WRITE DATA FIELD** routine is called in formatting a disk (RWTS Command 4) and in writing data to a sector (RWTS Command 2). In formatting, the 256-word and 86-word coded data buffers contain all ZEROs, so a NULL data field is written 50 cycles after the end of an address field. In a Command 2 write, first data is coded using the PRENIBBLE routine, then the desired address field is read, then the data is written with the WRITE DATA FIELD routine using the coded data buffers as a source file. This "real" data field is not centered on the NULL data field but lags it by approximately 50 cycles.

The WRITE DATA FIELD routine checks for write protection ($B830), writes four FF40s followed by an FF36 ($B83D), then stores the data field identifier, D5 AA AD, directly to the data register at 32 cycle intervals. Then the coded data buffers are written in 32 cycle loops, using the exclusive-OR of the current 6-bit word and the previous 6-bit word to index the Write Table at $BA29 to obtain the value to be written. This odd storage method is reversed in the read operation, and it creates a checksum by which the validity of data transfer is checked. The 86-word buffer is read first for output from the top down ($B862), then the 256-word buffer is read from the bottom up ($BA7B). Afterwards, the DE AA EB trailer is written with the EB completed. This is opposed to the incomplete EB at the end of the DOS 3.3 address field and DOS 3.2 data field.

Read Routines

The **READ ADDRESS FIELD** routine is at $B944. This routine is used to locate any address field, fetch the volume, track, and sector, and to check validity of the read. It is performed in formatting to verify correct sector distribution and content, and it is used in reading or writing the data field of a sector to locate the correct sector.

The routine starts by looking for any D5 AA 96 sequence. This should occur within roughly 385 valid data register words from any point on the disk. If it doesn't occur within 772 valid data words ($10000 minus $FCFC), the routine is exited ($B94D) with the carry status set, indicating an error condition. After finding D5 AA 96, the four parameters are read in a 4-4 read loop while accumulating the checksum ($B96D). Volume, Track, Sector, and Checksum are stored at $2C, $2B, $2A, and $29 respectively. Next, the presence of the trailing DE AA is verified. Checksum failure or absence of DE AA causes the carry to be set, indicating an error condition. The calling routine will not process the data if the error flag is set or the volume, track and sector are not those desired. RWTS calling routines will attempt to find the correct sector 48 times, bang the head against the stop, then reposition, then try to find the correct sector 48 more times before giving up and deciding there is an error. Reading of the incorrect volume, however, causes the immediate return with a VOLUME MISMATCH error unless volume 0 was specified in the IOB.

The sector which is read is not taken directly from the IOB. Rather, the IOB value is used to index the Sector Interleave Table at $BFB8 which contains 0, D, B, 9, 7, 5, 3, 1, E, C, A, 8, 6, 4, 2, F. As an example, if the IOB specifies sector 1, the sector which will be sought will be sector D. This leads to the following effective order of sectors on each track: 0, 7, E, 6, D, 5, C, 4, B, 3, A, 2, 9, 1, 8, F. Presumably it is chosen to minimize access time to sequential sectors in the DOS environment.

The **READ DATA FIELD** routine is at $B8DC. This is called when the sector writing is verified while formatting or when a Read Sector (Command 1) call is made to RWTS. Reading begins after the desired sector is located via the READ ADDRESS FIELD routine.

Since the READ DATA FIELD routine is always called after the address field has been read and verified, the data field should pass under the read/write head very soon. If more than 32 valid words are read ($B8DC) and D5 AA AD isn't found, the routine is exited with carry set to indicate an error. Oddly, finding D5 AA XX other than D5 AA AD gives the routine 86 extra chances to find D5 AA AD.

After finding the identifier, the data field is read into the 86 word buffer, top byte first ($B8FF) and into the 256 word buffer, bottom byte first ($B913). Each valid word read from the data register is used to index the Read Table which begins at $BA96. This table is the inverse of the Write Table. Each table value is exclusively ORed with the "running total" to get the value stored in the big and little buffers. This is the inversion of the writing process and the running total is checked for correctness at $B92A. A nonmatching checksum or absence of a trailing DE AA causes return with carry set, indicating a read error.

The formatting routine calls the READ DATA FIELD routine just to check the carry status and to verify its own handiwork. When reading the data field in a Command 1 call to RWTS, the data must be decoded from the six bit words in the big and little buffers into the 256 byte RAM block that was specified by the IOB. A "**POSTNIBBLE**" routine which performs this is located at $B8C2.

HARDWARE APPLICATION

INSTALLING A WRITE PROTECT SWITCH ON THE DISK II DRIVE

Did you ever want to store information on a write protected diskette? This involves removing the write protect tab or cutting a write protect notch, writing the file, then sticking on a new write protect tab. Here's another one. Did you ever delete a file, then immediately regret it? If you haven't run into one of these situations, then I'll only say that your ball sure bounces more nicely than mine does. This Application Note details a simple modification to the Disk II drive which enables you to write on write protected disks and also gives you time to have second thoughts about writing files to or deleting files from a disk.

The modification involves installing a single switch on the front of the Disk II drive. The three position switch allows selection between normal operation, forced write protection, or bypass of write protection. If you normally leave the switch in the protect position, you will always have to take an extra step to write, delete or rename a file. This is a fairly normal feature of expensive storage peripherals associated with mainframe computers. Normally, taking the extra step required to overwrite data makes the operator think twice about possible destruction of important data.

The bypass position of the switch allows you to write on a protected disk. This might be of use in writing on disks you have protected for your own reasons, writing on special diskettes that have no notch, and writing on the backs of single sided disks if you do that sort of thing. If you happen to be a software publisher, this mod is a must.

The idea of the write protect switch is not my own. It was pointed out to me that the modification had been suggested in magazine articles. Once it is realized that such a modification is possible, the design of the modification is fairly obvious.

Figure 9.21 shows the modification. Before performing the modification, please read the NOTE OF CAUTION in the front of this book. What is involved is installation of the new switch and rewiring of the already present switch which is activated by the write protect notch. The type of switch required is a ON-OFF-ON SPDT (Single Pole, Double Throw) switch. The installation procedure given here involves installing it on the front of the drive, but you may prefer to put the switch on a remote box. When buying the switch, select one that switches very easily so a minimum of stress is placed

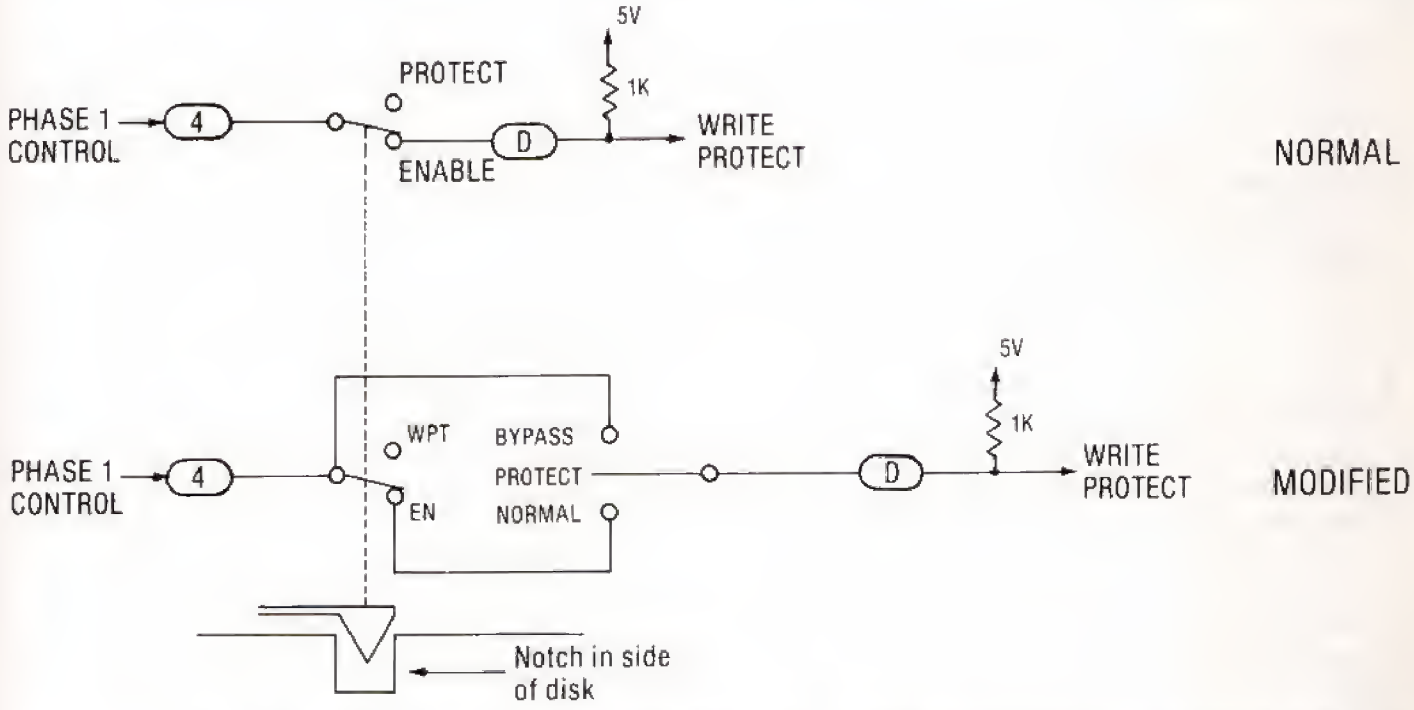


Figure 9.21 A Single Switch on the Front of the Drive Allows the User to Select Normal, Forced Write Protect, or Bypass Write Protect.



Figure 9.22 A Drive With the Write Protect Switch Installed.

on the plastic front panel of the drive. Since the notch activated switch is mounted below the disk slot, the new switch should also be mounted below the slot so wires will not interfere with disk insertion. Choose a mounting point near a reinforcing structure on the back of the front panel to give added strength to your installation. All wiring should be soldered, and 24 gauge insulated wire works nicely. The purpose of steps 5,7, and 11 is to improve accessibility, and you may elect not to perform them if you are good at working in tight spaces.
Installation Procedure:

1. Turn off the computer and remove the controller from its slot. Mark pin 1 of the 20-pin ribbon cable connector and plug with fingernail polish so you will not reinstall the cable incorrectly. Disconnect the ribbon cable from the controller and move the drive to a convenient work area.
2. Remove the four screws from the bottom of the drive. Remove the white case by holding the drive on your palm and sliding the case to the rear.
3. The notch activated switch can be seen toward the front on the left side of the drive. Select the location of your ON-OFF-ON switch so that wires can be connected between the two switches in a way that drive mechanisms are not interfered with. Hold the switch near the selected spot to make sure no problems will arise. The location shown in Figure 9.22 works well with a small bodied switch.
4. Mark and drill a hole for your switch. Clean out any plastic filings which fall into the drive.
5. The big horizontally mounted card is the analog card. You may remove it by disconnecting the two plugs at the back and by removing the four-pin read/write head plug. Remove the retaining screw on either side, and the analog card slides out. You may wish to clean the head with alcohol and cotton swabs at this point.
6. Remove the two beveled head machine screws from each side of the front panel. Position the panel to the side, attempting not to strain the wires connected to the IN USE indicator.
7. Use a fine lead pencil to outline the position of the notch activated switch. This way you can reinstall it in the same position. It will also help to slide a disk in and out of the drive while you observe the switch action, so you will be able to reproduce the same action at reinstallation. Remove the two allen screws which hold the switch to the side of the drive.
8. A black wire is connected to the normally closed contact of the notch activated switch. Desolder the black wire. This wire needs to be connected to the ON-OFF-ON switch and it will probably require a short splice. Splice a short jumper between the black wire and the center or common contact of the ON-OFF-ON switch. Solder both connections and insulate the splice connection with electrical tape or by another suitable method.
9. Connect a wire between the desoldered terminal of the notch activated switch and the NORMAL mode contact of your ON-OFF-ON switch. If you choose to have NORMAL mode be the down position, then the upper contact will be the NORMAL mode contact, and vice versa.
10. A brown wire is connected to the common contact of the notch activated switch. Solder one end of a jumper wire to the same contact as the

brown wire. Solder the other end to the BYPASS mode contact of your ON-OFF-ON switch. The BYPASS mode contact will be opposite the NORMAL mode contact.

11. Remount the notch activated switch to the side of the drive, aligning it to the outline you drew with a pencil. Slide a disk in and verify that the switch clicks on and off as the disk notch is engaged and disengaged.
12. Mount the ON-OFF-ON switch to the front panel.
13. Reinstall the front panel on the drive, making sure the spindle engagement mechanism mates with the grooves on the hinged door on the front panel.
14. The remainder of reinstallation is the reverse of the dismantling steps. You can easily verify operation by attempting to delete some test files and observing the write protect indication. Do not operate with any disk containing important files until you have verified correct operation of the modified drive.

You should mark the functions of the new switch on the front panel. I used white dry transfer letters, available in electronic stores, for this purpose. Afterwards, I sprayed the switch area with an acrylic coating, also available in electronic stores, to protect the lettering. All drive openings should be covered when spraying with acrylic coating to prevent accidental coating of the read/write head.

Some alternate source drives for the Apple use a light emitter and photo switch to check for write protection. Rewiring the photo switch in conjunction with a new ON-OFF-ON switch should also be possible on most or all such drives. I happen to own a FOURTH DIMENSION drive manufactured by Siemens. I added the write protect switch mod to it by splicing into the wires going to the photo switch. On the 4D drive, the photo switch is mounted over the disk slot. The white wire is the input to the photo switch and is the equivalent of the brown wire in the above procedure. The yellow wire is the output of the photo switch and is the equivalent of the black wire in the above procedure.

chapter 10



Maintenance and Care of the Apple II

The modern microcomputer is such a marvelous thing. Just think of the accumulated knowledge and industrial capability of the human race represented by such a machine. Invented by man, feared by man, exploited by man, and hated by man. Especially hated by man when it doesn't work right. After years of having computer systems subjecting us to impersonal and illogical errors, we have advanced to the point where we have computer systems in our own homes subjecting us to personal and illogical errors.

The vast majority of computer mistakes are caused by imperfect programs. The more involved a program, the greater the chance of an oversight by the programmer. We used to curse the computers. Now, when a husband writes a program that allows his wife to enter her kitchen recipes, then destroys them in milliseconds, it's not the computer that gets cursed. Yes, today's computers are very personal.

Occasionally, computer malfunctions will actually be caused by a hardware failure rather than a program in disarray. This should be a fairly rare occurrence, because digital electronics circuitry is so reliable. Yet, hardware failures do occur, and most of us encounter them eventually in our home or business system. In this chapter, attention will be given to the maintenance philosophy of the Apple computer. There will be some discussion of what options you have when your system fails, and of some simple fault isolation steps which can be taken by you in your home. We also will discuss ways of reducing the probability of hardware failure in your system.

APPLE HARDWARE RELIABILITY

There is no electronic circuitry more reliable than modern digital electronic circuitry. Digital ICs routinely operate for thousands of hours without failure, and they are easy to replace if they do fail. The Apple computer is consequently a very reliable machine. There are, however, less reliable facets of a computer than the ICs that populate it. Some weak links in the reliability chain are discussed here.

Tinkering Users

If you are very involved with your hardware, trying new and different things all the time, you are bound to make some mistakes which cause hardware casualties. People who like to tinker with their

computer should do so, because it's as fun as all get out. Those same people would be naive to think they may not occasionally mess something up. Even though my wife thinks otherwise, I have probably set no records in this area. I do, however, consider myself an Apple clobberer of the first degree. The personality traits necessary to reach this plateau of destructive potential are an infantile curiosity and terminal absentmindedness.

The possibility of causing hardware failure by tinkering leads to the following common sense rule. If your Apple is your bread and butter—if it costs you money when it's not running—don't mess with it. If your Apple is your creative outlet, then play with your toy any way you please.

The Peripheral Slots

It is this author's opinion that the most important hardware feature contributing to the Apple's marketing success is the concept of peripheral slots, mounted on the motherboard with address decoded control signals generated on the motherboard. The peripheral slots do represent a reliability weak point though. Suppose you had a television which allowed the owner to enhance it in all sorts of ways by lifting the lid on the television and installing cards in slots on a big motherboard. Surely, every owner who reconfigured his television a lot would mess it up eventually. The TV repair industry would be happy with the extra business, but give them another bonus. Mount the motherboard on a few nylon posts so that it flexes terribly when someone installs or removes a card, and make sure a lot of different manufacturers make the cards so that some of the cards will fit too tightly. How interesting it was of Apple to combine such admirable electronic engineering and such ghastly mechanical engineering in the same peripheral slot concept. Recent Apples have a greatly improved motherboard mounting with a steel bar reinforcing the motherboard near the peripheral slots.

The sort of thing that can go wrong when pulling or removing cards is that a card might get installed backwards. I've only done this twice. It tends to wipe out one or more chips on the card. As soon as a chip shorts a power supply voltage to ground, the power supply shuts itself off and damage to further chips is prevented. An impatient owner might remove or install a card with power applied to the Apple. You can usually get away with this, but when you see a spark, cross your fingers. I've never burned up more than one chip at a time doing this, but it's possible to wipe out every chip connected to D0 of the data bus. This is because D0 is adjacent to +12 Volts on the peripheral slot pins and they may get shorted together. Combining two reliability hazards, you come up with the most common cause of hardware failure in the Apple, a tinkering owner who installs and removes cards while power is applied.

One of the worst things that can happen while installing or removing a peripheral card is that flexing the motherboard might cause a hairline fracture of a current trace or solder joint. Of course the same thing could happen if you drop your Apple or if a manufacturing defect starts to show symptoms. The resulting marginal electrical contact can cause system problems to come and go in a random way based on such variables as temperature, motherboard stress, and the price of hogs in Kansas City. A good computer technician might be able to isolate a problem like this on a good day if you can afford the wages of a good technician for a whole day.

Peripherals with Moving Parts

Moving parts are a reliability problem in any industrial creation. Compare an automobile to a computer. The automobile might run 100,000 miles before its effective life is over. This will be 5000 operating hours at 20 miles per hour with many parts replaced along the way. Yet you could turn on your Apple and let it run for 208 days in a row (5000 hours) and have a very reasonable chance of experiencing no hardware failure. If a part fails, you can replace it and go another few thousand hours without a failure. The main limiting factor on effective life is obsolescence.

Now take a disk drive, an electro-mechanical device with precise mechanical alignment. Don't expect to run a disk drive for 208 solid days without hardware failure. Friction can cause the motors or head assembly to wear out, and like the front end alignment of a car, drive alignment sometimes goes out. Cleanliness becomes a factor. If you are really putting a lot of hours on your Apple, then you should expect to eventually have to have some disk drive maintenance performed. The same is true of a printer. Heavy usage results in wear on the moving parts and in probable eventual maintenance requirements. This is why printer manufacturers will advertise how few moving parts there are in their products.

The Power Supply

In the hypothetical 208 day reliability test that was mentioned earlier, if any unit failed, it would most likely be the power supply. This is because the electric currents in the power supply are so much

greater than in any IC. Of course the power supply is rated to handle a lot of current, but high current devices are usually more apt to fail than low current devices. Also, if the AC line voltage fluctuates, the power supply is the unit most likely to be damaged by the resulting current surges.

Application and removal of power to the Apple can be thought of as a controlled fluctuation. The ICs and power supply are designed to handle the current surge that occurs when the switch is turned on. Still, there is no time when your computer is more likely to fail than when the power is fluctuating, including when you turn the power switch on. This means you should turn the Apple off if power starts fluctuating, as when the lights in the house go dim during a storm. It also means that you should not needlessly turn the power to the Apple off and on.

A particular reliability problem with the Apple power supply is the power switch. The switch arcs sometimes at power up, and this can eventually cause the switch to malfunction. The problem is compounded because the Autostart ROM allows software to hang the system in a way that pressing RESET will not help. The user has no choice, in this instance, but to turn the Apple off, then on to reinitialize the system. This causes extra wear on the switch and exposes the computer to possible power-up casualties. In more recent Apples, the power switch is rated for 10 amps instead of the 8 amps of older Apples. Maybe this reduced the likelihood of switch failures. More importantly, Apple now allows computer stores to replace the switch instead of effectively requiring them to replace the whole power supply as they used to do. This allows a reputable store to repair the problem for about one fourth of the previous cost.

IMPROVING YOUR APPLE'S RELIABILITY

The reliability weak links give some hints on how to improve the reliability of your own Apple:

1. Above all else, never remove or install peripheral cards or ICs with power applied.
2. To improve reliability, don't tinker with the Apple or peripherals. This must be each person's compromise between the conflicting desires of wanting a reliable computer and wanting to tinker. You probably know that my personal choice is to tinker all I want.
3. Keep the Apple covered when not in use so that the electronics stay clean and nothing is accidentally dropped inside. Don't set coffee or sodas on the Apple, because you may spill them.
4. Don't operate the Apple on an unstable power source. Be wary of operating during electrical storms because power may fail.
5. Connect power to the Apple through a bus bar or other device with a switch on it and turn the Apple on and off using this switch to save wear on the power supply switch. The bus bar may have current surge suppressors built in which help stabilize the power applied.
6. You may elect to reduce the operating temperature of the Apple by mounting a fan on the case.

Products are available which perform the three tasks of providing an external switch, surge suppression, and temperature reduction. None of these is necessary for operation of the Apple, but it can be argued that each one could improve its reliability. Necessity of temperature reduction is the most questionable. The Apple uses commercial grade (as opposed to military grade) components which are guaranteed to operate within specifications over the 0-70 degrees Centigrade (32-158 degrees Fahrenheit) operating range. My measurements of the Apple's operating temperature indicate that it operates under 130 degrees Fahrenheit just above the 6502, which is the hottest spot I could find. This is well within the 158 degree specification of the components. Reducing this operating temperature should still reduce thermal expansion, reduce the possibility of malfunctioning of components that are not up to specifications, and reduce the possibility of malfunctioning due to overloading of signals by too many peripheral slot cards. Also, by reducing power supply temperature, one would expect to increase the amount of current that can be supplied before overheating and failure of power supply components occurs.

To see for myself the effect of using a fan on the operating temperature of the Apple, I ran a four hour test, measuring the temperature at the top of the power supply and just above the MPU with and without a fan running. The fan used was a Super Fan II made for the Apple by R. H. Electronics. It hangs from the left side of the case and has an external on/off switch and surge suppression. Four peripheral cards were installed in the Apple under test, which was a Revision 7D Apple II PLUS. The temperature measuring device was a pyrometer which utilizes a thermocouple for a probe. The results of the test are shown in the graph in Figure 10.1. As the graph indicates, it gets very warm near the surface of the MPU. Also, the fan reduces the operating temperature by about 15 degrees Fahrenheit. Whether or not this is worth the price tag is

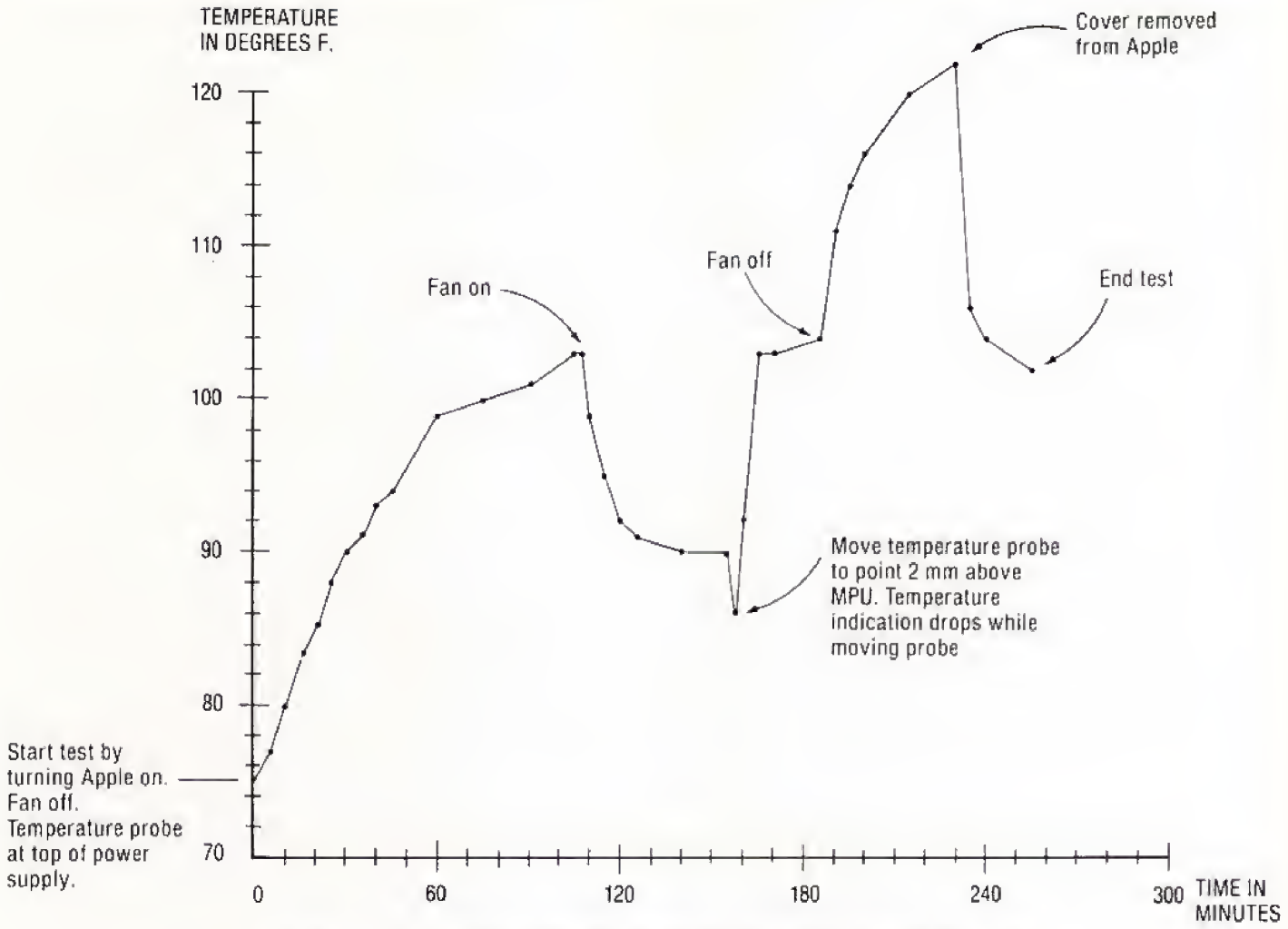


Figure 10.1 Temperature Measurements Inside an Apple II Plus.

a subjective matter. Please note that the graph is probably a good indication of the relative temperatures with the fan off and on, but that there are many variables which affect the absolute reading, including the temperature of the room, probe placement, and instrument accuracy. The measurements made at the surface of the power supply were probably closest to the ambient air temperature in an Apple. It would be fairly accurate to say that the ambient air temperature in the Apple cabinet is about 30 degrees F. greater than room temperature and that the temperature differential can be reduced by about 15 degrees F. with a fan.

REPAIR OF THE APPLE II

Repair of a broken digital computer is different than repair of other sorts of electronic equipment. Many uncertainties of electronic circuit operation do not exist in digital equipment, because most of the circuitry is made up of two-state electronic switches. In most circuit elements, either current flows or it doesn't flow. This is a simple condition compared to the infinite variety of signal conditions which exist in analog electronics.

The complexity of digital equipment lies not in complex electronics but in complex logical capabilities. As a result, any difficulty in the repair task will often be due to this logical complexity. This is good, because it means many hardware functions in a computer can be verified, and many casualties can be isolated by self diagnostic programs. More troublesome problems can be diagnosed by external computers designed and programmed to troubleshoot certain classes of problems. Since computer casualties are often logical malfunctions, what better way is there to solve them than by logical analysis using a computer?

There is virtually no end user self diagnostic capability in the Apple II computer. There is no hardware or firmware based timing verification, I/O verification, or memory verification. RAM based diagnostic programs are available on disks to computer dealers but are not available to owners. There is, therefore, very little an owner can do by way of isolating difficult casualties unless there is an in house computer technician with some test equipment. There are some checks that can be made by anybody, but more on that later.

The typical computer retailer will have what Apple calls a Level I repair capability. They will often have a shop and a computer technician. They will also have several disks full of diagnostic programs and a pipeline to Apple. If their diagnostics will load, they will verify and isolate faults in RAM, ROM, the keyboard, video display, and Apple manufactured peripheral devices. The technician or salesman can then replace components that are indicated to be bad and hopefully fix the malfunctioning Apple very cheaply. The dealers also have disk drive alignment disks and procedures which allow their technicians to precisely align the Disk II drive.

Dealerships with more sophisticated repair capabilities will also work on some products not manufactured by Apple, such as 80 column cards or printers. If there is a resident Apple technician with an oscilloscope, he probably will repair many problems not pointed out or isolated by the diagnostics.

When a problem is beyond the capability of a dealership or the repair of a problem will be so time consuming that it will not be cost effective for the dealer or customer, the dealer will swap out a major assembly for a very reasonable cost. For example, a motherboard swap costs $120, a power supply costs $90, and a disk controller card costs $55.* He then sends your repairable assembly to an Apple Level II repair facility. Apple can repair the assembly more easily because they have sophisticated test fixtures, documentation, and assets not found in a computer dealership. Other companies besides Apple will also have a turn around policy on their Apple compatible products. Therefore, when a peripheral card fails, you may well be able to get a quick swap at a computer dealership.

Apple will not allow Level I repair shops to perform some tasks. They maintain this control by refusing to swap assemblies upon which unauthorized work has been performed. For example, a Level I shop can change the analog card in a disk drive, but they can't change the drive motor. Also, with few exceptions, Apple won't accept a modified assembly. One exception is the shift key mod when wired to pin 4 of the game I/O socket or pin 1 of socket H14 (the same point electrically). Apple liked that mod so much, they included it as part of the Apple IIe.

Apple's no swap rule for modified assemblies is fairly reasonable considering that they must make their own repair operation cost effective. An example of where this rule was carried too far is the power supply switch. Apple did not allow any repair work on the power supply and put two rivets in the bottom so it was hard to gain access to the internal components. If those two rivets were removed, then Apple wouldn't accept that assembly for swap out without special permission. As a result, if the switch went bad, the customer was required to pay $90 for a power supply swap out when switch replacement should only be about $25 for parts and labor. Then Apple changed the power supply from the silver colored one to the gold colored one, to which access can be gained by simply removing some screws. They now allow dealers to change the switch on either type power supply, although some dealers will still swap out the whole power supply at a $90 cost.

The primary hardware based self diagnostic feature of the Apple is an empty peripheral slot. An empty slot becomes a diagnostic port when you plug a specially designed test fixture into it. There can be no doubt that Apple must use such test fixtures for production check out as well as fault isolation, because they are an obvious necessity. What can you tell about an Apple from a peripheral slot? You can verify power supply voltages, verify timing, check RAM and ROM and all address bus command features via DMA, measure for shorts at all pins, exercise the 6502 with a test program via the INHIBIT' line, and verify correct interrupt response. Problems with individual memory chips can be isolated to the chip while other problems may be isolated only to an area. Further isolation of problems can be performed by attaching jumpers from the smart test fixture to various ICs. Apple probably has an area full of engineers who do nothing but design test fixtures for Apple products, program them, and write test procedures for them.

We pay for this diagnostic capability when we buy Apple products, even though it isn't built into the Apple. Large scale automated checkout and fault

*Suggested prices recommended by Apple for out of warranty repairs, August 1, 1981. Still valid as of March 15, 1983.

isolation of a product is the only way to provide quality assurance and service a complex mass production device in a cost effective way. Money for developing this capability must come from sales and service revenues.

WHEN YOUR APPLE BREAKS

When your Apple breaks, chances are that you will have to take it to a computer dealer for repair. Yet there are some very simple checks you can make which might get your Apple up in a hurry. These are checks which can be made by anyone, and they are the kind of checks a salesman might make if you brought your Apple in on the service technician's day off. Be aware that any damage you cause while making these checks will void your warranty if it is still in effect. Also, any use of test equipment by unauthorized service personnel might put your warranty in jeopardy.

In these discussions, the use of a multimeter, logic probe, or oscilloscope will be occasionally called for. If you do not have access to the instrument mentioned, or you do not know how to use it, it is time to get your system to a dealer. Incidentally, you can buy a logic probe and a multimeter for $20 each at *Radio Shack* and learn how to use them in a few minutes.

If you manage to find the exact cause of a problem, you can buy most Apple components in computer electronics stores or, more expensively, at computer dealers. The computer dealer will charge you more because he is not in business to sell electronic components. Like virtually all American maintenence operations, the computer dealer will put a big markup on his parts prices to improve the profitablity of his service department.

There are hazardous voltages inside the Apple, but they are all in the power supply. Nevertheless, it is a good idea to pull the plug on the Apple anytime you are working inside. Never work on the power supply with the line cord attached. Many of the power supply components are not isolated from the line voltage and there are numerous dangerous voltage points in the power supply.

The Peripheral Card Check

A check that should be made at an early point for almost any persistent symptom is to turn off the Apple, remove all the cards, turn the Apple on, and see if the symptom disappears. If it does, you can find which card is causing the fault by turning the computer on with each card installed by itself. Then you can operate the Apple, losing only the capabilities represented by the malfunctioning card until it is repaired. Even when a peripheral is malfunctioning, it is a good idea to check operation with all other peripheral cards removed. For example, your disk controller may be loading down the RESET' line and causing the firmware card to misbehave. This procedure of isolating a problem to a peripheral card will be referred to as the **peripheral card check**. Other than this most basic of checks, your course of action will depend on your symptoms. The most easily recognized symptom is a completely dead Apple, normally indicating a power supply problem.

Power Supply Problems

There are two symptoms you will normally encounter with power supply problems. The Apple is dead and there is a low level clicking noise coming from the power supply, or the Apple is dead and silent too. If there is a clicking noise, the power supply is quite likely good, but a motherboard or peripheral card malfunction is causing an overload condition. If the Apple is dead silent, there may be a casualty in the power supply itself.

The clicking noise is the tinkerer's symptom. Chances are very good that somebody was installing, removing, or modifying something in the Apple. When any casualty symptom follows tinkering, concentrate your investigations in the areas that were tinkered with. If something was touched, look it over.

When the clicking noise occurs, turn the computer off immediately. If a component has just shorted, it may be hot or show evidence of burning. Visually inspect the motherboard and peripheral cards under good lighting. Look for such things as ash or black marks on the components. Touch all the ICs lightly to see if any are hot. Be careful or you might burn your finger when you get to the right one. Check that all cards, plugs, and ICs that may have been tinkered with are correctly installed, not reversed, and with no shorted pins. Perform the peripheral card check, but don't leave the computer on with the clicking noise for any longer than necessary. The peripheral card check will indicate whether the motherboard or a peripheral card is the probable cause. It can be further determined that the problem is not in the keyboard by disconnecting the keyboard plug from its motherboard socket.

If you were unable to isolate the exact problem cause, it may be time for you to take your system to a dealer for repair. At this point, you will be able to describe the symptoms to the service technician on a level which will be helpful to him. He will probably



Figure 10.2 A Power Supply With the Bottom Off.

verify a short to ground exists with a multimeter and try to isolate the short by removing ICs from the motherboard or peripheral card in groups until the short goes away. You can do this yourself, but beware. Even experienced technicians damage ICs or install them incorrectly on occasion. If you remove all the ICs from a card and put them back in, you very well may create some casualties that weren't there when you started. Also, the problem may be more difficult than a dead short, and you might end up requiring a board replacement if it is too difficult.

The second bad power supply symptom that can be observed is a completely dead Apple with no clicking noise. This can be verified to be a power supply problem by measuring the +12, -12, +5, and -5 Volt lines at any peripheral slot with a multimeter. If the voltages are good, then a timing problem is indicated. If incorrect voltages are present, there is probably a power supply problem which will require a $90 swap out from a dealer. If no voltages are present, you may still have a $90 swap out coming, but you may only have a bad switch.

In this event, you have two options. You may take the computer to a dealer who might verify the switch is bad and replace it or who might insist on a $90 swap out. You may also verify the switch is bad yourself and replace it. Replacement is easier to accomplish in the newer, gold colored supply. This is because the newer supply is not riveted closed and because the switch is a standard sized rocker switch. The switch on the older, silver colored supply is undersized and not easily found in part stores. Any Apple dealer can obtain either switch from Apple

It used to be that Apple would not accept an older supply for swap out if the rivets had been removed, but since the newer supply was introduced, they accept the older ones with rivets drilled out. If you enlarge the mounting hole on an older supply to make it take a standard sized rocker switch, it probably will not be accepted for swap out if a more serious casualty occurs in the future. While the cost of a power supply with turn in is $90, the cost without turn in is $300. You should, therefore, weigh the consequences before embarking on any course of action. Also, even though Apple has been accepting older power supplies with the rivets removed, they have not made any public announcement to that effect. Neither Quality Software nor myself can be held responsible for any consequences of your possible decision to remove the rivets and attempt to replace the switch. If you do make such a decision, here is a rough procedure:

1. Turn the Apple off.
2. Remove the power cord from the power supply.
3. Disconnect the power supply plug from the motherboard connector.
4. Remove the power supply (four screws through Apple base plate).
5. Drill out the middle rivet on both sides of the power supply (1/8" bit).
6. Remove four screws from both sides and separate top from bottom.
7. Verify switch casualty with ohmmeter.
8. Replace switch.
9. Reverse the dismantling procedure except for installation of rivets.

DO NOT APPLY POWER TO THE POWER SUPPLY WHILE INTERNAL COMPONENTS ARE EXPOSED. THE VOLTAGES INSIDE ARE VERY HAZARDOUS WHILE POWER IS APPLIED.

Peripheral Failures

It is sometimes fairly obvious that the only problem lies in a peripheral or its interface card. If everything else works, but a printer won't print, it's pretty cut and dried. Other peripheral failures are less obvious. Cards that steal ROM addressing like the firmware or RAM card are so integrated into the overall operation that when one fails, symptoms can be the same as motherboard failures.

When you are certain that a fault lies with a peripheral, there are some steps you can take to try to determine the exact cause. First, with the computer off, remove all the other peripheral cards to be certain that one of them isn't somehow causing the problem. If that doesn't help, turn the computer off, and give the suspect peripheral card a visual inspection. Assuming the ICs are mounted in sockets, wiggle them all to make sure that they're properly seated. Verify that any plugs are properly installed. Clean the contacts of the card's edge connector with a pencil eraser or with alcohol and cotton swabs, preferably the latter. Reinstall the card and verify that the problem still exists. You can perform the same steps on any cards mounted in the peripheral itself.

Just the act of removing a peripheral card and installing it will often cure many problems, at least temporarily. Some lower quality contact materials will tend to make poor electrical contact when the temperature rises. Just wiggling the card can cure the problem, but be sure to wiggle it with the computer turned off. Cards with gold plated contacts are much less likely to cause this sort of trouble.

Another thing you can try is to run the peripheral in a different slot than its ordinary one, assuming it is not slot dependent. If it is slot dependent, try running another peripheral in that slot, the object being to prove there is nothing wrong with the slot's signals or connections. If the problem persists, you may as well start calling computer dealers to find one who will work on your peripheral.

If you cause a malfunction by removing a card with power applied, suspect that ICs connected to the INHIBIT' line are burned out. This includes the 74LS09 on a firmware card or a RAM card. You can burn up these LS09s by removing any card from any slot with the power on and accidently shorting pin 32 (INHIBIT') to pin 33 (-12V). If you are less lucky, you might short pin 50 (+12V) to pin 49 (D0). In the latter instance, you may possibly destroy numerous ICs.

Here is one last tip for a special situation. I have known the LS125 on the Disk II analog card to be damaged on three separate occasions. On two of those occasions, the LS125 failure was caused by a person plugging the 20-pin ribbon cable connector into the controller incorrectly. The symptom was that the drive was always configured for writing, even when the controlling program was attempting to read data. Booting or cataloging a disk, for example, would clobber the disk. A typical operator will clobber several disks under these circumstances before realizing what he is doing. He thinks he is trying to read bad disks, but he is really making disks go bad by trying to read them. If you encounter these symptoms, try replacing the LS125 on the analog card of the offending drive or drives. Then verify operation with disks containing non-critical data. This tip also pertains to most second source 5-1/4 inch drives available for the Apple. They generally use an LS125 for the same functions as the one in the Disk II drive.

Other Symptoms

There are no error lights built into the Apple, but there are some very distinct error indications which you can interpret if you understand the Apple. First, does it beep when you turn it on? The beep isn't made by some oscillator. It's made by programmed control of the speaker by the MPU. It means that a power-up RESET was generated, the 6502 works and is capable of address bus and data bus control and data bus reception, the F8 ROM works, timing works, page 0 and 1 of RAM work, and a good portion of the address decoding circuitry works. If you can get to the monitor, you can try the following RAM test:

```
*1 CTRL-P (ENABLE PRINTER)

*C050 C053 C054 C057 N 265:FF N 266<265.
BFFEM 266<265.BFFEV 265:0 N 266<265.BFFE
M 266<265.BFFEV 34:14 (CR)
```

Please note that there is a space between the ending "34:14" and the RETURN. These keystrokes, when entered from the monitor, will enable the Slot 1 printer and then take turns filling RAM with ONEs then ZEROs. It verifies the transfer of each byte and prints out any errors. RAM locations $265 through $BFFF are checked out, and the test can be cycled indefinitely with every error recorded on the printer. The video display gives a visual indication that the test is in progress. It can be terminated by pressing RESET.

It is important to verify the errorless operation of RAM because a random error in memory transfer can cause seemingly illogical program crashes. Most 6502 programmers could write a good RAM diagnostic program and save it on disk, but what good would it do if a RAM casualty prevented the operation of DOS? In fact, what good are any disk based diagnostics if any casualty prevents the operation of DOS? This is why the little monitor based RAM diagnostic can be pretty handy.

The second big indicator of the nature of a problem is the video display. The scanning of RAM for video output is done independently of the MPU, the address bus, and the data bus. The only common bond is timing. The address bus, data bus, MPU, RAM and ROM can all be burned to a crisp, and you will still have the display window on the screen. The contents would be jibberish, but the window would be there with its black margins on all sides. The presence of the window means that timing, the video scanner, and most of the video generator are working.

The four possible combinations of the two operational indicators can greatly narrow the field of possible causes of a given problem. The following interpretations of symptoms should help. In all cases, perform the peripheral card check, visually inspect the motherboard, and verify that all ICs are properly seated. Refer to the foldout in the back of the book entitled "Motherboard Component Locations" to see which functional areas are affected by various ICs.

1. **No beep, no display.** This is one dead Apple and power supply problems are the probable cause. This can be verified by measuring the +5V, -5V, +12V, and -12V supplies at any peripheral slot with a multimeter. If any of the voltages is incorrect, read the **Power Supply Problems** section of this chapter for an indication of how to proceed. If the power supply voltages are good, timing generator problems are indicated. Verify the presence of the timing generator signals using a logic probe or oscilloscope. Refer to Figures 3.2, 3.8, and 3.9, and replace possible malfunctioning ICs if you have spares.
2. **No beep, display window present.** Timing is good, but the computer does not execute a stored sequential program. Use a logic probe or oscilloscope to verify whether or not signals are normal at all pins on one of the peripheral slots. Pay particular attention to the address bus, data bus, INHIBIT', RESET', and other 6502 control lines. The condition of the peripheral slot signals should guide you to possible causes.
3. **Beep, no display window.** Most of timing is good, but check LDPS' and LD194 with a logic probe or oscilloscope. Also check the video scanner outputs and the BLANKING, SYNC, PICTURE, and VIDEO output signals in the video generator.
4. **Beep, window present, programs crash.** The computer executes programs but gets into trouble in certain cases. Turn off the computer and remove and reinstall your firmware card or RAM card to see if that is the problem. Try operating with all cards removed except the firmware card or RAM card as well as doing the normal peripheral card check. Attempt the monitor based RAM diagnostic that was shown earlier. Verify various ROM locations using the monitor. It is helpful to have the Applesoft and Integer programs stored on disk. If disk I/O works, you can load either program into RAM and verify your firmware using the VERIFY feature of the monitor. Check all the signals on a peripheral slot with a logic probe or oscilloscope.

Intermittent problems are the bain of all computer servicemen as well as computer users. They are often dependent on the temperature and they are often the product of a mechanical defect like imperfect electrical contact. They are sometimes impossible to repair in a cost effective way and the best



Figure 10.3 Some People Just Shouldn't Handle ICs.

thing that can be done in this instance is to swap out the defective assembly at a computer dealership. There are two tricks which can be used to make the problems become more regular so the causes can be identified. One is to subject the equipment to a severe mechanical jolt, like Humphrey Bogart in *The African Queen*. This will tend to prove or disprove the notion that there is a mechanical problem. Less drastically, you can flex the motherboard and reseat suspect peripheral cards and ICs. The second trick is to raise and lower temperature to make the problem occur. "Cold problems" can be made to appear by spraying cold spray, available at electronic stores, on suspected areas. "Hot problems" can be made to appear by directing a heat gun, or, less effectively, a hair dryer at the suspected area.

There are no doubt many symptoms not covered by the guidelines that have been given here. The intention was only to give some helpful hints, not a full blown maintenance aid. Serious readers of this book, however, should be in a very good position to correctly interpret the symptoms of any malfunctions which occur in their machines. In the absence of sophisticated diagnostic aids, full understanding of operation is the most important asset in isolating faults in a digital computer. It is good for Apple owners to possess this level of understanding, and it is also good for them to locate computer dealerships which employ service personnel with this level of understanding.

A Note of Thanks

In preparing this chapter, I benefited from being allowed to spend a day in the excellent repair department at Rainbow Computing, 9719 Reseda Boulevard, Northridge, CA 91324. I was able to discuss Apple maintenance with Roger Wilbur and Eric Waller, who are experts at repairing Apples, and I was also allowed to assist them in some maintenance tasks. Eric has been working on Apples so long that when he first called Apple Computer about a hardware problem, the phone was answered by Steve Job's mother, who called Steve Wozniak in from the garage to answer the phone! While Roger and Eric are not responsible for my interpretations of our discussions, they both contributed measurably to my understanding of Apple maintenance procedures. My thanks to them and to the management of Rainbow Computing.

glossary

address bus. A multi-line electrical connection from the MPU to various devices in a microcomputer by which the MPU specifies the location with which it will communicate. The address bus in the Apple is 16 lines and the MPU can specify 65536 different locations for data transfer.

ampere, amp. The unit of measure of electrical current.

analog. Pertaining to quantities which vary through a continuous range such as a voltage which ranges from +5V to -5V. See digital.

AND gate. A logic gate from which the output will be true if and only if all of its inputs are true.

Applesoft BASIC. The floating point BASIC interpreter language written for the Apple by Microsoft Corp. and distributed by Apple Computer, Inc.

ASCII, American Standard Code for Information Interchange. A code for representing numbers, letters, and symbols in computers. ASCII is used in the Apple for representing text in the keyboard input, text screen map, printer output, and DOS text files.

assembler. A program which converts an assembly language source file into a machine language object file.

assembly language. A language which specifies machine language commands on a one to one basis but in which the computer manages many of the details of generating machine language code.

bank switching. A method of accessing more memory locations than the normal addressing range an MPU will allow. In bank switching, the MPU is allowed by hardware to address more than one memory bank using the same address range. An example is the firmware card, which allows the MPU to access Applesoft or Integer BASIC at the $D000-$FFFF address range.

BASIC, Beginners All-purpose Symbolic Instruction Code. The primary high level language used in personal computers. It was originally developed at Dartmouth College as a training language, and has been developed into a powerful and usable tool by the microcomputer industry.

binary numbering system. A system based on powers of 2, as opposed to powers of 10 in the decimal system. The two symbols of the binary system are 0 and 1. See hexadecimal.

bit. A two state unit of information. The information is in one state or the other—on or off, for example.

bomb, crash. When a program is bombed or when it simply crashes, it loses control of the computer and must be restarted and possibly reloaded. It is particularly likely for a program to crash when it is first written and still has bugs in it.

bootstrap. The process by which a computer loads large operating systems using a small firmware program.

buffer. (1) A temporary holding area in memory in which data resides before, during or after transfer operations. (2) Any hardware device which provides electrical isolation between two electrical points or sets of points.

bus. A multi-line electrical connection which distributes an associated group of signals among two or more communicating devices.

bus driver. A group of amplifiers which allow a group of signals to control a heavily loaded bus. The driver gives electronic leverage to the control signals so they can "drive" many devices.

byte. A group of 8 bits. The 6502 is an 8-bit MPU, thus, it transfers and manipulates data one byte at a time.

BYTE FLAG. A term used in this book to describe the sync bit which leads groups of eight bits in the Apple DOS data formats.

card cage. A row of receptacles into which printed circuit cards with edge connectors are plugged. The receptacles are wired in the back in a method which serves the design purpose of the cage.

cascaded. Being arranged in stages such that each stage depends on the preceding stages. We perform counting and arithmetic in a cascaded numbering system, utilizing carry and borrow processing. In a computer, logic devices can be cascaded to form a larger device conceptually similar to the devices which make it up. For example, four 4-bit counters can be connected in cascade to form a 16-bit counter, as is the case in the Apple's video scanner.

cathode ray tube, CRT. A device in which a screen display is created by a high velocity stream of electrons striking a phosphor coating. The impact point on the screen is controlled by deflecting the electron stream via an electromagnetic or electrostatic field. The picture tube in a television is a CRT.

central processing unit, CPU. The electronic assembly which performs the arithmetic and logical operations of a computer.

chip. See integrated circuit.

color burst. In a television color video signal, a short sample of the color reference signal which occurs just after the horizontal sync pulse. From the color burst, a television or monitor can reconstruct the color reference.

compiler. A program which converts high level language source programs into machine language object programs.

complement. The complement of a binary number is a binary number in which binary 1's replace 0's, and 0's replace 1's in the original number. For example, 11010 is the complement of 00101.

complementary colors. Pertaining to the Apple, colors produced by signals 180 degrees out of phase with each other—HIRES green and HIRES violet for example.

composite video. A complex video display signal containing horizontal and vertical sync, luminance and chrominance signals, and a color burst. The video output of the Apple can loosely be called composite video.

current. The motion of charged particles due to voltage. Generally, in electronics, the movement of electrons through conductive paths. Current is measured in amperes.

data bus. A multi-line electrical connection over which data passes between the MPU and various devices in a microcomputer. The data bus in the Apple is 8 lines, so one byte can be transferred per MPU cycle.

debug. To perfect a program by removing the bugs (defects) from it.

decimal numbering system. The system by which we normally represent numeric quantities. Ten symbols (0-9) represent quantities, while the position of each symbol in a number represents the significance or weight of that symbol. The weight increases by powers of ten as position shifts right to left.

digital. Pertaining to quantities which vary in discrete increments such as integer numbers. See analog.

DIP, dual in line package. A type of IC structure in which the pins run lengthwise in two parallel rows. All ICs in the Apple are DIP ICs.

disassembler. A program which attempts to interpret data in memory as a machine language program and converts it to an assembly language listing. A firmware disassembler in the Apple can be called via the monitor "L" command.

DMA, Direct Memory Access. Direct access to memory from devices other than the MPU. In the Apple, data is directly accessed from memory by the video scanner/video generator combination without passing through the MPU. Additionally, a card in any peripheral slot can directly access memory and other motherboard devices by pulling the DMA' line low.

dot matrix. A method of forming displayed or printed characters in which individual dots at fixed positions in a matrix are displayed as necessary to form the characters.

dynamic memory. Memory in which data will bleed off and lose its validity if it is not regularly refreshed. RAM in the Apple is dynamic and must be refreshed every 2 milliseconds.

Easter egging. A troubleshooting method where possibly failed components or assemblies are replaced with known good units. Easter eggers can sometimes repair equipment they know little or nothing about.

exclusive OR gate. A logic gate from which the output will be true if and only if at least one, but not all, inputs are true.

firmware. Programs and data stored in ROM. Firmware determines many of the operational features of the Apple II.

flag. A memory location used by a program to signify some sort of status. A common way to use a location as a flag is to set or reset its most significant bit.

flip-flop. A 1-bit storage device capable of storing data in response to its logical inputs and a clockpulse. Registers of older computers were comprised of a number of flip-flops with a substantial amount of associated logic gating.

float. If all devices capable of controlling the voltage on an electrical conductor are isolated from the conductor, the conductor is said to float. In the Apple, all conductors on the address bus or data bus can be isolated from control, so these buses sometimes float. Logic which creates this condition floats the bus.

flux. Lines of force used to represent a magnetic field. The lines of force provide a mental picture for visualizing the substance of a magnetic field. In theoretical calculations, field strength is proportional to flux density.

font, character. Patterns of ones and zeroes stored in memory which represent the dot image of dot matrix text or graphics characters.

gate. A logic curcuit having one output and more than one input. Like a gate in a fence, the logic gate allows intelligence to pass when the inputs are correct. AND gates and OR gates are two types of gates. When an input activates a logic device, it is said to "gate" it on.

general purpose computer. A computer whose stored program may be altered to change its purpose. This is normally achieved by storing the program in random access, read/write memory. See special purpose computer.

hacker, hack, computer hack. A person who builds or modifies computer electronic assemblies, mostly for fun.

handler. A program designed to handle a specific occurance such as an interrupt or system reset.

hardware. The components and assemblies of which a computer and its peripherals are made.

Hertz, Hz. A measurement of frequency which used to be referred to more sensibly as cycles per second or cps.

hexadecimal numbering system. A system based on powers of 16, as opposed to the powers of 10 in the decimal system. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

high level language. A computer language whose commands correspond to machine language routines. High level languages are easy to use and powerful. The predominate high level language of the Apple II is Applesoft interpreter BASIC.

horizontal scan. The movement of the electron beam in a television from left to right across the face of the CRT.

impedance. The quality of hindrance of an electrical device to current flow at a given signal frequency or range of signal frequencies. Impedance, like resistance, is measured in ohms. In computer three-state logic, the three states are high voltage, low voltage, and high impedance (high isolation).

input/output, I/O. The process of moving intelligence to and from a computer, as in keyboard input, video output, disk I/O, and printer output.

Integer BASIC. The original BASIC interpreter language supplied with the Apple II. It was written by Steve Wozniak, the principle designer of the Apple II computer.

integrated circuit, IC, chip. An electronic component into which the functions of many other components are integrated. Typically, a chip will be the equivalent of thousands of diodes, transistors, and resistors.

interface. Communication circuitry between two devices, such as the interface between an Apple and a printer.

interlacing. A technique in which alternating television vertical scans are displaced from each other. This increases vertical resolution without introduction of screen flicker.

interpreter. A program which interprets stored sequences of high level commands and executes them via fixed machine language subroutines. The Apple II is supplied with Applesoft and Integer BASIC interpreters.

interrupt. A signal or instruction which, when active, causes a computer to interrupt sequential program execution and branch to an interrupt handling program. The 6502 has four types of interrupts: RESET, the Non-Maskable Interrupt, the Interrupt Request, and the BREAK instruction.

I/O port. The conceptual entry point through which data flows in I/O operations. For example, in the Apple, $C000 is the address of the keyboard input port, and keyboard data is loaded from address $C000.

joystick. A device which converts the two dimensional motion of a lever into measureable electrical equivalents of the X and Y components of the motion. Normally an Apple joystick will be made of two potentiometers, one which responds to y-axis motion of the lever and one which responds to x-axis motion.

least significant bit, LSB. The bit of a binary word or number which has the least weight or significance. The rightmost bit of a binary number.

linear IC. A type of IC used in linear amplification and other analog functions, as opposed to digital switching functions. Some linear ICs used in the Apple are 555, 556, and 558 timers and the 741 cassette input amplifier.

LSTTL, Low Powered Schottky TTL. A type of TTL which provides a good compromise of high speed and low power consumption. Most Apple TTL is LSTTL. The name comes from Schottky-Barrier clamping and coupling diodes inside the IC.

machine cycle. A clocked cycle of an MPU. The machine cycle of the 6502 in the Apple is the period between high to low transitions of the PHASE 2 clock.

machine language. The language of the central processor of a computer (6502 machine language in the Apple).

mainframe computer. When computers were physically large, the structure which held the central processor was called the mainframe. Computers which have such separate structures are mainframe computers.

Megahertz, MHz. One million Hertz. One million cycles per second.

memory cell. A portion of memory capable of storing one bit of information.

memory mapped I/O. A method of I/O implementation in which addresses which might otherwise be assigned to memory are assigned to I/O functions. In the Apple, addresses $C000 to $CFFF are assigned to I/O functions.

memory mapped video. A method of computer video display generation in which a map of the screen display is placed in memory. In the Apple, the MPU builds the screen map in memory, and the memory map is independently scanned for video processing by the video scanner.

microprocessing unit, MPU, microprocessor. The one chip central processing unit of a microcomputer. This definition is a good subject for an argument.

microsecond, μsec. One millionth of a second. One thousand nanoseconds.

millisecond, msec. One thousandth of a second. One thousand microseconds.

modulate. To vary a high frequency signal as a function of a lower frequency signal. The video output signal of the Apple can be used to amplitude modulate a television frequency signal, and that modulated signal can be received by a television set. The high frequency signal carries the video to the TV and is called an RF carrier.

monitor. A program which provides for communication with a computer at a very basic level. Common capabilities include program start up, memory modification, and monitoring of computer registers.

monitor, video. An electronic device which generates a display from a video input signal. It is not capable of television radio frequency signal reception.

monochrome. Of one color. A name for black and white television which accurately describes the fact that there is only one color tone displayed.

MOS, integrated circuit. A chip using metal-oxide semiconductor technology. MOS ICs in the Apple include the 6502, ROM, and RAM.

most significant bit, MSB. The bit of a binary word or number which has the greatest weight. The leftmost bit of a binary number.

motherboard. A printed circuit card into which smaller printed circuit cards can be plugged.

multimeter. An instrument which can measure electrical voltage, resistance, or current.

multiplex. To combine multiple sources of information onto one line. There are numerous examples of time-multiplexing in the Apple in which several possible signals are switched onto a line, one after the other.

NAND gate. A logic gate from which the output will be false if, and only if, all of its inputs are true.

nanosecond, nsec. One billionth of a second.

negative logic. A system of logic analysis in which the high voltage state is considered to be false or zero, and the low voltage state is considered to be true or one. See positive logic.

NOR gate. A logic gate from which the output will be false if, and only if, any of its inputs are true.

object file. The result of an operation which processes a group of data and stores the result elsewhere. In assembly language processing, the assembly language source file is assembled into a machine language object file.

octal numbering system. A system based on powers of 8, as opposed to the powers of 10 in the decimal system. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7.

ohm. The unit for measuring electrical resistance and impedance.

ohmmeter. A device which measures electrical resistance. Usually resistance is measured with a VOM (Volt-Ohm Meter) or multimeter which performs other functions besides resistance measurement.

op code, operation code. The part of a machine language instruction which specifies the command which is to be performed. The op code of each 6502 instruction is the first byte of that instruction.

operand. The entity operated on by a machine language instruction. In "LDA $00", the operand is the contents of memory location 0. In "SEC", the operand is the carry bit of the 6502 Status Register.

OR gate. A logic gate from which the output will be true if, and only if, any of its inputs are true.

oscilloscope. A test instrument which produces a cathode ray tube display of a test voltage, plotted against time.

PAGE. (1) The 6502 memory addressing range of $10000 bytes is divided up into $100 pages of $100 bytes each. $000-$0FF is PAGE 0; $100-$1FF is PAGE 1; etc. (2) There are four areas of memory which can be scanned for video output in the Apple. These are TEXT/LORES PAGE 1, TEXT/LORES PAGE 2, HIRES PAGE 1, and HIRES PAGE 2. In any screen mode, the PAGE 1 memory map may be selected for scanning via programmed reference to $C054, and the PAGE 2 memory map can be selected by reference to $C055.

parallel data transfer. Simultaneous transfer of n bits of data on n lines, as in 8-bit parallel data transfer between the MPU and memory in the Apple.

peripheral slots, peripheral bus, Apple bus. The eight slots in the back of the Apple and their associated electrical connections.

phase. The angular position of a cyclical event referenced to some event of the same frequency. For example, HIRES violet video is 180 degrees out of phase with HIRES green video.

pipelining. A process by which program execution speed is increased in the 6502. In pipelining, the next instruction's op code is fetched during the last execution cycle of instructions which do not write to the data bus.

positive logic. A system of logical analysis in which the high voltage state is considered to be true or one, and the low voltage state is considered to be false or zero. The Apple and most modern computers use positive logic. See negative logic.

potentiometer, pot. A mechanically variable resistor. Typically, resistance will be proportional to the shaft rotation of the pot. The Apple paddles are pots.

power supply. An electronic assembly which converts an AC (alternating current) line voltage to usable DC (direct current) power. The Apple power supply converts household power to +12V, -12V, +5V, and -5V referenced to ground.

printed circuit card, PC board. A thin card made of an insulating material upon which electronic components are mounted. The component wiring is "printed" on the board. The printing process involves starting with a card completely coated with conductive metal, then etching away everything but the desired conductive parts using photo-chemical methods.

program counter. A counter in a computer which contains the memory address of the instruction being executed. The 6502 has an internal 16-bit program counter.

propagation delay. The time it takes for a signal or voltage to travel between two points. In logic gating, the time required for an output to respond to a change in associated inputs.

RAM, alterable memory, read/write memory. Memory in which a computer can store or access data.

random access memory. Memory in which any location can be accessed at will, such as the RAM and ROM in the Apple. See serial access memory.

raster. The pattern of scan lines produced on the screen of a monitor or television. The Apple raster contains 262 lines with no interlacing.

read cycle. A machine cycle in which the MPU receives data from the addressed location via the data bus.

refresh. (1) The process of renewing data in dynamic memory before it bleeds off. (2) The process of renewing an image on a cathode ray tube by rescanning the image before it fades away.

register. A temporary storage device which holds more than one bit of information. It will normally serve some special logic purpose. Examples include the 6502 internal registers and the data register of the Disk II controller.

relocatable program. A program which does not have to be in one specific memory range to be properly executed. It can be relocated to different memory areas for execution.

resistance. The quality of hindrance of an electrical device to direct current flow. Resistance in a current path may be controlled by installing fixed or variable resistors. The unit of measurement of electrical resistance is the ohm.

RF modulator. A device which varies a high frequency signal as a function of a lower frequency signal. See modulate.

ROM, read only memory, non-volatile memory, non-alterable memory. A type of memory which the computer cannot write to or otherwise alter. It holds programs and data which are always available when power is applied, such as BASIC and the monitor in the Apple.

serial access memory. Memory which can only be accessed by sequencing through locations until the correct location is found. High speed magnetic tape and bubble memory are examples of serial memories. See random access memory.

serial data transfer. Transferring data one bit at a time over a single line, as in the shifting of text patterns to the PICTURE signal.

software. Programs and data stored in RAM and on storage media such as disks.

source file. A source of data for data processing. In assembly language processing, the assembly language source file is assembled into a machine language object file.

special purpose computer. A computer whose function cannot be changed by altering a stored program. The functions of a special purpose computer may be hard wired, or the computer may execute fixed programs stored in ROM.

stack. In microcomputers, an area of memory set aside for temporary storage and subroutine return link information. In programming, the stack is conceptually similar to a stack of cards which can be drawn from or discarded to, one card at a time.

static memory. Memory which requires no refreshing to retain its data, such as the static ROM in the Apple.

status register. A register in a central processor which contains control information. In the 6502, the status register contains indicators of the logical results of various executed commands.

strobe. A short pulse that performs a triggering or clocking action. Strobes in the Apple include RAS', CAS', the C040 STROBE', and the keypress strobe from the keyboard.

television sync. That part of the television signal which synchronizes the scanning of the electron beam in the CRT. It includes horizontal and vertical sync.

tri-state logic, three-state logic. A logic system in which there are three states: high voltage, low voltage, and high impedance. Devices connected to the Apple data bus have tri-state outputs so the various devices are able to share control of the data bus.

troubleshoot. To isolate and repair the casualties in a failed piece of hardware.

TTL, Transistor Transistor Logic. The logic family to which most general purpose ICs in the Apple belong. Both the inputs and outputs of TTL chips are connected to transistors inside the chip. As opposed to MOS devices like RAM, ROM, and the 6502, TTL circuits are made using bipolar technology.

underware. A substitute for firmware used by some of Apple's competitors to cut costs.

vector. An address or jump instruction stored in a memory location which contains program flow information in case of certain events. An example is the interrupt vectors stored in high memory in a 6502 based computer.

vertical scan. The movement of the electron beam in a television down the face of the CRT. In the Apple, 262 horizontal scans occur during every vertical scan.

video. A signal which can be used to control the energy of the electron beam of a CRT, thus controlling display intensity, as in television video, radar video, and oscilloscope video. In television processing, the combination of picture, sync, and color information is commonly referred to as video.

volt. A unit for measuring voltage. Voltages of +12, -12, +5, and -5 volts are distributed throughout the Apple.

voltage, electromotive force, EMF. A force of nature which, when present, causes charged particles to move. The force which causes electric current. Voltage is measured in volts.

wetware. A gray matter found within the cranium of most humans.

wire-OR, collector-OR. A low level OR gate formed by wiring various signals together. The RESET', IRQ', NMI', INHIBIT', DMA', and USER1' signals of the Apple II are examples of wire-OR connections. Any peripheral card may bring any of the wire-OR lines low, but cards not bringing a line low must present a high impedance to that line. If no peripheral card is bringing a wire-OR line low, a 1000 ohm motherboard resistor will pull the line high.

write cycle. A machine cycle in which the MPU sends data to the addressed location via the data bus.

appendix A

References

Apple Computer, Inc. *Apple II Reference Manual*. January 1978.

Apple Computer, Inc. *Applesoft II BASIC Programming Reference Manual*. 1978.

Apple Computer, Inc. *The Applesoft Tutorial*. 1979.

Apple Computer, Inc. (Christopher Espinoza). *Apple II Reference Manual*.1979.

Apple Computer, Inc. (originally by Phyllis Cole and Brian Howard). *The DOS Manual*. 1980.

Apple Computer, Inc. (Allen Watson). *Apple II Reference Manual for IIe Only*. 1982.

Bishop, Bob. "Have an Apple Split." *SOFTALK*, p. 54, Oct. 1982.

Brown, Chris and Maloney, Eric. "FCC Takes Aim Against RFI Polluters." *Microcomputing*, p. 30, April 1981.

Ciotti, Paul. "Revenge of the Nerds." *California*, p. 73, July 1982.

Derfler, Frank J. "Trying to Live in Harmony with Harmonics." *Microcomputing*, p. 36, April 1981.

Devry Institute of Technology. *Electronics Technology*. Volumes 7 and 10, 1971. This textbook was the primary source of information relating to television.

Gayler, Winston D. *The Apple II Circuit Description*. Howard W. Sams & Co, Inc., 1983.

Lancaster, Don. *TTL Cookbook*. Howard W. Sams & Co, Inc., 1974.

Leventhal, Lance A. *6502 Assembly Language Programming*. OSBORNE/McGraw-Hill, 1979.

Mazur, Jeffrey. "HARDTALK." *SOFTALK*, p. 46, July 1982. Concerns EPROM.

Mazur, Jeffrey. "HARDTALK." *SOFTALK*, p. 168, Aug. 1982. Concerns EPROM.

Mazur, Jeffrey. "HARDTALK." *SOFTALK*, p. 208, Sept. 1982. Concerns disk drives.

Moore, Robin B. "Graphics Character Generator." *Microcomputing*, p. 108, Aug. 1980.

MOS Technology, Inc. *MCS6500 Microcomputer Family Hardware Manual*. 1976.

Osborne, Adam. *An Introduction to Microcomputers, Volume 1, Basic Concepts*. Adam Osborne and Associates, 1976.

Osborne, Adam and Kane, Gerry. *Osborne 4 & 8 bit Microprocessor Handbook*. OSBORNE/McGraw-Hill, 1981.

Sippl, Charles J. *Microcomputer Dictionary*. 2nd edition, Howard W. Sams & Co, Inc., 1981.

Synertek, Inc. *SY6500/MCS6500 Microcomputer Family Programming Manual*. Aug. 1976.

Synertek, Inc. *Applications Information AN2, SY6500 Microprocessor Family, Microprocessor Products*. Oct. 1981.

Texas Instruments, Inc. *Designing with TTL Integrated Circuits*. 1971.

White, Robert M. "Disk-Storage Technology." *Scientific American*, p. 112, Aug. 1980.

Williams, Richard. "How to Use the Hooks." *MICRO*, p. 30:7, Nov. 1980.

Worth, Don and Lechner, Pieter. *Beneath Apple DOS*. Quality Software, 1981.

Wozniak, Stephen. "The Apple II: System Description." *BYTE*, p. 34, May 1977.

Wozniak, Stephen. "SWEET16: The 6502 Dream Machine." *BYTE*, p. 150, Nov. 1977.

Wozniak, Stephen. Speech made at Applefest. Anaheim, CA, April 17, 1983.

Component Data:

Fairchild. *MOS Memory Data Book*. 1981.

Fujitsu Microelectronics. *MOS 16384-Bit Dynamic Random Access Memory*. July, 1981.

General Instrument Corp. *Microelectronics Data Catalog*. 1982.

General Instrument Corp. *ROM 3*. no date.

Hitachi America, Ltd. *IC Memories*. no date.

Jameco Electronics. *1983 Catalog*. p. 12.

MOS Technology, Inc. *6500 Microprocessors*. March 1980.

Motorola Semiconductors. *MC3470 Floppy Disk Read Amplifier System*. 1978.

National Semiconductor. *MM5740 90-Key Keyboard Encoder*. 1973.

National Semiconductor. *Interface Databook*. 1980.

National Semiconductor. *Memory Databook*. 1980.

National Semiconductor. *Logic Databook*. 1981.

National Semiconductor. *Linear Databook*. 1982.

Rockwell International. *R6500 Microcomputer System Data Sheet*. Rev. 4, Nov. 1981.

Synertek. *1981-1982 Data Catalog*.

Texas Instruments, Inc. *The TTL Data Book for Design Engineers*. 1976.

United Technical Publications. *IC MASTER*. 1981.

appendix **B**

Trademarks

The following is a list of Registered Trademarks referred to in the text of *Understanding the Apple II*.

Apple	Apple Computer, Inc.
Apple II	Apple Computer, Inc.
Apple II Plus	Apple Computer, Inc.
Apple IIe	Apple Computer, Inc.
Applesoft	Apple Computer, Inc.
CP/M	Digital Research, Inc.
Donkey Kong	Nintendo
Microsoft	Microsoft, Inc.
Softcard	Microsoft, Inc.
TRI-STATE	National Semiconductor Corporation
Z80	Zilog, Inc.

appendix C

6502 Data

In attempting to analyze Apple timing, it is discouraging to find that the three 6502 manufacturers have different specifications, even though the MSC6502, R6502, and SY6502 should all perform identically. It is even more discouraging to find that the specifications are well beyond the range of typical operations. I therefore feel that Figure 4.5, which shows some measurements made in an SY6502 in an Apple, is a more realistic indicator of 6502 timing than the manufacturers' data sheets. Nevertheless, a partial reproduction of Rockwell International's data sheet is given here. This data is reprinted with the permission of Rockwell International Corporation, Copyright 1981, all rights reserved. Rockwell's timing specification charts are followed by the author's compilation of the 1 MHz timing charts of the Synertek, Rockwell International, and MOS Technology data sheets. Every attempt was made to make the data in this compilation faithfully represent the data contained in each manufacturer's data sheet. The final page of this appendix is a layout of the author's which shows the execution periods of the various 6502 instructions.

Rockwell

R6500 Microcomputer System DATA SHEET

R6500 MICROPROCESSORS (CPU)

SYSTEM ABSTRACT

The 8-bit R6500 microcomputer system is produced with N-Channel, Silicon Gate technology. Its performance speeds are enhanced by advanced system architecture. This innovative architecture results in smaller chips – the semiconductor threshold is cost-effectivity. System cost-effectivity is further enhanced by providing a family of 10 software-compatible microprocessor (CPU) devices, described in this document. Rockwell also provides memory and microcomputer system– as well as low-cost design aids and documentation.

R6500 MICROPROCESSOR (CPU) CONCEPT

Ten CPU devices are available. All are software-compatible. They provide options of addressable memory, interrupt input, on-chip clock oscillators and drivers. All are bus-compatible with earlier generation microprocessors like the M6800 devices.

The family includes six microprocessors with on-board clock oscillators and drivers and four microprocessors driven by external clocks. The on-chip clock versions are aimed at high performance, low cost applications where single phase inputs, crystal or RC inputs provide the time base. The external clock versions are geared for multiprocessor system applications where maximum timing control is mandatory. All R6500 microprocessors are also available in a variety of packaging (ceramic and plastic), operating frequency (1 MHz, 2 MHz and 3 MHz) and temperature (commercial and industrial) versions.

MEMBERS OF THE R6500 MICROPROCESSOR (CPU) FAMILY

Microprocessors with Internal Two Phase Clock Generator

Model	Addressable Memory
R6502	64K Bytes
R6503	4K Bytes
R6504	8K Bytes
R6505	4K Bytes
R6506	4K Bytes
R6507	8K Bytes

Microprocessors with External Two Phase Clock Input

Model	Addressable Memory
R6512	64K Bytes
R6513	4K Bytes
R6514	8K Bytes
R6515	4K Bytes

FEATURES

- Single +5V supply
- N channel, silicon gate, depletion load technology
- Eight bit parallel processing
- 56 Instructions
- Decimal and binary arithmetic
- Thirteen addressing modes
- True indexing capability
- Programmable stack pointer
- Variable length stack
- Interrupt capability
- Non-maskable interrupt
- Use with any type of speed memory
- 8-bit Bidirectional Data Bus
- Addressable memory range of up to 64K bytes
- "Ready" input
- Direct Memory Access capability
- Bus compatible with M6800
- 1 MHz, 2 MHz, and 3 MHz versions
- Choice of external or on-chip clocks
- On-the-chip clock options
  - External single clock input
  - Crystal time base input
- Commercial and industrial temperature versions
- Pipeline architecture

Ordering Information

Order Number: R65XX _ _ _

- Temperature Range:
  - No suffix = 0°C to +70°C
  - E = -40°C to +85°C (Industrial)
- Package: C = Ceramic, P = Plastic
- Frequency Range:
  - No suffix = 1 MHz
  - A = 2 MHz
  - B = 3 MHz
- Model Designator:
  - XX = 02, 03, 04, . . . 15

R6500 Signal Description

Clocks (ϕ_1, ϕ_2)

The R651X requires a two phase non-overlapping clock that runs at the V_{CC} voltage level.

The R650X clocks are supplied with an internal clock generator. The frequency of these clocks is externally controlled.

Address Bus (A0-A15)

These outputs are TTL compatible, capable of driving one standard TTL load and 130 pF.

Data Bus (D0-D7)

Eight pins are used for the data bus. This is a bidirectional bus, transferring data to and from the device and peripherals. The outputs are tri-state buffers capable of driving one standard TTL load and 130 pF.

Data Bus Enable (DBE)

This TTL compatible input allows external control of the tri-state data output buffers and will enable the microprocessor bus driver when in the high state. In normal operation DBE would be driven by the phase two (ϕ_2) clock, thus allowing data output from microprocessor only during ϕ_2. During the read cycle, the data bus drivers are internally disabled, becoming essentially an open circuit. To disable data bus drivers externally, DBE should be held low.

Ready (RDY)

This input signal allows the user to halt or single cycle the microprocessor on all cycles except write cycles. A negative transition to the low state during or coincident with phase one (ϕ_1) will halt the microprocessor with the output address lines reflecting the current address being fetched. If Ready is low during a write cycle, it is ignored until the following read operation. This condition will remain through a subsequent phase two (ϕ_2) in which the Ready signal is low. This feature allows microprocessor interfacing with the low speed PROMs as well as Direct Memory Access (DMA).

Interrupt Request ($\overline{IRQ}$)

This TTL level input requests that an interrupt sequence begin within the microprocessor. The microprocessor will complete the current instruction being executed before recognizing the request. At that time, the interrupt mask bit in the Status Code Register will be examined. If the interrupt mask flag is not set, the microprocessor will begin an interrupt sequence. The Program Counter and Processor Status Register are stored in the stack. The microprocessor will then set the interrupt mask flag high so that no further interrupts may occur. At the end of this cycle, the program counter low will be loaded from address FFFE, and program counter high from location FFFF, therefore transferring program control to the memory vector located at these addresses. The RDY signal must be in the high state for any interrupt to be recognized. A 3KΩ external resistor should be used for proper wire-OR operation.

Non-Maskable Interrupt ($\overline{NMI}$)

A negative going edge on this input requests that a non-maskable interrupt sequence be generated within the microprocessor.

$\overline{NMI}$ is an unconditional interrupt. Following completion of the current instruction, the sequence of operations defined for $\overline{IRQ}$ will be performed, regardless of the state interrupt mask flag. The vector address loaded into the program counter, low and high, are locations FFFA and FFFB respectively, thereby transferring program control to the memory vector located at these addresses. The instructions loaded at these locations cause the microprocessor to branch to a non-maskable interrupt routine in memory.

$\overline{NMI}$ also requires an external 3K Ω register to V_{CC} for proper wire-OR operations.

Inputs $\overline{IRQ}$ and $\overline{NMI}$ are hardware interrupts lines that are sampled during ϕ_2 (phase 2) and will begin the appropriate interrupt routine on the ϕ_1 (phase 1) following the completion of the current instruction.

Set Overflow Flag (S.O.)

A negative going edge on this input sets the overflow bit in the Status Code Register. This signal is sampled on the trailing edge of ϕ_1 and must be externally synchronized.

SYNC

This output line is provided to identify those cycles in which the microprocessor is doing an OP CODE fetch. The SYNC line goes high during ϕ_1 of an OP CODE fetch and stays high for the remainder of that cycle. If the RDY line is pulled low during the ϕ_1 clock pulse in which SYNC went high, the processor will stop in its current state and will remain in the state until the RDY line goes high. In this manner, the SYNC signal can be used to control RDY to cause single instruction execution.

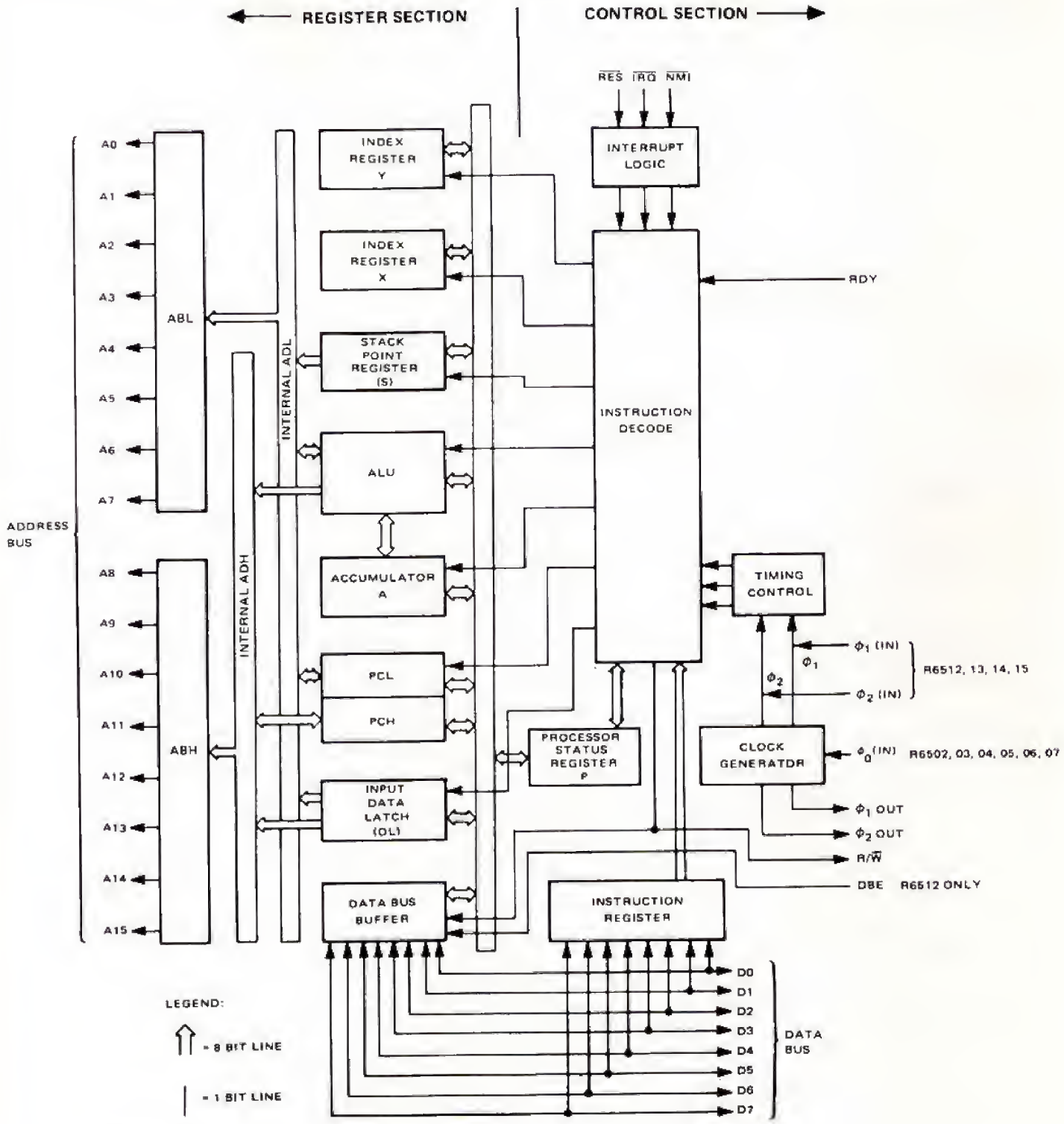
Reset

This input is used to reset or start the microprocessor from a power down condition. During the time that this line is held low, writing to or from the microprocessor is inhibited. When a positive edge is detected on the input, the microprocessor will immediately begin the reset sequence.

After a system initialization time of six clock cycles, the mask interrupt flag will be set and the microprocessor will load the program counter from the memory vector locations FFFC and FFFD. This is the start location for program control.

After V_{CC} reaches 4.75 volts in a power up routine, reset must be held low for at least two clock cycles. At this time the R/W and (SYNC) signal will become valid.

When the reset signal goes high following these two clock cycles, the microprocessor will proceed with the normal reset procedure detailed above.



Note: 1. Clock Generator is not included on R6512, 13, 14, 15
2. Addressing Capability and control options vary with each of the R6500 Products.

R6500 Internal Architecture

SPECIFICATIONS

Maximum Ratings

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	Vdc
Input Voltage	V_{in}	-0.3 to +7.0	Vdc
Operating Temperature	T		°C
Commercial		0 to +70	
Industrial		-40 to +85	
Storage Temperature	T_{STG}	-55 to +150	°C

NOTE
This device contains input protection against damage to high static voltages or electric fields; however, precautions should be taken to avoid application of voltages higher than the maximum rating.

Electrical Characteristics

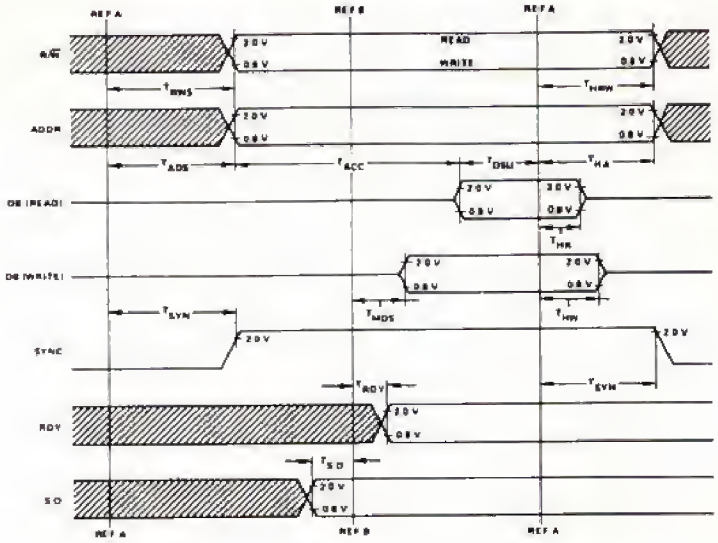
(V_{CC} = 5.0 ±5%, V_{SS} = 0)

ϕ_1, ϕ_2 applies to R6512, 13, 14, 15, $\phi_{o(in)}$ applies to R6502, 03, 04, 05, 06 and 07.

Characteristic	Symbol	Min	Max	Unit
Input High Voltage	V_{IH}			Vdc
Logic, $\phi_{o(in)}$		2.0	V_{CC}	
ϕ_1, ϕ_2		-0.3	V_{CC} + 0.25	
Input Low Voltage	V_{IL}			Vdc
Logic, $\phi_{o(in)}$		-0.3	0.8	
ϕ_1, ϕ_2		-0.3	0.4	
Input Leakage Current	I_{in}			μA
(V_{in} = 0 to 5.25V, V_{CC} = 0)				
Logic (Excl. Rdy, S.O.)		–	2.5	
ϕ_1, ϕ_2		–	100	
$\phi_{o(in)}$		–	10.0	
Three-State (Off State) Input Current	I_{TSI}			μA
(V_{in} = 0.4 to 2.4V, V_{CC} = 5.25V)				
Data Lines		–	10	
Output High Voltage	V_{OH}			Vdc
(I_{LOAD} = -100 μAdc, V_{CC} = 4.75V)				
SYNC, Data, A0-A15, R/$\overline{W}$, ϕ_1, ϕ_2		V_{SS} + 24	–	
Output Low Voltage	V_{OL}			Vdc
(I_{LOAD} = 1.6 mAdc, V_{CC} = 4.75V)				
SYNC, Data, A0-A15, R/$\overline{W}$, ϕ_1, ϕ_2			V_{SS} + 0.4	
Power Dissipation	P_D			mW
1 and 2 MHz		–	700	
3 MHz		–	800	
Capacitance at 25°C	C			pF
(V_{in} = 0, f = 1 MHz)				
Logic	C_{in}	–	10	
Data		–	15	
A0-A15, R/W, SYNC	C_{out}	–	12	
$\phi_{o(in)}$	$C\phi_{o(in)}$	–	15	
ϕ_1	$C\phi_1$	–	50	
ϕ_2	$C\phi_2$	–	80	

NOTE
$\overline{IRQ}$ and $\overline{NMI}$ require 3K pull-up resistor.

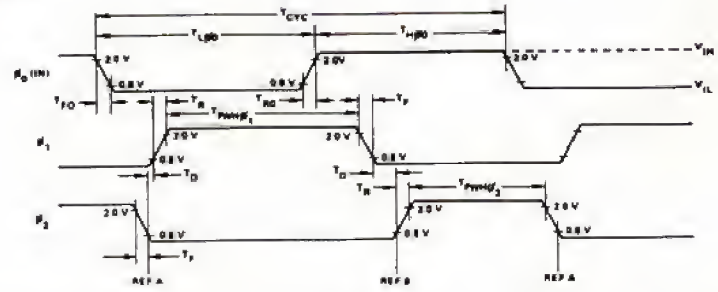
R65XX Timing



Characteristic	Symbol	1 MHz		2 MHz		3 MHz		Unit
		Min	Max	Min	Max	Min	Max	
R/$\overline{W}$ Setup Time	T_{RWS}	–	225	–	140	–	110	ns
R/$\overline{W}$ Hold Time	T_{HRW}	30	–	30	–	15	–	ns
Address Setup Time	T_{ADS}	–	225	–	140	–	110	ns
Address Hold Time	T_{HA}	30	–	30	–	15	–	ns
Read Access Time	T_{ACC}	–	650	–	310	–	170	ns
Read Data Setup Time	T_{DSU}	50	–	40	–	35	–	ns
Read Data Hold Time	T_{HR}	10	–	10	–	10	–	ns
Write Data Setup Time	T_{MDS}	–	175	–	100	–	85	ns
Write Data Hold Time	T_{HW}	30	–	30	–	30	–	ns
SYNC Hold Time	T_{SYH}	30	–	30	–	15	–	ns
RDY Setup Time*	T_{RDY}	100	–	50	–	35	–	ns
S.O. Setup Time	T_{SO}	100	–	50	–	35	–	ns
SYNC Setup Time	T_{SYN}	–	225	–	175	–	100	ns

NOTE
*RDY must never switch states within T_{RDY} to end of $\phi 2$
LOAD = 130 pF + 1 TTL

R650X CPU Clock Timing



Characteristic	Symbol	1 MHz		2 MHz		3 MHz		Unit
		Min	Max	Min	Max	Min	Max	
Cycle Time	T_{CYC}	1.0	10	0.5	10	0.33	10	µs
$\phi_{o(In)}$ Low Time	$T_{L\phi O}$	480	–	240	–	160	–	ns
$\phi_{o(In)}$ High Time	$T_{H\phi O}$	470	–	240	–	160	–	ns
$\phi_{o(In)}$ Rise and Fall Time*	T_{RO}, T_{FO}	–	10	–	10	–	10	ns
ϕ_1 Pulse Width	$T_{PWH\phi_1}$	460	–	235	–	155	–	ns
ϕ_2 Pulse Width	$T_{PWH\phi_2}$	470	–	240	–	160	–	ns
Delay Between ϕ_1 and ϕ_2	T_D	0	–	0	–	0	–	ns
ϕ_1, ϕ_2 Rise and Fall Time*	T_R, T_F	–	25	–	25	–	15	ns

NOTE
*Measured between 0.8 and 2.0 points on waveform load 130 pF + 1 TTL.

Table C.1 6502 Timing Comparisons.

PARAMETER	SYMBOL	SYNERTEK		ROCKWELL		MOS TECHNOLOGY			UNITS
		MIN	MAX	MIN	MAX	MIN	TYP	MAX	
CYCLE TIME	TCYC	1.0	40	1.0	10	1.0	-	---	us
PHS0 LOW TIME	TL0	480(2)	---	480	---	---	-	---	ns
PHS0 HIGH TIME	TH0	460(2)	---	470	---	---	-	---	ns
PHS0 PULSE WIDTH	PWH0	---	---	---	---	460(7)	-	520(7)	ns
PHS0 RISE/FALL TIME	TR0/TF0	0(1)	30	---	10(6)	---	-	10	ns
PHS1 PULSE WIDTH	TPWH1	TL0-20	TL0	460	---	TL0-20(7)	-	TL0(7)	ns
PHS2 PULSE WIDTH	TPWH2	TL0-40	TL0-10	470	---	TL0-40(7)	-	TL0-10(7)	ns
DELAY BETWEEN PHS1 AND PHS2	TD	5	---	0	---	5(7)	-	---	ns
PHS1, PHS2 RISE/FALL TIMES	TR/TF	---	25(1)(3)	---	25(6)	---	-	25(8)	ns
PHS0 NEG TO PHS1 POS DELAY	T01+	10(5)	70(5)	---	---	---	-	---	ns
PHS0 NEG TO PHS2 NEG DELAY	T02-	5(5)	65(5)	---	---	---	-	---	ns
PHS0 POS TO PHS1 NEG DELAY	T01-	5(5)	65(5)	---	---	---	-	---	ns
PHS0 POS TO PHS2 POS DELAY	T02+	15(5)	75(5)	---	---	---	-	---	ns
R/W' SETUP TIME	TRWS	---	225	---	225	---	100	300	ns
R/W' HOLD TIME	TRWH	30	---	30	---	30	60	---	ns
ADDRESS SETUP TIME	TADS	---	225	---	225	---	100	300	ns
ADDRESS HOLD TIME	TADH	30	---	30	---	30	60	---	ns
READ ACCESS TIME	TACC	---	650	---	650	---	-	575	ns
READ DATA SETUP TIME	TDSU	100	---	50	---	100	-	---	ns
READ DATA HOLD TIME	THR	10	---	10	---	10	-	---	ns
WRITE DATA SETUP TIME	TMDS	20	175	---	175	---	150	200	ns
WRITE DATA HOLD TIME	THW	60	150	30	---	30	60	---	ns
SYNC SETUP TIME	TSYS	---	350	---	225	---	-	350	ns
SYNC HOLD TIME	TSYH	30	---	30	---	---	-	---	ns
RDY SETUP TIME	TRS	200(4)	---	100(4)	---	100	-	---	ns
S.O. SETUP TIME	TSO	---	---	100	---	100	-	---	ns

(1) Measured between 10% and 90% points on waveform.
(2) Measured at 50% points.
(3) Load = 1 TTL load +30 pF.
(4) RDY must never switch states within TRS to end of PHS2.
(5) Load = 100 pF.
(6) Measured between .8v and 2.0v points on waveform, load 130 pF + 1 TTL.
(7) Measured at 1.5v.
(8) Measured .8v to 2.0v. Load 1/2 30pF 1/3 1 TTL.

Table C.2 6502 Instruction Execution Periods in Machine Cycles.

	IMP	REL	IMM	ACC	ØPG	ØPG X	ØPG Y	ABS	ABS X	ABS Y	IND	IND X	IND Y
ADC AND CMP EOR LDA ORA SBC			2		3	4		4	4*	4*		6	5*
ASL LSR ROL ROR				2	5	6		6	7				
BCC BCS BEQ BMI BNE BPL BVC BVS		2+n **											
CLC CLD CLI CLV DEX DEY INX INY NOP SEC SED SEI TAX TAY TSX TXA TXS TYA	2												
BIT					3			4					
BRK	7												
CPX CPY			2		3			4					
DEC INC					5	6		6	7				
JMP								3			5		
JSR								6					
LDX			2		3		4	4		4*			
LDY			2		3	4		4	4*				
PHA PHP	3												
PLA PLP	4												
RTI	6												
RTS	6												
STA					3	4		4	5	5		6	6
STX					3		4	4					
STY					3	4		4					

* +1 cycle if indexing crosses page boundary.
** n=0 if branch does not occur.
n=1 if branch within page occurs.
n=2 if branch across page boundary occurs.

appendix D

BASIC Program Listings

The following pages contain the BASIC program listings which produce Figures 5.8 and 5.9.

```
0  REM
1  REM
2  REM
3  REM  HIRES MEMORY MAP - DISPLAYED SCAN ONLY.
4  REM
5  REM
6  REM
10  DIM HX$(15)
20  DATA  "0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"
30  FOR A = 0 TO 15: READ HX$(A): NEXT A
40  TEXT : HOME
50  PRINT "                                TOP SCREEN/         MIDDLE SCREEN/      BOTTOM SCREEN/"
60  PRINT "                                 FIRST 40            SECOND 40           THIRD 40        UNUSED 8"
70  PRINT "   PAGE 1      PAGE 2   LIN# PAGE 1 RANGE   LIN# PAGE 1 RANGE   LIN# PAGE 1 RANGE   PAGE 1 RANGE"
90  REM
91  REM
100  FOR A = 0 TO 7: FOR B = 0 TO 7:BASE = A * 128 + B * 1024 + 8192
110 LN$ = "$":DEC = BASE: GOSUB 5000: REM GET PAGE 1 HEX
190  REM
191  REM
200 DEC$ =  STR$ (BASE): IF  LEN (DEC$) < 5 THEN LN$ = LN$ + " "
210 LN$ = LN$ + " " + DEC$ + " $"
220 DEC = BASE + 8192: GOSUB 5000: REM GET PAGE 2 HEX
230 LN$ = LN$ + " " +  STR$ (DEC) + " "
290  REM
291  REM
300 SCAN = 8 * A + B: GOSUB 6000: REM  GET SCAN NUMBER
310 DEC = BASE: GOSUB 5000
320 LN$ = LN$ + "-$"
330 DEC = BASE + 39: GOSUB 5000
340 LN$ = LN$ + "    "
390  REM
391  REM
400 SCAN = SCAN + 64: GOSUB 6000
410 DEC = BASE + 40: GOSUB 5000
420 LN$ = LN$ + "-$"
430 DEC = BASE + 79: GOSUB 5000
440 LN$ = LN$ + "    "
490  REM
491  REM
500 SCAN = SCAN + 64: GOSUB 6000
510 DEC = BASE + 80: GOSUB 5000
520 LN$ = LN$ + "-$"
530 DEC = BASE + 119: GOSUB 5000
590  REM
591  REM
600 LN$ = LN$ + "     $"
610 DEC = BASE + 120: GOSUB 5000
620 LN$ = LN$ + "-$"
630 DEC = BASE + 127: GOSUB 5000
640  PRINT LN$: NEXT B: NEXT A
650  PRINT : PRINT
660  PRINT "FIGURE 5.8 - HIRES DISPLAYED MEMORY MAP.  PAGE 1 AND PAGE 2 ARE EACH MADE UP OF 64 128-BYTE MEMORY SEGMENTS."
670  GET B$: END
4090  REM
4091  REM
4092  REM
4093  REM  SUBROUTINE 5000 CONVERTS THE DECIMAL ADDRESS IN DEC TO
4094  REM  HEXADECIMAL AND CONCATINATES THE HEX NUMBER TO LN$.
4095  REM
4096  REM
4097  REM
5000 H4 = DEC / 4096:H4% = H4
5010 H3 = (H4 - H4%) * 16:H3% = H3
5020 H2 = (H3 - H3%) * 16:H2% = H2
5030 H1% = (H2 - H2%) * 16 + .5
5040 LN$ = LN$ + HX$(H4%) + HX$(H3%) + HX$(H2%) + HX$(H1%)
5050  RETURN
5190  REM
5191  REM
5192  REM
5193  REM   SUBROUTINE 6000 ADDS LEADING ZEROES TO THE SCAN #
5194  REM
5195  REM
5196  REM
6000  IF SCAN < 100 THEN LN$ = LN$ + "0"
6010  IF SCAN < 10 THEN LN$ = LN$ + "0"
6020 LN$ = LN$ +  STR$ (SCAN) + "  $"
6030  RETURN
```

Figure D.1 BASIC Listing: Program that Produces Figure 5.8.

```
0  REM
3  REM  FOUR PAGE HIRES MEMORY MAP - INCLUDES VBL & HBL.
6  REM
10  DIM HX$(15)
20  DATA  "0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"
30  FOR A = 0 TO 15: READ HX$(A): NEXT A
40 SCAN =  -1
100  REM
102  PRINT  TAB( 56)"SCREEN TOP": PRINT : PRINT
110  GOSUB 4000: REM  PRINT HEADINGS
120  FOR A = 0 TO 7: FOR B = 0 TO 7:BASE = A * 128 + B * 1024 + 8192
130  GOSUB 1000: REM  DO LINE
140  NEXT B: NEXT A
150  GOSUB 5000: GET A$
200  REM  ********************
202  PRINT : PRINT : PRINT  TAB( 55)"SCREEN MIDDLE": PRINT : PRINT
210  GOSUB 4000
220  FOR A = 0 TO 7: FOR B = 0 TO 7:BASE = A * 128 + B * 1024 + 8232
230  GOSUB 1000
240  NEXT B: NEXT A
250  GOSUB 5000: GET A$
300  REM  ********************
301  PRINT : PRINT : PRINT  TAB( 55)"SCREEN BOTTOM": PRINT : PRINT
310  GOSUB 4000
320  FOR A = 0 TO 7: FOR B = 0 TO 7:BASE = A * 128 + B * 1024 + 8272
330  GOSUB 1000
340  NEXT B: NEXT A
350  GOSUB 5000: GET A$
400  REM  ********************
401  PRINT : PRINT : PRINT  TAB( 46)"VERTICAL BLANKING PERIOD (VBL)": PRINT : PRINT
410  GOSUB 4000
420  FOR A = 0 TO 7: FOR B = 0 TO 7:BASE = A * 128 + B * 1024 + 8312
430  GOSUB 1000
440  NEXT B: NEXT A
450  FOR B = 2 TO 7:BASE = 9208 + B * 1024: REM 7*128+8312+B*1024
460  GOSUB 1000
470  NEXT B
480  GOSUB 5000: GET A$
490  END
1000  REM
1002  REM  PRINT A LINE
1004  REM
1010 LN$ = "":SCAN = SCAN + 1: IF SCAN < 10 THEN LN$ = LN$ + " "
1020  IF SCAN < 100 THEN LN$ = LN$ + " "
1025 LN$ = LN$ +  STR$ (SCAN) + "  "
1030 HBL = BASE - 24: IF  INT (HBL / 128) <  >  INT (BASE / 128) THEN HBL = HBL + 128
1040 DEC = HBL: GOSUB 3000: REM  DO PAGE 1
1050 DEC = HBL + 8192: GOSUB 3000: REM  DO PAGE 2
1060 SYMBOLS$ = "++++++++++++ ": IF SCAN > 223 AND SCAN < 228 THEN SYMBOLS$ = "############ "
1065 LN$ = LN$ + "++++++++++####" + SYMBOLS$
1070 DEC = BASE: GOSUB 3000: REM  DO PAGE 1
1080 DEC = BASE + 8192: GOSUB 3000: REM DO PAGE 2
1090 SYMBOLS$ = "++++++++++++++++++++++++++++++++++++++++"
1100  IF SCAN > 223 AND SCAN < 228 THEN SYMBOLS$ = "########################################"
1110 LN$ = LN$ + SYMBOLS$: PRINT LN$
1120  RETURN
3000  REM
3002  REM  GET HEX & DECIMAL STRINGS
3004  REM
3005 H4 = DEC / 4096:H4% = H4
3010 H3 = (H4 - H4%) * 16:H3% = H3
3020 H2 = (H3 - H3%) * 16:H2% = H2
3030 H1% = (H2 - H2%) * 16 + .5
3040 LN$ = LN$ + "$" + HX$(H4%) + HX$(H3%) + HX$(H2%) + HX$(H1%) + " "
3050 DEC$ =  STR$ (DEC): IF  LEN (DEC$) < 5 THEN LN$ = LN$ + " "
3060 LN$ = LN$ + DEC$ + " "
3070  RETURN
4000  REM
4002  REM  PRINT HEADINGS
4004  REM
4010  PRINT "                 HORIZONTAL BLANKING (HBL)                              HORIZONTAL DISPLAY ENABLE"
4015  PRINT
4020  PRINT "LINE                                          11111111                                        1111111111111111222222222"
4030  PRINT "NUM     PAGE 1      PAGE 2   00123456789ABCDEF01234567    PAGE 1      PAGE 2   0123456789ABCDEF0123456789ABCDEF01234567"
4040  RETURN
5000  REM
5002  REM  PRINT FIGURE NUMBER
5004  REM
5005  PRINT : PRINT
5010  PRINT "FIGURE 5.9 - THIS HIRES SCANNING MAP SHOWS THE ADDRESSES SCANNED DURING DISPLAY PERIODS AND BLANKING PERIODS."
5020  RETURN
```

Figure D.2 BASIC Listing: Program that Produces Figure 5.9.

appendix E

A Logic Circuits Primer

Bits of information in a computer are generally represented by voltages. In positive level logic like that used in the Apple, a high voltage (about 3 volts) is considered to be true, and a low voltage (about 0 volts) is considered to be false. The electronic circuits in the Apple are designed primarily to treat signal voltages as true or false indications and to process them logically. In studying the Apple, it is advisable to concentrate on the logical function of the components rather than their electronic function.

The most basic functional building blocks are simple logic gates. For example, a two input AND gate will bring its output high if and only if both inputs are high. In other words, both input A AND input B must be true if the output is to be true. The two input AND gate is represented in logic diagrams as follows:



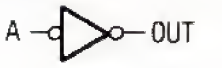
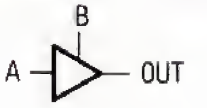
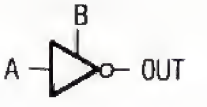
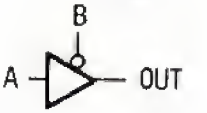
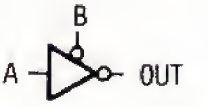
This AND function is identical to the 6502 AND instruction, except that the 6502 instruction is performed on 8 bits simultaneously and is logically equivalent to 8 two input AND gates.

A way of demonstrating a logic function is a truth table. The truth table shows the state of an output for every possible combination of inputs. The true state can be represented by T or 1 or H (for high assuming positive logic), and the false state can be represented by F or 0 or L. We will use H and L because this usually eliminates possible ambiguities. The truth table for the positive logic AND gate is:

INPUT A	INPUT A	OUTPUT
L	L	L
L	H	L
H	L	L
H	H	H

This clearly demonstrates that both inputs of the positive logic AND gate must be high before the output will go high. Table E shows the truth table, schematic representation, and equivalent 6502 instruction, where applicable, of some simple logic gates used in this book.

Table E Basic Logic Gates.

NAME	REPRESENTATION	SECONDARY REPRESENTATION	TRUTH TABLE B A OUT	6502 EQUIVALENT
AND			L L L L H L H L L H H H	AND XXXX
OR			L L L L H H H L H H H H	ORA XXXX
NAND			L L H L H H H L H H H L	AND XXXX EOR #SFF
NOR			L L H L H L H L L H H L	ORA XXXX EOR #SFF
EXCLUSIVE OR			L L L L H H H L H H H L	EOR XXXX
AMPLIFIER			L L H H	NOP
INVERTER			L H H L	EOR #SFF
TRI-STATE AMPLIFIER HIGH ENABLE			L L Z L H Z H L L H H H	
TRI-STATE INVERTER HIGH ENABLE			L L Z L H Z H L H H H L	
TRI-STATE AMPLIFIER LOW ENABLE			L L L L H H H L Z H H Z	
TRI-STATE INVERTER LOW ENABLE			L L H L H L H L Z H H Z	

The little circles on the gates represent the concepts of inversion and active-when-low signals. The two concepts are closely related and sometimes impossible to separate. Inversion is the process of turning a signal into the logically opposite signal. When a signal is low, its inversion is high, and vice versa. When examining a logic gate, the absence of circles can be read as active-when-high and the circles can be read as active-when-low. For example, the NAND gate has a circle on its output, meaning both inputs must go high to make the output go low. This can be stated in a second way. If either input goes low, the output will go high. This results in a second way of representing the NAND gate, as an OR gate with little circles on the inputs. The circles take a little getting used to and are used in some pretty unusual ways in some drawings. Just associate the circle with the word "low" and you should get the message.

The tri-state amplifiers of Table E represent a different sort of logic device. In addition to the normal binary states of high and low voltage, the tri-state device has a third state, high isolation or high impedance. The line coming in from the side is the output enable line and it controls the isolation. When the output enable is not active, the device is isolated from the output line, so another device can control the output line. In electronic terms, the device presents a high impedance to the output line. In the truth tables of Table E, the high impedance state is represented by a "Z". A more detailed discussion of tri-state logic is contained in the chapter on bus structure.

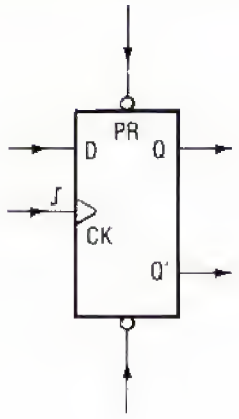
A building block of equal importance to the logic gates is the clocked flip-flop. This is a 1-bit storage device which will respond to its logic inputs when it senses an active transition on its clockpulse input. Figure E.1 shows a diagram of a D type flip-flop and its truth table. The flip-flop shown is clocked by a low-to-high transition of its clockpulse input and is like the 74LS74 flip-flop used several places in the Apple. The Q output will follow the D input every time the clockpulse rises, and the Q′ output will be the inversion of the Q output. The CLEAR and PRESET inputs cause the flip-flop to change states without requiring a clock, and actually override the clocked D input. Bringing PRESET low forces Q high and Q′ low. Bringing CLEAR low forces Q low and Q′ high.

The clockpulse adds synchronization to logic. If the same clockpulse triggers a hundred different actions, then the actions all occur simultaneously. This clockpulse synchronization is common to all digital computers. Certain devices react to the clock. Other devices react to those clocked devices, and so on. After a given period of time, all reactions are complete and the logic signals are all stable, waiting for the next clock. The computer thus operates one cycle at a time.

As an example of clocked operation, Figure E.2 shows a logic function similar to the 6502 AND instruction. In the AND instruction, the value in the accumulator is ANDed with a different value to get the new accumulator value. In Figure E.2, the flip-flop represents one bit of the accumulator. When the flip-flop clock rises, the flip-flop goes to a state determined by its old value ANDed with a second value.

Most logical circuitry is made up of some combination of simple logic gates and flip-flops or their equivalents inside an integrated circuit. In modern computers, many complex functions are available packaged in integrated circuits. Typical of such complex functions are comparison, counting, coding, decoding, and shifting. Even more complex are the functions of chips like the 6502, RAM, and ROM in the Apple. A good way to familiarize yourself with the variety of logic functions available is to peruse the data books published by manufacturers. Of particular help in the Apple is a TTL data book. TTL (Transistor Transistor Logic) is the name of the logic family to which most of the Apple's general purpose chips belong. National Semiconductor is a company which is very good at making their data books available to the public at reasonable prices. Their TTL data is contained in the *Logic Databook*, priced at $9.00 as of March, 1982. Books can be obtained by writing:

National Semiconductor Corporation
ATTN: Literature Distribution MS/14208
2900 Semiconductor Drive
Santa Clara, CA 95051



TRUTH TABLE

INPUTS				OUTPUTS	
PRESET	CLEAR	CLOCK	D	Q	Q'
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H	H
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	HLD	HLD
H	H	H	X	HLD	HLD

TYPICAL OPERATION

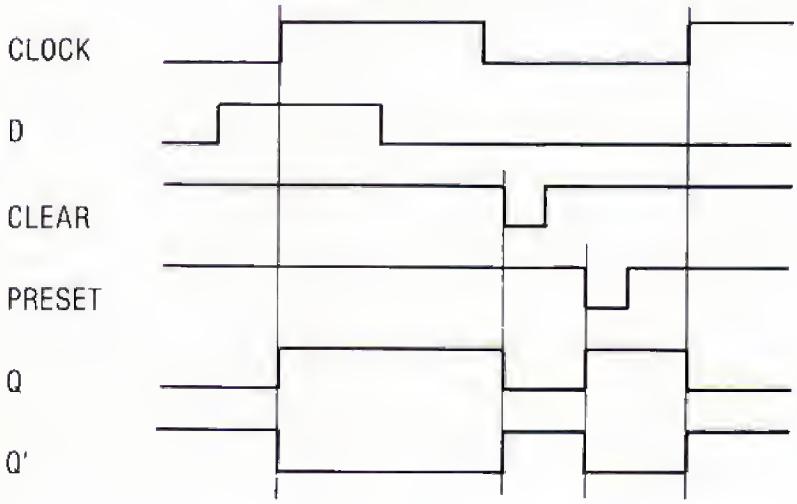


Figure E.1 A D-Type Flip Flop.

Understanding the Apple II uses logic equations to describe the logical makeup of certain signals. A typical logic equation is

VERTSYNC =
VBL • V2 • V1' • V0 • VC' • (H4+H5),

which is read VERTSYNC = VBL AND V2 AND NOT V1 AND V0 AND NOT VC AND (H4 OR H5). The dot represents the AND function, the plus sign represents the OR function, and the prime symbol represents the NOT or INVERSION function.* This selection of symbols makes the equation look like an equation of common algebra, and it's no accidental coincidence. The manipulation of such equations has parallels in the field of algebra and is, in fact, referred to as Boolean algebra, after symbolic logic pioneer, George Boole.

*In representing the NOT function with a prime symbol, this book is following the sensible lead of the *Apple IIe Reference Manual*. The more common convention is to overscore the term or terms to which the NOT function is applied. The overscore is not a particularly workable representation because it is not a common typographical symbol and, more importantly, there is no code for it in standard computer text coding systems such as ASCII. Engineering and manufacturing printouts normally use an asterisk or prime symbol after a term to which the NOT function is to be applied. Apple should be commended for taking the lead in using this notation in published documents. Now, what are they going to do about that dot that represents the AND function?

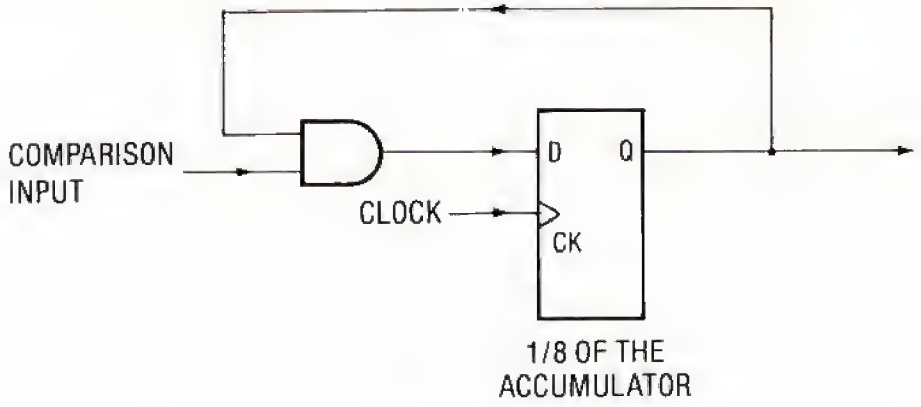


Figure E.2 The Circuit Equivalent of the 6502 AND Instruction.

Other functions besides AND, OR, NOT, and parenthesis grouping can be represented in logic equations, but only these basic functions are represented in logic equations in this book. The purpose in using such equations in *Understanding the Apple II* is only to describe some details of signal generation in a concise way. No algebraic manipulations are described, and none are required on the part of the reader.

appendix F

A Number System Primer

In our daily lives, we represent numerical quantities in the base 10, or decimal, numbering system. For example, by 359 we mean the sum of 9 x (10 EXP 0) plus 5 x (10 EXP 1) plus 3 x (10 EXP 2). This use of the decimal numbering system gives us some unusual biases that would occur only to mathematicians if we used a different base for our numbering systems. For instance, we place special significance on numbers like 1,000,000 (10 EXP 6) but not on 2,985,984 (12 EXP 6, equal to 1,000,000 in the base-12 or duodecimal numbering system). Mathematicians have studied number systems for years, but now, because of the growing influence of computers, knowledge of numbering systems other than base 10 is becoming very common indeed.

You see, the electronics of digital computers is based on hundreds of thousands of two state, or binary, electronic switches which can be on or off. The on or off state of each binary switch can be represented numerically as a ONE or a ZERO, and the information as to whether the switch is on or off is a bit of information. The simultaneous states of eight binary switches can be combined into an 8-bit binary word such as 10011110. Because of the two state nature of digital computer building blocks, digital analysis and design has been performed since day one using the base 2, or binary, numbering system. In this system, there are two digits—1 and 0. The binary number 110 represents the sum of 0 x (2 EXP 0) plus 1 x (2 EXP 1) plus 1 x (2 EXP 2), which is equal to 6 in decimal.

Actual performance of binary arithmetic is very unwieldy, particularly if you consider fractions. Addition and subtraction of 6502 addresses would require 16 digits. For example, subtracting decimal address 35000 from 35003 looks like this:

```
  1000100010111011
- 1000100010111000
------------------
                11
```

If that looks clumsy, you should try multiplying 143 x 247:

```
        11110111
    x   10001111
      ----------
        11110111
       11110111
      11110111
     11110111
    11110111
 11110111000
-----------
1000100111111001
```

To prevent carrying out operations like this, computer programmers use other number systems based on powers of two, such as octal (base 8) and hexadecimal (base 16). Arithmetic is much easier to perform in these systems and conversion to and from binary is so easy that you can do it on sight with a little practice. For example, the above product can be read as hexadecimal 89F9 or octal 104771.

Conversion between binary and octal consists of dividing the binary number into groups of three, from right to left:

1 000 100 111 111 001 = 104771 (base 8)

These patterns of three digits can each be converted to one of eight octal digits from the table below. With exposure, these patterns become very familiar. As you would guess, there are eight symbols in the octal system, 0-7.

The hexadecimal system has 16 digits, 0-9, A, B, C, D, E, and F. The use of letters to represent numbers is sometimes confusing, but that's the convention we're stuck with. When converting between binary and hexadecimal, the binary number is divided into groups of four digits starting from the right:

1000 1001 1111 1001 = 89F9 (base 16).

6502 programming convention calls for use of the hexadecimal numbering system for representing addresses, machine language code, and much data. Convention further calls for preceding hexadecimal numbers with a dollar sign ($89F9) and binary numbers with a percent sign (%10001001), to distinguish them from decimal numbers. Following convention, the Apple monitor represents all numbers in hexadecimal. As a result, some skill in hexadecimal arithmetic and hexadecimal/decimal conversion is very desirable for Apple programmers. In addition, the well rounded computer programmer will be familiar with the binary and octal systems.

Here are two numerical facts of life about 6502 based microcomputers like the Apple. First, there are 16 address lines connected to the 6502. 16 lines can be in 65536 different possible combinations of states (0-65535, $0-$FFFF, or %0-%1111 1111 1111 1111). Second, there are eight data lines connected to the 6502. Eight lines can be in 256 different possi-

Table F Number System Equivalent Representations.

DECIMAL	BINARY	HEXADECIMAL	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

ble combinations of states (0-255, $0-$FF, or %0-%11111111). These numerical features of the 6502 account for some limiting numbers which occur in the BASIC language like 65536, 255, 32767, and 127.

Addresses are normally referred to in hexadecimal in *Understanding the Apple II*. This is because the hexadecimal representations make sense and are easy to remember. Numbers like $10000, $C000, and $2000-$3FFF are much more to the point than 65536, 49152, and 8192-16384 when representing the number of Apple addresses, the start of I/O addressing, and the address range of the HIRES PAGE 1 memory map. It is also far easier to remember that the Cassette output port can be toggled by a reference to any address in the range $C02X ($C020-$C02F) than the range 49184-49199. Dear reader, the effort you spend learning your binary and hex systems will be well worth it.

If this is your first exposure to number systems, then you have just scratched the surface. It is highly recommended that you spend some time with mathematics or computer arithmetic courses, solving problems and familiarizing yourself with this sort of numerical manipulation. You need to get a little used to thinking like a computer. Probably the best exercise possible for you would be to write some 6502 assembly language programs. You'll sink or swim in hexadecimal and binary number systems.

appendix G

Revisional Information

There have been three important revisions to the Apple II motherboard: Revision 1, Revision 7, and the RFI Revision (Radio Frequency Interference reduction). Revision 1 and Revision 7 corrected various minor bugs and added circuits which enhanced the operational features. The RFI Revision was a direct response to a crackdown by the FCC on computer manufacturers whose products emitted RFI. Virtually all personal computer manufacturers were affected by this FCC action.

Between the important revisions, there were a number of inconsequential revisions, some of which were apparently never released. Information about these minor revisions is sketchy because Apple's documentation of the revisions has been sketchy. Revision 1 was documented in the *Apple II Reference Manual*. The next documentation to appear was an addendum to the reference manual which shows the schematic of the RFI Revision Apple II. What happened in each of the interim revisions is a subject of speculation and it is possible that no one person knows what each revision accomplished. Mr. Winston Gayler achieved a bit of a breakthrough in this area by obtaining Apple schematics for the important revisions and getting permission to reproduce them in his book, *The Apple II Circuit Description* (Howard W. Sams & Co. Inc., 1983). *The Apple II Circuit Description* is highly recommended for its schematics and timing diagrams, and because it can provide alternate perspectives of those portions of this book devoted to circuit descriptions of the Apple II motherboard.

The part number of the original Apple II motherboard is 820-0001-00. The "-00" indicates it is Revision 0, and the author suspects it was called Revision 0 only when Revision 1 came out. Apple got up to "-09" before they wore out the old part number, and switched to 820-0044-01 when the RFI revision came out. Then they quickly ran out of numbers and switched over to letters to differentiate between revisions. Do not be bothered if the numbering system seems illogical to you. You probably just do not communicate with yourself on the same level that a big company communicates with itself. Table G.1 is an attempt to categorize the revisional part numbers. Question marks mean that, as far as the author knows, a particular revision may not exist or may not have been released to the public.

Table G Apple II Revision Numbers.

PART NUMBER	REVISION
820-0001-00	REVISION 0
820-0001-01	REVISION 1
820-0001-02	
820-0001-03	
820-0001-04	
820-0001-05 ?	
820-0001-06 ?	
820-0001-07	REVISION 7
820-0001-08 ?	
820-0001-09 ?	
820-0044-01	RFI REVISION
820-0044-A ?	
820-0044-B ?	
820-0044-C	
820-0044-D	

You probably can read the part number of your Apple on your motherboard. There was no part number printed on Revision 0 motherboards. The part number of motherboards prior to the RFI Revision can be seen by removing the 6502 chip or by looking at the bottom of the motherboard. RFI Revision motherboards have the part number printed in plain view near the F1 socket. Some notes about the revisions follow.

Revision 1

Revision 1 was the most important revision from the standpoint of operational enhancements. The addition of delayed HIRES video, the power-up RESET, and the TEXT mode COLOR BURST killer were significant improvements. The changes made in Revision 1 are documented in the *Apple II Reference Manual*. They include:

1. Addition of a COLOR BURST killer circuit to remove colors from text.
2. Addition of power-up RESET circuitry.
3. Addition of the HIRES delayed video feature which gives the blue and orange colors.
4. Solving a memory addressing problem for Apples with 20K or 24K of RAM.
5. Addition of 50 Hz Eurapple jumpers.
6. Connection of COLOR REFERENCE and video SYNC signals to pins 35 and 19 of slot 7.
7. Addition of a wire wrap post which is connected to the VIDEO output signal.
8. Disconnection of the speaker output from the cassette output.
9. Redesign of the cassette input amplifier.
10. Reduction of vertical sync pulse width and addition of horizontal syncing serrations during the vertical sync pulse.

You can tell you have Revision 1 or later by examining the screen text on a color television or monitor. In Revision 0, all screen text is colored violet, green. and white. In Revision 1 or later, TEXT mode text is white, but MIXED MODE text is violet, green, and white.

Revisions 2 through 6

This is a very gray area in which very minimal changes were made. It would be very nearly correct to say all of these revisions are the same as Revision 1. In Revision 4, Apple started soldering 16K RAM configuration plugs into the motherboard instead of sockets. From this point on, the obsolete 4K RAM capability was eliminated. To the author's knowledge, Revisions 5 and 6 were never released.

APPLE II PLUS

The Apple II Plus came out when some revision between 1 and 6 was in production, although it did not represent a revision to the motherboard. The only operational difference between the Apple II and the Apple II Plus is in firmware. The Apple II has Integer BASIC and the old Monitor in ROM, while the Apple II Plus has Applesoft BASIC and the Autostart Monitor in ROM. At approximately the same time as the introduction of the II Plus, the keyboard was changed so CTRL could be required for RESET to function, and the keyboard electronics were moved to a separate card. Numerous versions of the keyboard have been available, but this one major operational difference is associated with the Apple II Plus. Apple may have produced the Apple II and the Apple II Plus concurrently for a while, but they soon switched to the II Plus exclusively. The author's Apple II has a Revision 3 motherboard, so the Apple II was still available when Revision 3 was in production.

Revisions 7 through 9

Revision 7 was the next important revision after Revision 1. Its most significant effect was to change the screen text ROM from a 2513 to a 2316B. This enabled the owner to substitute his own upper/lower case EPROM for the standard ROM. The Revision 7 changes include:

1. Replacement of 2513 screen text ROM with 2316B ROM. The 2316B is fully pin compatible with 2716 EPROM.

2. Removal of RAM configuration sockets and IC E2.
3. Reduction of horizontal sync pulse width from eight to four microseconds. (Transistor Q7 was added as part of this logic.)
4. Change to double pulse horizontal syncing serrations during vertical sync.
5. Addition of jumper pads 7 and 8 and a 4-pin connector (see Figure 7.9).

Revision 7 and later motherboards are easily identified by the absence of RAM configuration plugs at C1, E1, and F1.

Revisions 8 and 9

The author has never seen a Revision 8 or 9 motherboard but he is told they exist. A14 was probably added to the motherboard in one of these revisions for the purpose of improving the effectiveness of COLOR BURST killing in TEXT mode (See Figure 8.7, top right). This removed some distracting colored "ghosts" which occurred on some televisions or monitors in TEXT mode. Other minor changes were probably made.

RFI Revision

After home computers began to proliferate, the FCC became cognizant of the fact that most of them could interfere with nearby radio frequency reception under some circumstances. The FCC consequently revised its regulations to include computing devices. The pertinent restrictions are in part 15 of FCC regulations. Part 15 sets limits on the amount of radio frequency emmission that devices like televisions, garage door openers, and computers can give off. The Apple is a class B computing device, meaning that it is intended for use in residential areas. Restrictions on class B computers are tougher than they are for class A (industrial use) computers, because of the large number of radio frequency receiving devices in residential areas.

In October 1979, the FCC set a deadline of July 1, 1980 for computer manufacturers to bring their products into compliance with part 15. This was extended to January 1, 1981 because the first deadline proved to be unrealistic. Apple requested and was granted a waiver until April 1981. What Apple was doing during this period was finalizing the RFI Revision, which was the next important revision after Revision 7.

Reduction of RFI involved adding shielding to the cabinet and peripheral connecting cables, increasing the width of motherboard printed conductors, addition of filtering circuits, and interchanging the +5 volt and ground distribution so the +5 volt bus is on the top and the ground bus is on the bottom. A steel bar is firmly attached between the motherboard and the base plate in the rear. This bar reinforces the motherboard (Allah be praised) and connects the base plate to ground on the motherboard. These changes represent a substantial engineering effort by Apple, and the extent of the differences is, no doubt, what prompted Apple to change the base part number of the RFI motherboard.

Some minor functional changes were also made in the RFI Revision. These changes included:

1. Changing the double pulse serration in the vertical sync, introduced in Revision 7, back to single pulse serrations.
2. Replacement of two 8T28 ICs with one 8304 as the MPU data bus driver.
3. Elimination of Q7. The function of Q7 is performed by A14 on RFI Revision motherboards.
4. Generation of SOFT 5 with a 1000 ohm pull up resistor instead of two gates on A2 as was done in previous revisions. SOFT 5 is a voltage which represents a constant logical true or "1" in the Apple II.
5. Addition of a high frequency rejection filter at the video output jack.

There have been at least two revisions to the RFI motherboard. These revisions were made solely to reduce RFI emmission and had no operational effect.

appendix H

Historical Notes

The original Apple was designed in late 1975 by Steve Wozniak, a talented, 25 year old college dropout who designed computers for fun. At some point in time, Wozniak entered into a partnership with his friend Steve Jobs, named the machine after a fruit, and sold a few hundred of these Apples. This original Apple had a 6502 microprocessor, 8K of RAM, no motherboard ROM beyond the screen text ROM, a motherboard power supply, and a single slot into which a cassette interface board plugged. The Apple was sold only as a circuit card, but enclosures and keyboards were available.

Of central importance to the first Apple was the 6502 microprocessor, which was then brand new. The 6502 was simple, powerful, and available for $20.00 over the counter to all comers. This accessibility made it an inviting MPU for an independent designer like Wozniak. Steve was a pioneer in building hardware around the 6502 and in programming the 6502. His BASIC interpreter was probably the first BASIC written for the 6502. This program was written directly in machine code, as were the system monitor and Wozniak's other early programs for the Apple.

In fall of 1976, Wozniak completed the design of the Apple II. This new computer far surpassed its predecessor in sophistication with HIRES and LORES graphics capability, 48K of RAM, BASIC and system monitor in ROM, built in cassette I/O, and eight peripheral expansion slots with motherboard decoded slot control signals. The Apple II, no doubt, borrowed many features of hardware and program structure from the 1975 Apple, but most people would not recognize the older computer as an Apple.

While developing his Apple designs, Wozniak was not a lone talent working in solitude at his cerebral pastime. He was a member of the Homebrew Computer Club, the club to end all clubs, from whose membership rolls have come several microcomputer industry leaders. His friends were very interested in Steve's Apple and made substantial contributions to the Apple. Steve gives Allen Baum much of the credit for the peripheral slot structure. In his "Apple II: System Description,"* he mentions Baum for originating the Apple II debug software,

**BYTE Magazine*, May 1977

Doug Kraul for helpful suggestions on I/O structure, and Randy Wiggington and Chris Espinosa for testing Apple BASIC.

The contributions of Steve Jobs to the Apple II were of a different nature. Jobs was not very interested in designing computers, but he was very interested in selling the Apple. It was Jobs who thought big, who thought the Apple could be sold, and who pushed Wozniak in his development of a computer which was getting better and better. It was Jobs who talked big to people who counted: to Rod Holt who came over from Atari to help with electronic engineering tasks such as power supply design, to suppliers who were giving them components at discount prices with 30 days credit, to Mike Markkula who gave the new company business leadership and a quarter of a million green backs in seed money.

The Apple computer company officially came into existence in January of 1977. Company leaders included Markkula, Jobs, Wozniak, Holt, and Mike Scott who came over from National Semiconductor to be company president. Wozniak has a recollection of Scott answering phone calls to Apple while dubbing cassette tapes on a string of tape recorders. The company shipped its first Apple II in June of 1977 and had paid off all its debts by December of the same year. Growth of Apple II sales has increased ever since.

The spectacular success of the Apple II seems to have resulted from a fortunate combination of timing and talented individuals. Appearances suggest that the most critical talents were those of Wozniak, Jobs, and Markkula. At this writing, Jobs and Markkula still hold top management positions with the company. Wozniak has separated himself from Apple's executive processes and does his own thing. Apple attempts gamely to infuse the magic of the Apple II into newer products, while we wait and hope for their eventual success.

appendix I

A Technical Conversation with Steve Wozniak

The first draft of *Understanding the Apple II* was submitted to Quality Software in March 1983. Since this was an investigative work, there were many questions in the author's mind about points of the design of the Apple II: why things were done the way they were and why other things had not been done. Because of luck and the generosity of Steve Wozniak, the author received the answers to a number of his questions from Mr. Woz, himself.

My conversation with Steve occurred on the evening of April 17, 1983, after the Applefest Convention in Anaheim. Although he was tired and wanted only to get home to his family, Steve submitted himself to a ride to the airport in the author's 1972 Nova, a car which sometimes reaches its destination without malfunctioning. During the ride, we talked of nothing but the design of the Apple II computer, which, it turns out, is something Steve loves to talk about and had not seriously talked about in years. I would like to share with the reader a paraphrased synopsis of the comments he made about the design of the Apple II. Please be cautioned that these comments are as reliable as the memory of the author, who does not take notes and drive in LA traffic at the same time. The conversation is definitely not Steve's exact words, or the author's, for that matter.

S. Did Apple design the analog card in the disk drive or did Shugart?

W. Rod Holt designed the analog card. We took a Shugart SA400 drive and designed the interface on our own. Motorola had just come out with a new read interface chip for floppy disks, and we used it in the design. We saw how the head could be positioned through program control. The whole interface was originated at Apple.

S. The DEVICE SELECT' signal to a peripheral slot goes away before PHASE 2 falls on the 6502. Was that a design oversight?

W. I gave consideration to distributing PHASE 2 instead of PHASE 0 throughout the motherboard, but PHASE 0 was much easier to work with. I felt that PHASE 0 probably clocked the data to the 6502, but I no longer think that is

true. We could have changed the timing, but felt that we could control the situation since Apple would be designing all the peripheral cards. We had no idea that many companies would be designing peripherals for the Apple.

S. I think the long bleed off time of the data on the floating data bus will make the DEVICE SELECT' a valid transfer bus management signal in any design.

W. Yes, that's what makes some peripherals work, but some designs have had problems. You must be aware of the situation when designing peripherals for the Apple. There are no other areas where Apple timing doesn't meet component specifications.

S. There are a couple of other areas. One is the text ROM. Apple timing doesn't give the 2513 character generator 450 nanoseconds to put out the text patterns.

W. That doesn't sound right. We were very careful in that area, but I don't remember the details. It may be that Apple uses 350 nanosecond parts. You should call Peter Baum at Apple. He will be very interested.*

S. Is that Allen Baum's brother?

W. Yes. Allen contributed greatly to the Apple. He did most of the work in devising the peripheral slot structure.

S. Did anyone else work on the motherboard design?

W. Except for the peripheral slots, I designed everything. The I/O STROBE' to the slots was one good thing we did with the slots. We just had that signal sitting there and decided to use it for expansion ROM. That made it possible for cards like modems to come with their own driving firmware. That was very important in the cassette based Apple and it gave us a big boost. Do you know that by addition of two chips I could have made Apple screen memory contiguous?

S. You should have done it. Two chips would have been a small price to pay to make the graphics easier to program.

*I did call Peter Baum and he was very interested. He agreed that it sounded like the Apple timing did not meet the 450 nanosecond specification of a standard 2513 or 2316B ROM. Of course, with the Apple IIe in production, a timing abnormality in the old Apple II is not a particularly lively issue in Cupertino.

W. Advanced programming techniques have been developed to overcome that. It takes a little work, but graphics displays can be updated just as fast as if the memory were contiguous. We never dreamed how many people would be writing assembly language programs to manipulate the graphics. Our expectation was that most programs not written by Apple would be written in BASIC.

S. In my book, I call the ROM on the disk controller with the sequencing program a logic state sequencer instead of a state machine. Was it you that called it a state machine?

W. Yes, that's my title. I guess it could be called a sequencer.

S. I went to great lengths to explain the sequencer program in my book. I made up mnemonics for the sequencer commands and printed a program listing.

W. I have a lot of material on that subject which you could have used. There's a person named Gibson who wrote an assembler for the sequencer which actually assists in writing programs for it. He sent me a copy of it.

S. I read that you had a frustrating problem with reflections on conductors of the motherboard. Is that the reason for the resistors connected to the RAM address lines?

W. Yes, that was a big problem.

S. It was a big problem with me. The RAM data sheets don't recommend it and I wasn't sure of their purpose.

W. Exactly. We had to figure out what was going on for ourselves. Rod Holt was trying to analyze it with an oscilloscope, but I experimented with resistors to make the problem go away. Someone did a theoretical analysis later and found that the resistors I'd used were almost perfect.

S. Why didn't RA6 have termination resistors until the RAM configuration plugs were taken out of the Apple?

W. Because RA6 did not have as long a conductor when the configuration plugs were there.

S. On the 16K RAM card, the RAM address inputs are tied to the unused inputs of ICs. Is that to eliminate reflections?

W. I'm not sure, but it sounds right. Something else to consider is that leaving pins open on an IC can cause the IC to draw more power. That may be

the reason for the connections on the RAM card. Incidentally, one difference between the Apple IIe and the older Apples is in the way the RAM card handles resets. The reset did not cause the old RAM card to be automatically disabled; the reset handling program had to disable the RAM card if it was going to get disabled. In the IIe, the reset automatically disables the built-in RAM card.*

S. In my book, I show how you can bank switch motherboard RAM from a peripheral card in the old Apple.

W. I don't think there's much application for it. I think Apple made a mistake in bank switching the auxiliary RAM in the IIe.

S. I like the idea of having two applications resident in RAM simultaneously with instantaneous switching between them. It's like having Apple A and Apple B with selection between them.

W. That sounds pretty good. One problem that I have with the auxiliary RAM slot of the IIe is that it won't easily support 256K RAM chips. You run into problems because only the multiplexed RAM address bus for 64K RAM chips is available at the auxiliary slot.

S. I'd never thought of that. Large auxiliary bank switched memory units might be easier to design for the peripheral slots than the auxiliary slot.

W. I'll tell you about a timing problem that will interest you. My first design for the Apple II used a display method in which the 6502 was stopped for 40 microseconds. The Synertek data sheet said you could stop it for 40 microseconds, but I was having problems. The 6502s would work for a while, but the Apple would eventually stop working. I always had to have a new 6502 in my pocket in case it happened.

S. You mean the 6502 operated for 25 cycles, then the video display operated for 40 cycles?

W. That's right. When I designed the Apple II, dynamic RAM was just becoming available that could be accessed at two Megahertz. When it did become available, I changed the design of the Apple II to take advantage of it. I've told Synertek about the problem but they haven't changed their specification. New 6502s can be stopped for 40 microseconds, but they deteriorate. They are dynamic devices that store data in internal capacitive elements. As the 6502 wears in, its capacitive elements become less efficient.

S. I'm very glad you brought this up. Descriptions in my book are very vague about how long you could stop the 6502 during DMA. The data sheets are inconsistent. Synertek says 40 microseconds, Rockwell says 10 microseconds, and MOS Technology doesn't say at all.

W. Rockwell changed theirs to 10 microseconds? That's great.

S. Do you think 10 microseconds is a good number?

W. I'd stick with five or six. That's what Microsoft uses on their Z80 Softcard. It refreshes the 6502 every few cycles to keep it active. The 6502 is still used for things like disk I/O, even when the Z80 card is activated.

S. The R/W′ line on the Apple II does not disable ROM. If a program writes to a ROM address, the ROM fights with the 8T28 bus driver for control of the data bus.

W. Yes, that's right. That probably could have been done differently.

S. You could have connected R/W′ to the third chip select input to ROM.

W. That's an interesting idea, but there may be reasons why it won't work. I'd have to study the timing and schematics.

S. You said in your speech that Mike Scott assembled the old red reference manual and Chris Espinosa wrote the more recent one.

W. Yes. Chris started working with us when he was still in high school. He got in trouble at home for staying up late working with us. He did a terrific job on the reference manual and he was just a college student. I was very glad to see Apple list his name as the author inside the cover. He deserves a lot of credit.

S. Programming is a very slow process for me. I'm amazed at your ability to effortlessly knock out a program like Integer BASIC.

W. You've got that entirely wrong. It takes me forever to write involved programs, too. I just work very long hours for days on end when I'm on a project. Writing Integer BASIC was not easy for me.

**The Apple IIe has an area of RAM, referred to by Apple as the bank switched RAM, which behaves almost identically to the 16K RAM card in older Apples. Another difference, besides the one Steve pointed out, is that the old RAM card did not react to the INHIBIT′ line of the peripheral slots. Pulling the INHIBIT′ line low in the IIe inhibits the bank switched RAM, giving firmware cards and the like priority over the bank switched RAM.

S. It's too bad your BASIC doesn't do floating point arithmetic.

W. I've often regretted not changing it to floating point. The problem was that I had reached the limits of the program structure and it would have been difficult to expand it without a major rewrite.

S. In your speech, you talked about Randy working with you on the disk controller. Is that Randy Wiggington?

W. Yes, he made considerable contributions.

S. Who first wrote the DOS?

W. I wrote the parts that communicated directly with the controller, and Randy wrote the rest.

S. You came up with those phase control tables for stepping the head.

W. That's right, I did all of that stuff.

Arrival at the PSA terminal ends our conversation. Have a nice flight, Steve, and take care.

appendix J

Baseplate and Motherboard Removal

Separation of Baseplate from White Case

1. Unplug the Apple and remove all peripheral cards. Disconnect the video cable and any extraneous wires.
2. Move the Apple to a convenient work area. Place it face down on a soft surface which won't mar the case.
3. Remove the ten screws pictured in Figure J.
4. Place one hand underneath the Apple and one hand on top. Carefully turn the computer right side up while holding the baseplate against the case.
5. Remove the keyboard plug from its socket on the motherboard and separate the baseplate from the white case.
6. Reassembly is the reverse of disassembly.

Removal of Motherboard from Baseplate

1. Disconnect the speaker and power supply plugs from the motherboard.
2. In older Apples, there is a nut on a screw protruding through the center of the motherboard. Remove the nut using a nutdriver or socket and driver.
3. On newer Apples, there is a metal reinforcing bar in the back between the motherboard and the baseplate. It is held in by two or four screws. Remove the retaining screws and slide the reinforcing bar out.
4. There are a number of white plastic retainers protruding through the motherboard. Pinch the clips on each retainer together in turn and gradually work the motherboard off.
5. Reassembly is the reverse of disassembly.

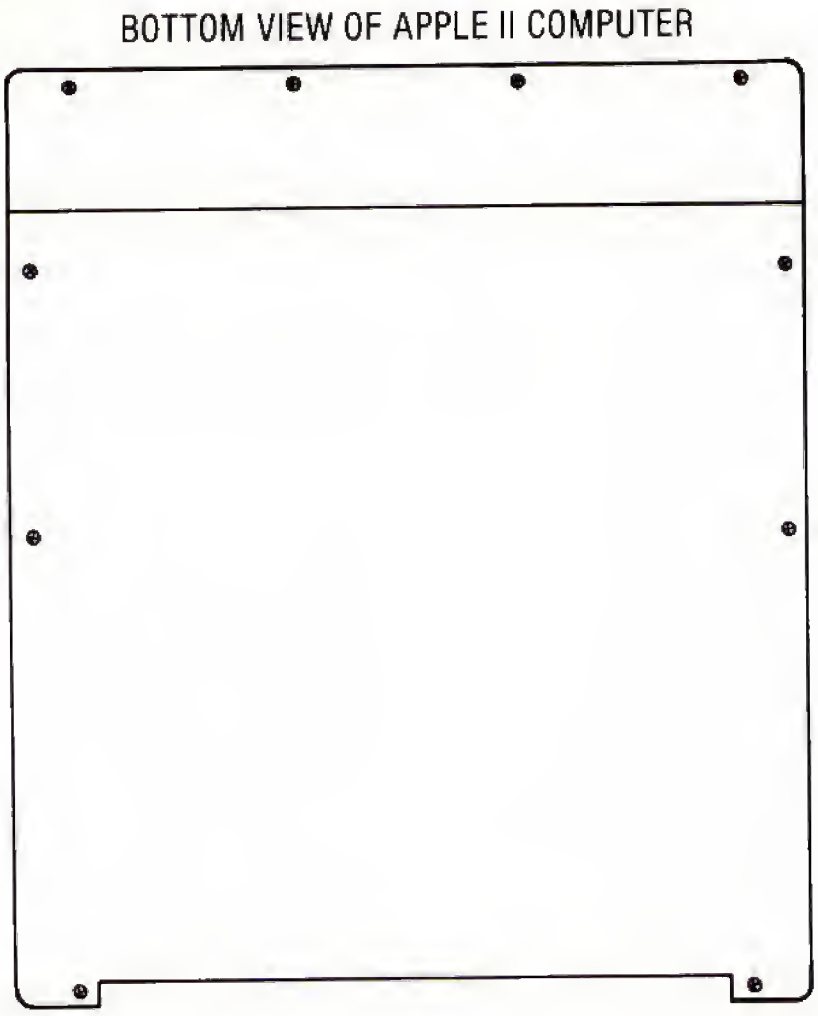


Figure J Remove the 10 Screws Pictured to Separate the Baseplate from the White Case.

appendix K

List of Figures and Tables

Figure 1.1 TEXT and LORES Graphics.
Figure 1.2 How the Game I/O Plug Should Be Designed.
Figure 2.1 A Hypothetical Four-Line Bus.
Figure 2.2 Basic Microcomputer Building Blocks.
Figure 2.3 Communication on the Bus System.
Figure 2.4 The Apple II Bus Structure Showing Disk, Cassette, and Serial I/O.
Figure 2.5 Secondary Address and Data Connections Extend the Address Bus and Data Bus.
Figure 3.1 Functional Flow: The Timing Generator and the Video Scanner.
Figure 3.2 Idealized Timing Diagram for the Timing Generator.
Figure 3.3 Timing Diagram for the Timing Generator, Showing Propagation Delay.
Figure 3.4 Propagation Delay Hierarchy.
Figure 3.5 Distribution of Timing Generator Outputs.
Figure 3.6 The 6502 Machine Cycle Slightly Lags the PHASE 0 Clockpulse.
Figure 3.7 Exaggerated View of a Television Scan. The Apple Scans 262 Times Horizontally for Each Vertical Scan.
Figure 3.8* Schematic Diagram: The Timing Generator.
Figure 3.9* Schematic Diagram: The Video Scanner.
Figure 3.10a Assembler Listing: A Screen Splitting Program.
Figure 3.10b BASIC Listing: A Split Screen Example.
Figure 3.11 Circuit to Generate a 6502 Interrupt 416 Cycles Before the Start of the Screen Display.
Figure 3.12 Circuit to Separate a Vertical Sync Pulse from the Television Sync Signal at Pin 19 of Slot 7.
Figure 3.13 Circuit that Duplicates the Vertical Section of the Video Scanner.
Figure 3.14 Assembler Listing: Syncronizing the Video Scan Simulator.
Figure 3.15 A Programmable Interrupter. It Generates an Interrupt Before Any One of 256 Selected Horizontal Lines.
Figure 3.16 A Programmable Video Interrupter Card.
Figure 4.1 6502 Pin Assignments.
Figure 4.2* Schematic: 6502 Connections in the Apple II.
Figure 4.3 6502 Clockpulse Relationships.
Figure 4.4 Some Worst Case 6502 Specifications.
Figure 4.5 Experimental 6502 Timing Relationships.
Figure 4.6 Reading the Disk Input Port Using Device Select'.
Figure 4.7 Cycle Stealing DMA.
Figure 4.8 Schematic: D MAnual Controller.
Figure 4.9 A Screen Mode Controller.
Figure 4.10 Schematic: An NMI Based Single Stepper.
Figure 4.11 The Author's Front Panel.
Figure 4.12 Assembler Listing: NMI Stepper Routines.
Figure 4.13 Design Concept for a Hardware Breakpoint Generator.
Figure 5.1 RAM Timing Signals from the Timing Generator.
Figure 5.2* Schematic: Apple II RAM.
Figure 5.3 The Functions of the Address Multiplexer.
Figure 5.4 128-Byte Video Memory Segments Consist of Three 40-Byte Sections, Each Mapped Into a Different Part of the Video Screen.
Figure 5.5 TEXT/LORES Displayed Memory Map.
Figure 5.6 TEXT/LORES Video Scanning Map.
Figure 5.7 HIRES Memory Areas (Page 1).
Figure 5.8 HIRES Displayed Memory Map.
Figure 5.9 HIRES Video Scanning Map.
Figure 5.10 Schematic: The Address Multiplexor.
Figure 5.11 CAS' Signal Example.
Figure 5.12 Timing Example: A 6502 Read Cycle to Address $1000.
Figure 5.13 Timing Example: A 6502 Write Cycle to Address $1000.
Figure 5.14 Timing Example: A 6502 Read Cycle to Address $C010.
Figure 5.15* Schematic: The 16K RAM Card.
Figure 5.16 Schematic: Bank Switching Motherboard RAM.
Figure 5.17 Screen Memory Scanning.
Figure 5.18 Integer BASIC Listing: Underline Program.
Figure 5.19 Assembler Listing: Underline Program.
Figure 6.1* Schematic: ROM in the Apple II.
Figure 6.2 Timing Example: ROM Read, Address $F000.
Figure 6.3* Schematic: The Firmware Card.
Figure 6.4 Timing Example: A Read From Address $F000, Firmward Card Enabled.
Figure 6.5 Construction of a Socket Adaptor, EPROM to ROM.
Figure 6.6a An Added Switch to Control the F8 ROM on the Firmware Card.
Figure 6.6b This Circuit Syncronizes Switching to Prevent Possible Program Crashing.

Figure 6.7 Circuit to Allow Independent Selection of the F8 ROM.
Figure 6.8 Method for Isolating the F8 Chip Selects from Other ROM Chip Selects.
Figure 7.1 Schematic: Address Decode.
Figure 7.2* Schematic: Generation of Address Decoded Signals.
Figure 7.3 Address Decoding in the Apple is an Exercise in Division by Eight.
Figure 7.4 Timing Example: 6502 Access to $C080.
Figure 7.5* Schematic: Serial I/O Devices.
Figure 7.6 Cassette Input Wave Shaping.
Figure 7.7* Schematic: The Apple II Keyboard.
Figure 7.8* Schematic: The Apple II Plus Keyboard and Encoder Board.
Figure 7.9* Peripheral Slot Connections.
Figure 7.10 Some Revision-7 Additions.
Figure 7.11 Assembler Listing: A Paddle Read Program.
Figure 7.12 A Modified Game I/O Plug.
Figure 7.13 Wiring a Paddle Set Plug.
Figure 7.14 This Game I/O Extender Can Support Two Sets of Paddles and Two Joysticks Simultaneously.
Figure 7.15 Game I/O Extender Configurations.
Figure 7.16 Schematic: Game I/O Extender.
Figure 7.17 The CTRL-RESET Modification, for Older Keyboards.
Figure 7.18 Installation of SHIFT Key Mod, for Older Keyboards.
Figure 7.19 Enabling Lower Case, Apple II Plus.
Figure 7.20 Two Volume Control Methods.
Figure 8.1 The MPU, Video Scanner, RAM, and Video Generator All Play a Part in Creating the Video Display.
Figure 8.2 The Apple II Video Signal.
Figure 8.3 Video Generator Functional Flow Diagram.
Figure 8.4 Apple Text Patterns.
Figure 8.5 Blanking During Television Scanning Causes the Black Margin Around the Apple Display.
Figure 8.6* Schematic: Video Generation.
Figure 8.7* Schematic: Revision-7 and RFI Version Changes to Video Generation.
Figure 8.8 Video Output Examples.
Figure 8.9 The Output of the Screen Splitting Program (Figure 3.10).
Figure 8.10 HIRES Interference at Pattern Borders.
Figure 8.11 LORES Patterns at B10, Pin 5.
Figure 8.12 LORES Colors.
Figure 8.13 Switching from GRAPHICS to TEXT in MIXED Mode.
Figure 9.1 Functional Diagram of the Disk Interface.
Figure 9.2 Data Transfer in Disk I/O.
Figure 9.3 Simplified Functional Diagram of Disk II Drive.
Figure 9.4 A Stepping Motor.
Figure 9.5 The Write Field is the Vector Sum of the WRITE and ERASE Fields.
Figure 9.6 When Reading, the Lack of a Read Pulse at a Regular Interval Represents a ZERO.
Figure 9.7 Block Diagram of the Disk II Controller.
Figure 9.8* Schematic: The Disk II Controller (Address References Assume Slot 6).
Figure 9.9 BASIC Listing: Program to List State Sequencer ROM.
Figure 9.10 The DOS 3.2 Logic State Sequencer.
Figure 9.11 The DOS 3.3 Logic State Sequencer.
Figure 9.12 Timing Example: Sequencer Control While Changing the READ/WRITE or SHIFT/LOAD Switches.
Figure 9.13 Timing Example: Switching to Write After Checking Write Protect.
Figure 9.14 Flowchart of the Write Sequence.
Figure 9.15 Diskette Formatting.
Figure 9.16 Simplified Flowchart of the Read Sequence.
Figure 9.17 Decision Points for Reading ZEROS.
Figure 9.18 Read Performance of the Logic State Sequencer.
Figure 9.19 Jim Aalto's Finite State Automaton Diagram.
Figure 9.20 Flowchart of RWTS Routine.
Figure 9.21 A Single Switch on the Front of the Drive Allows the User to Select Normal, Forced Write Protect, or Bypass Write Protect.
Figure 9.22 A Drive With the Write Protect Switch Installed.
Figure 10.1 Temperature Measurements Inside an Apple II Plus.
Figure 10.2 A Power Supply With the Bottom Off.
Figure 10.3 Some People Just Shouldn't Handle ICs.
Figure D.1 BASIC Listing: Program that Produces Figure 5.8.
Figure D.2 BASIC Listing: Program that Produces Figure 5.9.
Figure E.1 A D-Type Flip Flop.
Figure E.2 The Circuit Equivalent of the 6502 AND Instruction.
Figure J Remove the 10 Screws Pictured to Separate the Baseplate from the White Case.
Table 3.1 Durations and Frequencies of Timing Signals.
Table 4.1 6502 Instructions.
Table 4.2 6502 Instruction Cross Reference.
Table 4.3 Operation of Soft Switches from D MAnual Controller.
Table 4.4 Selection of 16K RAM Card or IIe Bank Switched RAM from D MAnual Controller.
Table 5.1 MPU/Scanner Equivalent Address Bits.
Table 5.2 Screen Memory Usage Summary.
Table 5.3 The Video Scanner Row Address Assignments.
Table 5.4 16K Ram Card Address Bus Commands.
Table 5.5 16K Dynamic RAM Chips.
Table 7.1 Address Decoded Signals.
Table 7.2 Relation of ASCII to Keypress.
Table 8.1 Eurapple/NTSC Differences.
Table 8.2 Size/Distance Ratios on the Apple Screen.
Table 8.3 The Division of Screen Text Patterns in a 2048 x 8 ROM.
Table 9.1 Disk II Controller Commands.
Table 9.2 Functions of the $C08C,X/$C08D,X and $C08E/$C08F Switch.
Table 9.3 Logic State Sequencer Command.
Table 9.4 Syncing the READ Sequence to Data.
Table 9.5 Data Valid Periods in Sequence Clocks.
Table C.1 6502 Timing Comparisons.
Table C.2 6502 Instruction Execution Periods in Machine Cycles.
Table E Basic Logic Gates.
Table F Number System Equivalent Representations.
Table G Apple II Revision Numbers.

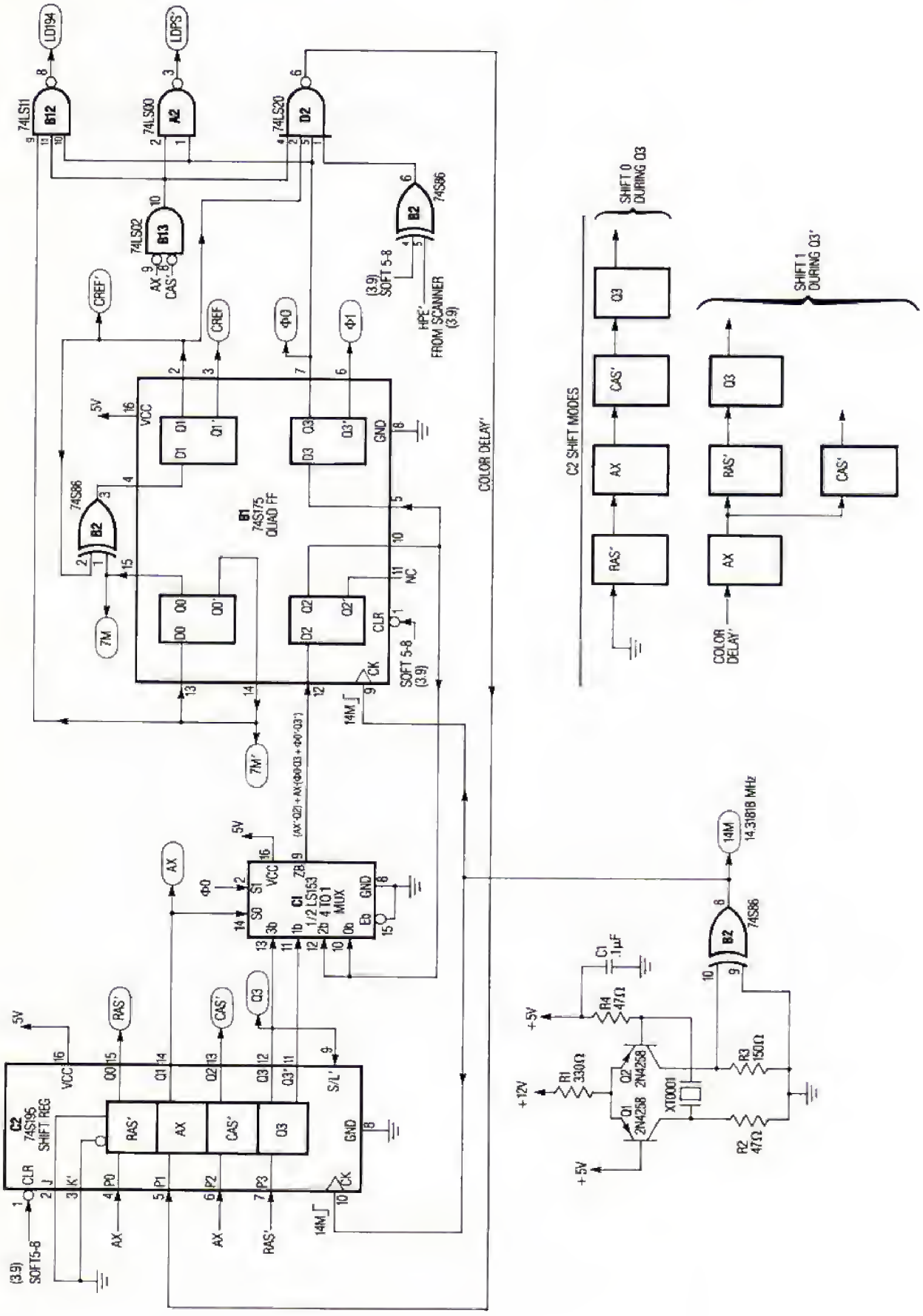
*This schematic can be found in the Schematic Diagram section, as well as in the text.

Schematic Diagrams

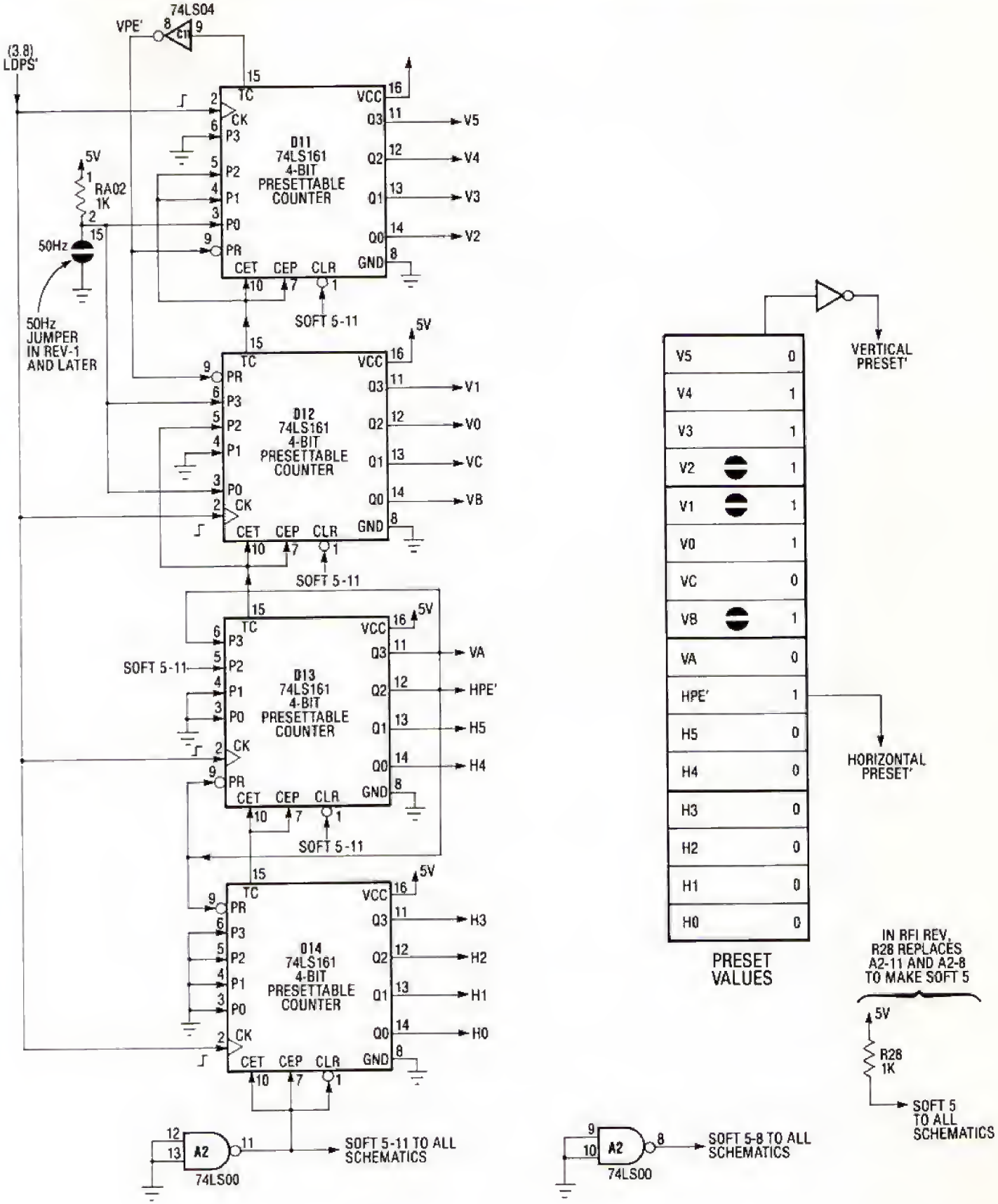
For ease of reference, most of the schematic diagrams which appear in the various chapters of the book are grouped together here. This includes the schematics of the hardware manufactured by Apple Computer, Inc. The schematics in this section are identified in Appendix K with an asterisk next to the figure number.

Numbers in parentheses next to signal terms in the schematics are figure references showing the number of the figure where a signal is generated or to which a signal is sent. Small circles with numbers in them on peripheral card schematics represent pins on the card's edge connector.

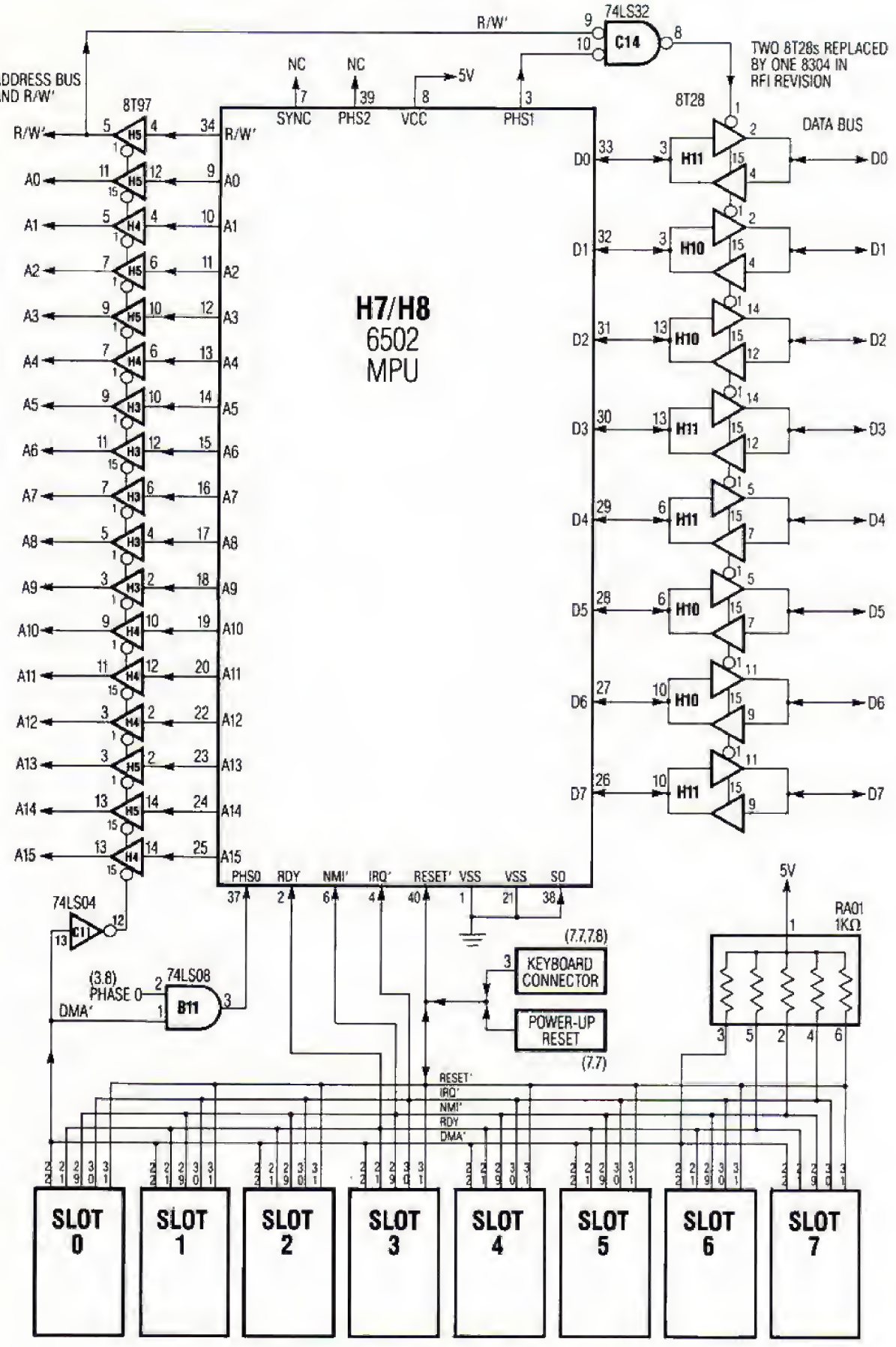
The schematics contained in *Understanding the Apple II* are layouts created by the author in an attempt to make the information contained there more understandable to the reader. The primary sources of detail for these drawings were the schematics contained in the *Apple II Reference Manual*, and the *The DOS Manual*. Schematics of the firmware card, the 16K RAM card, the Apple II Plus keyboard, the read/write head assembly, and some undocumented Revision 7 changes are the result of investigations of the actual hardware by the author. Some changes were made to the schematics to reflect revisional information which is contained in *The Apple II Circuit Description* by Winston Gayler, (Howard W. Sams & Co., Inc. 1983).



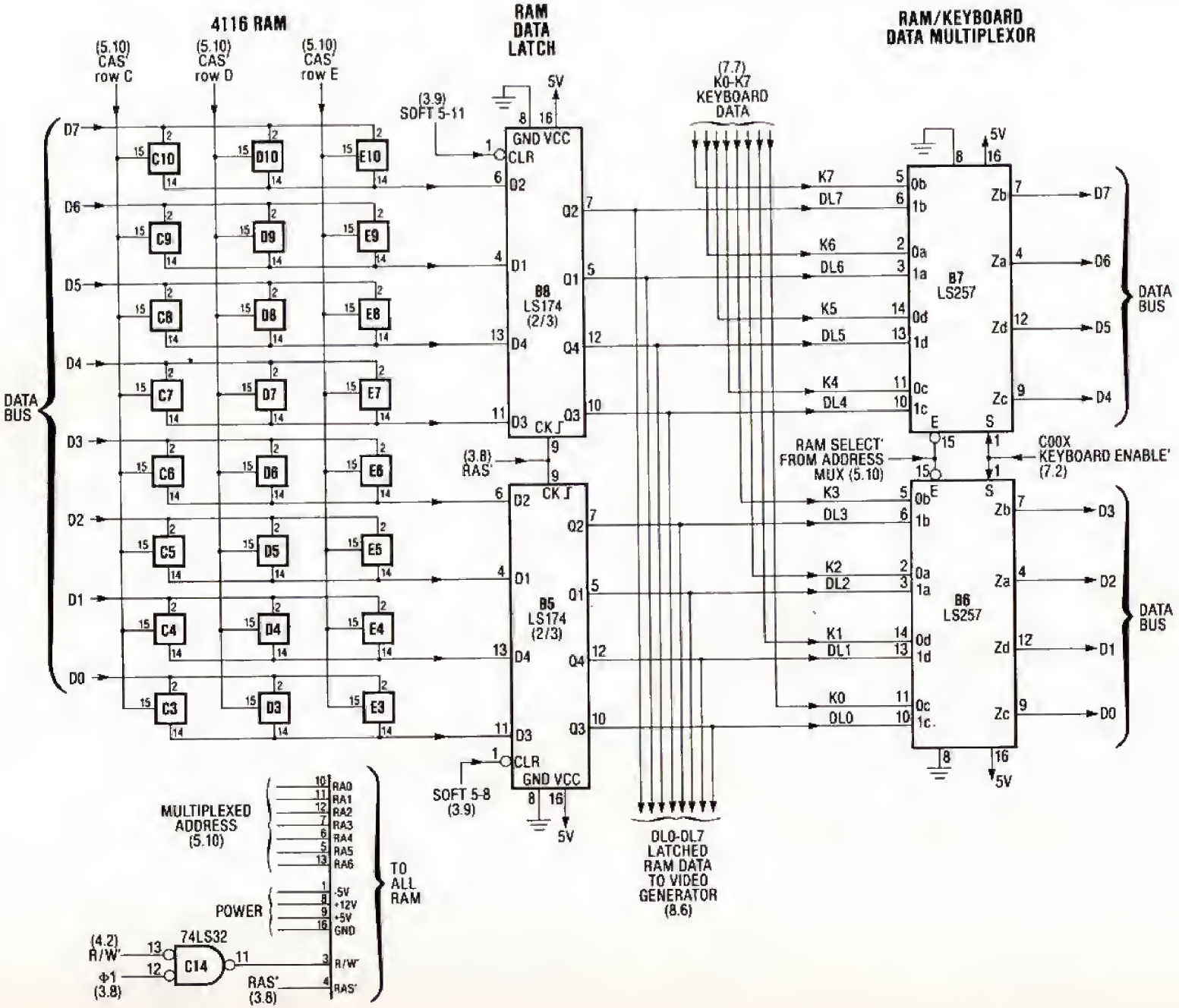
The Timing Generator (Figure 3.8)

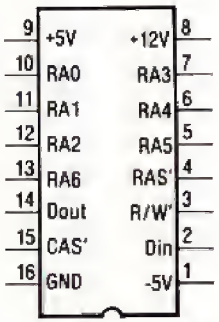


The Video Scanner (Figure 3.9)



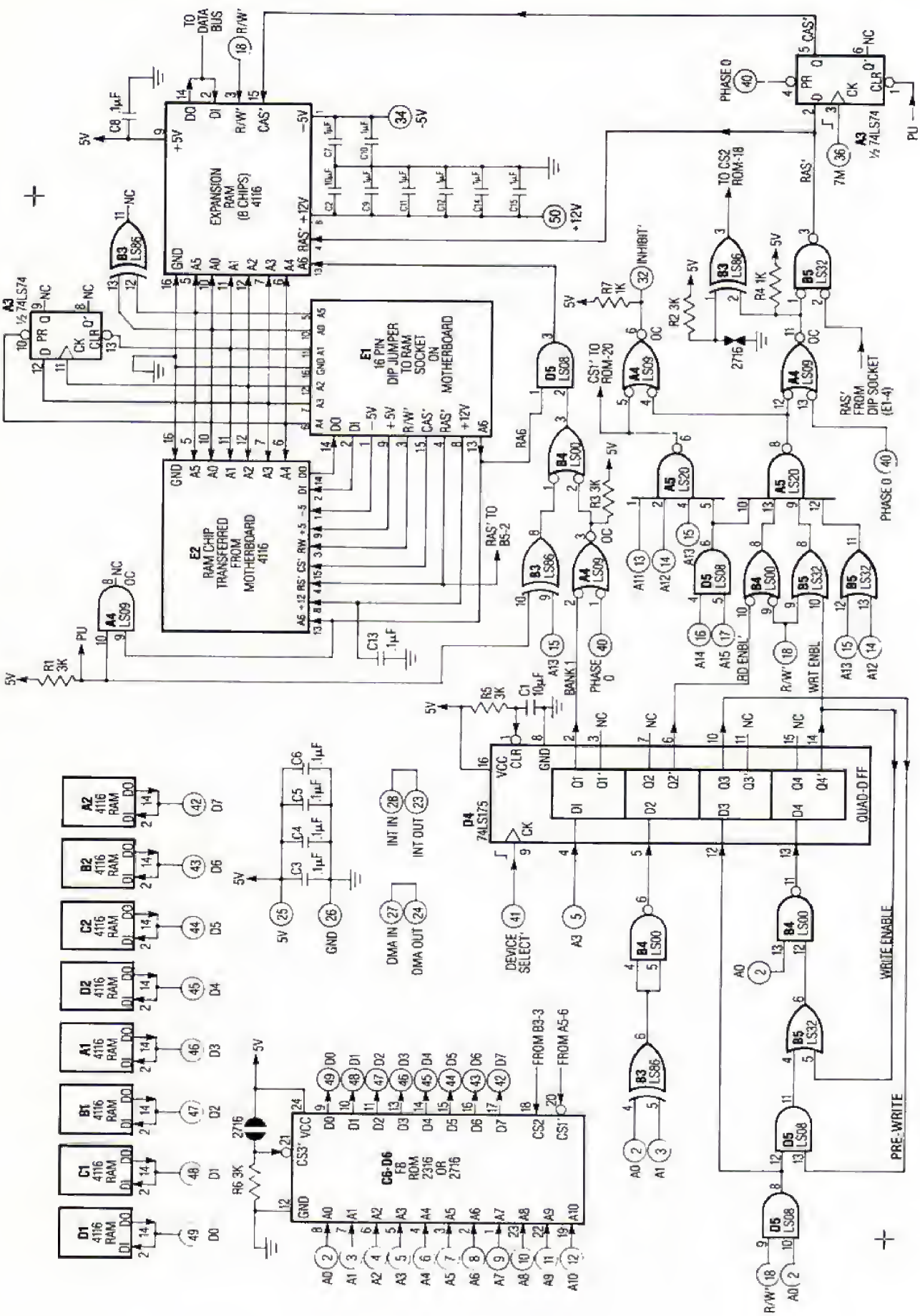
6502 Connections in the Apple II (Figure 4.2)



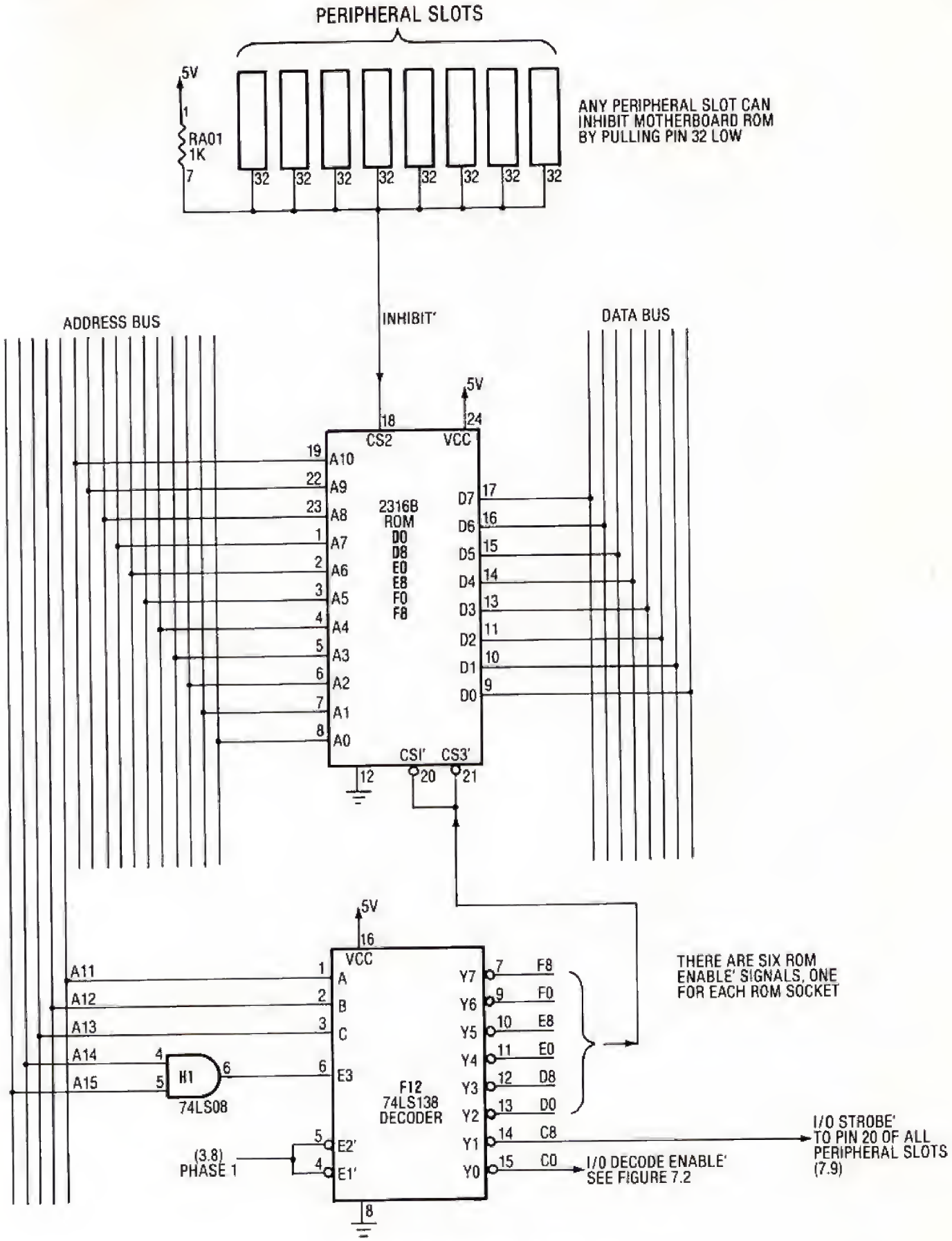


16K DYNAMIC RAM PIN ASSIGNMENTS

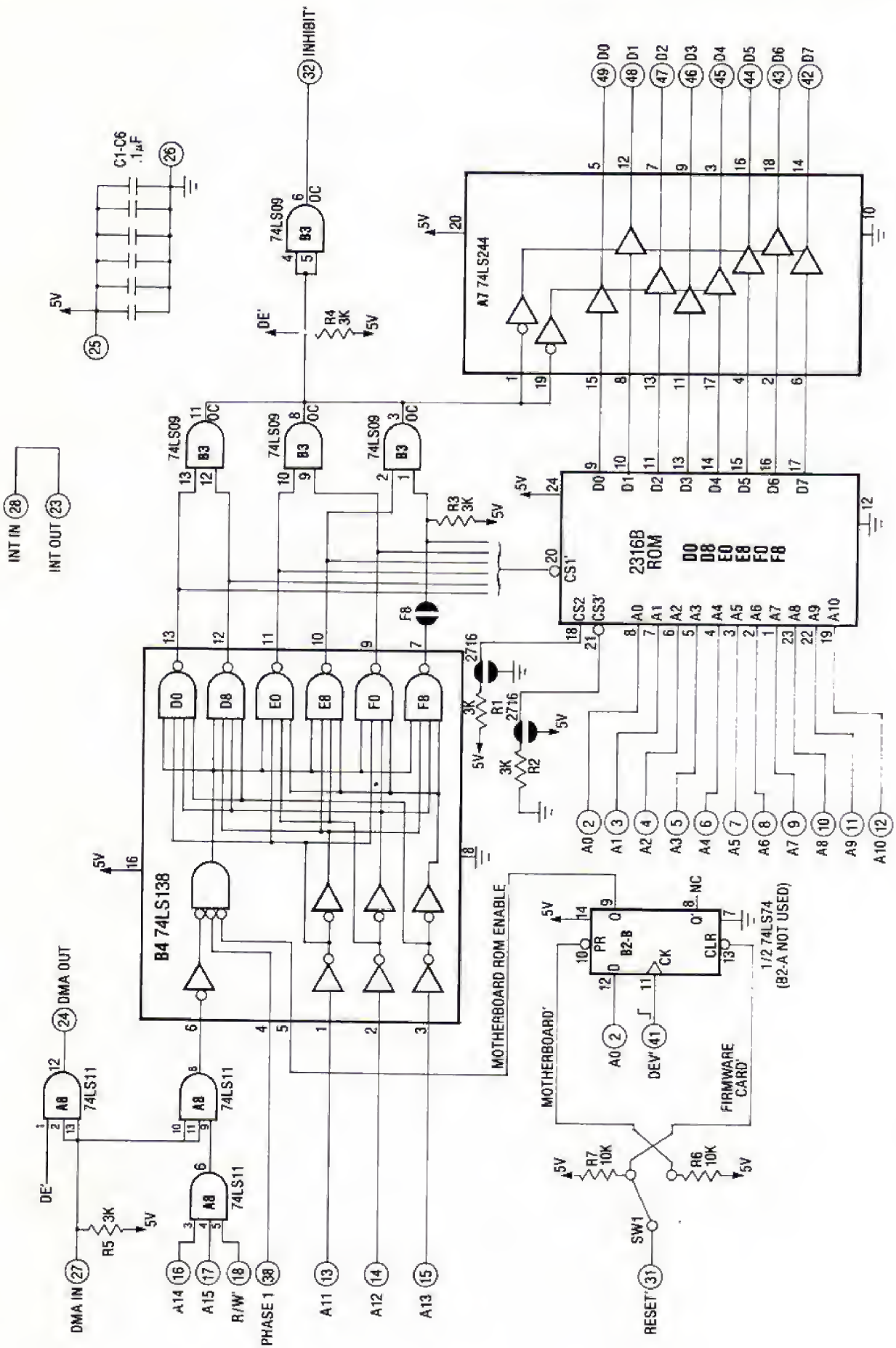
Apple II RAM (Figure 5.2)



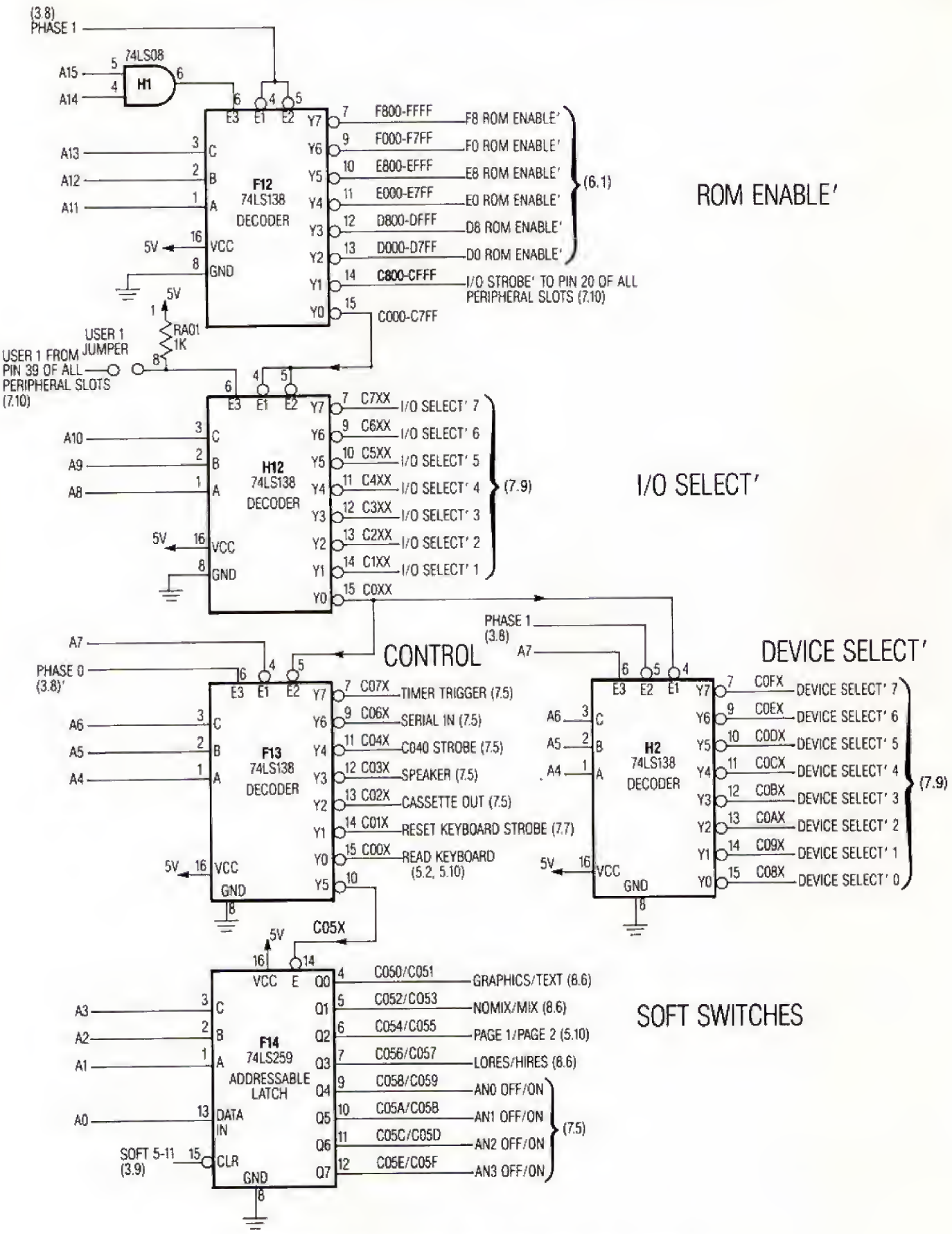
The 16K RAM Card (Figure 5.15)



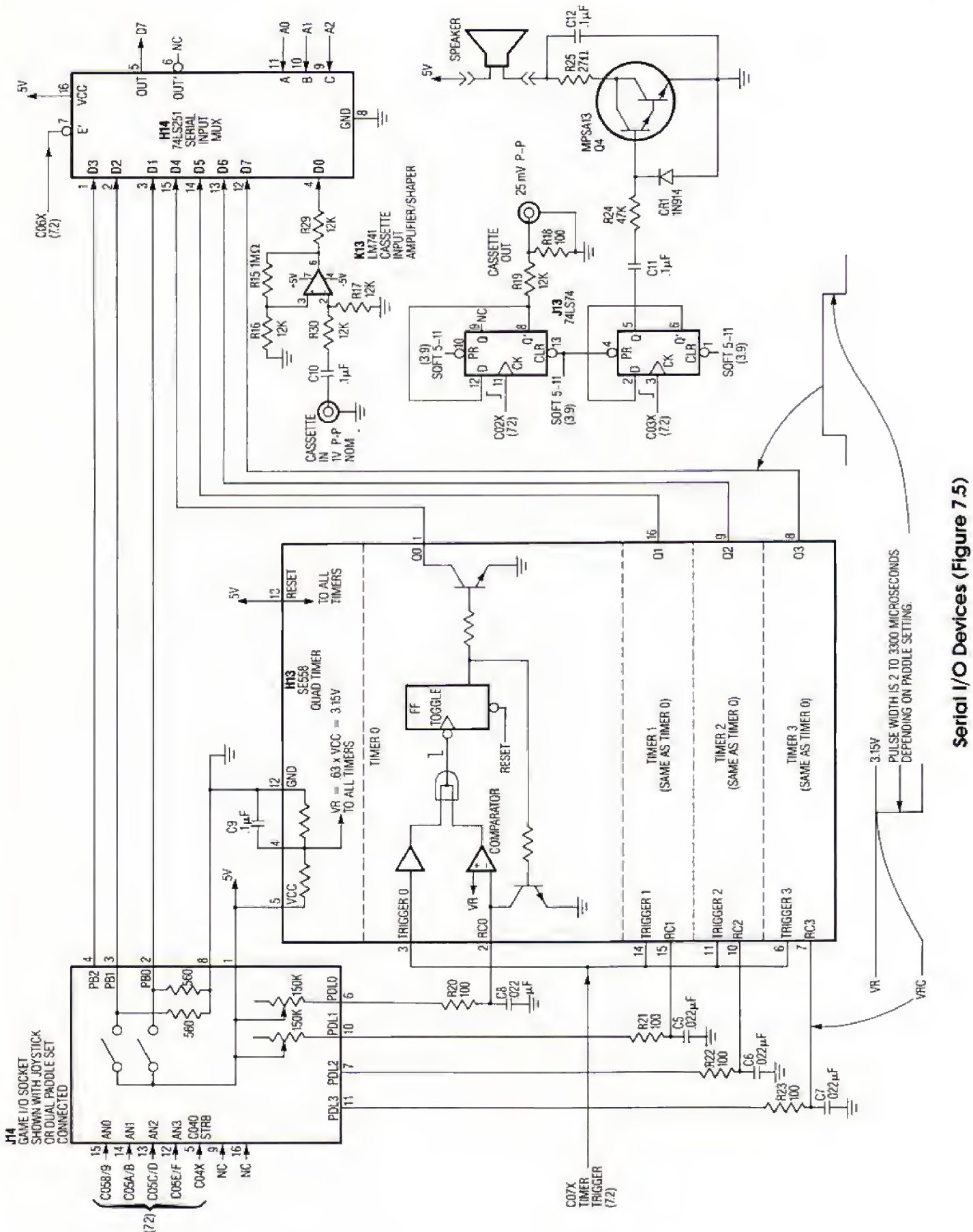
ROM in the Apple II (Figure 6.1)



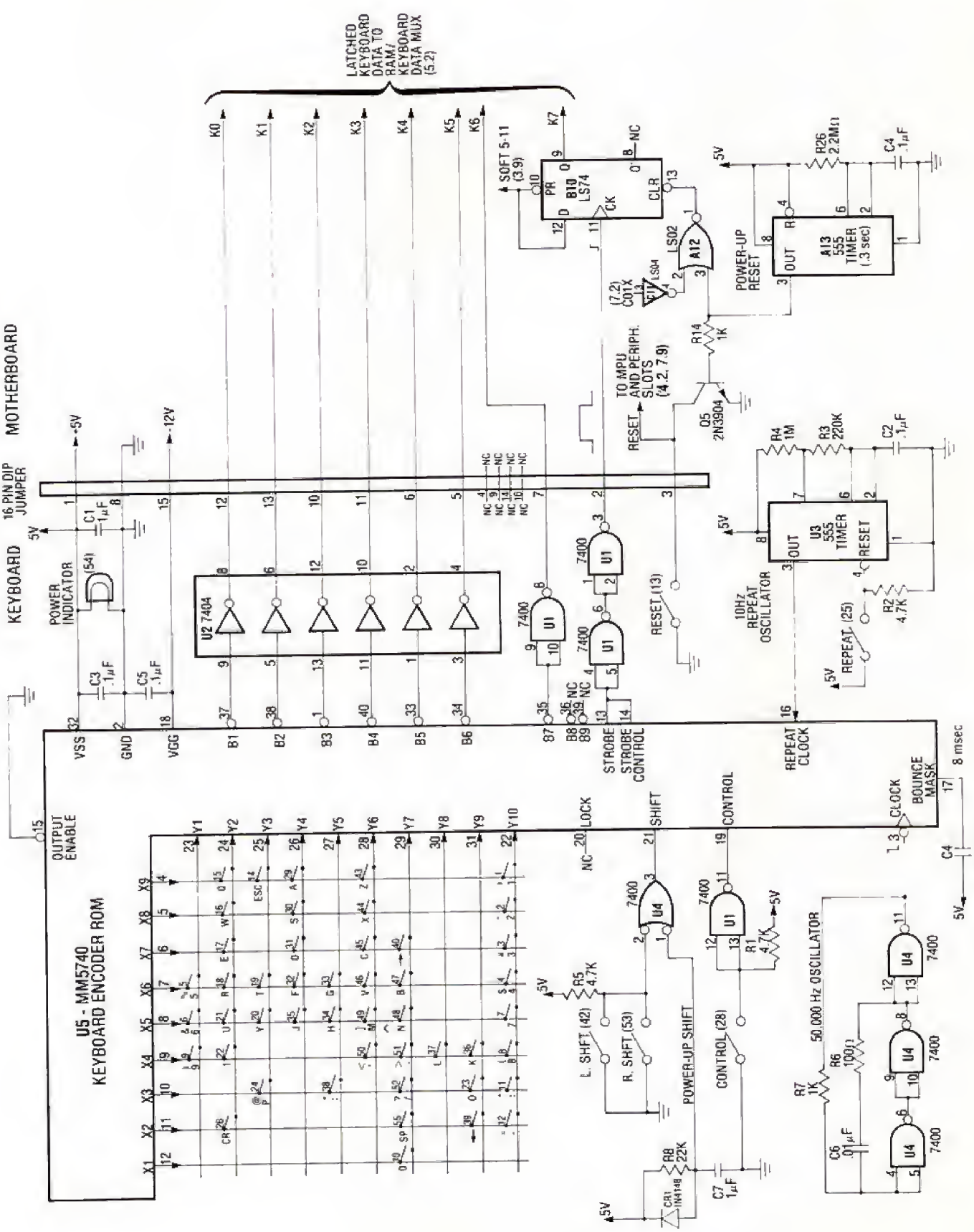
The Firmware Card (Figure 6.3)



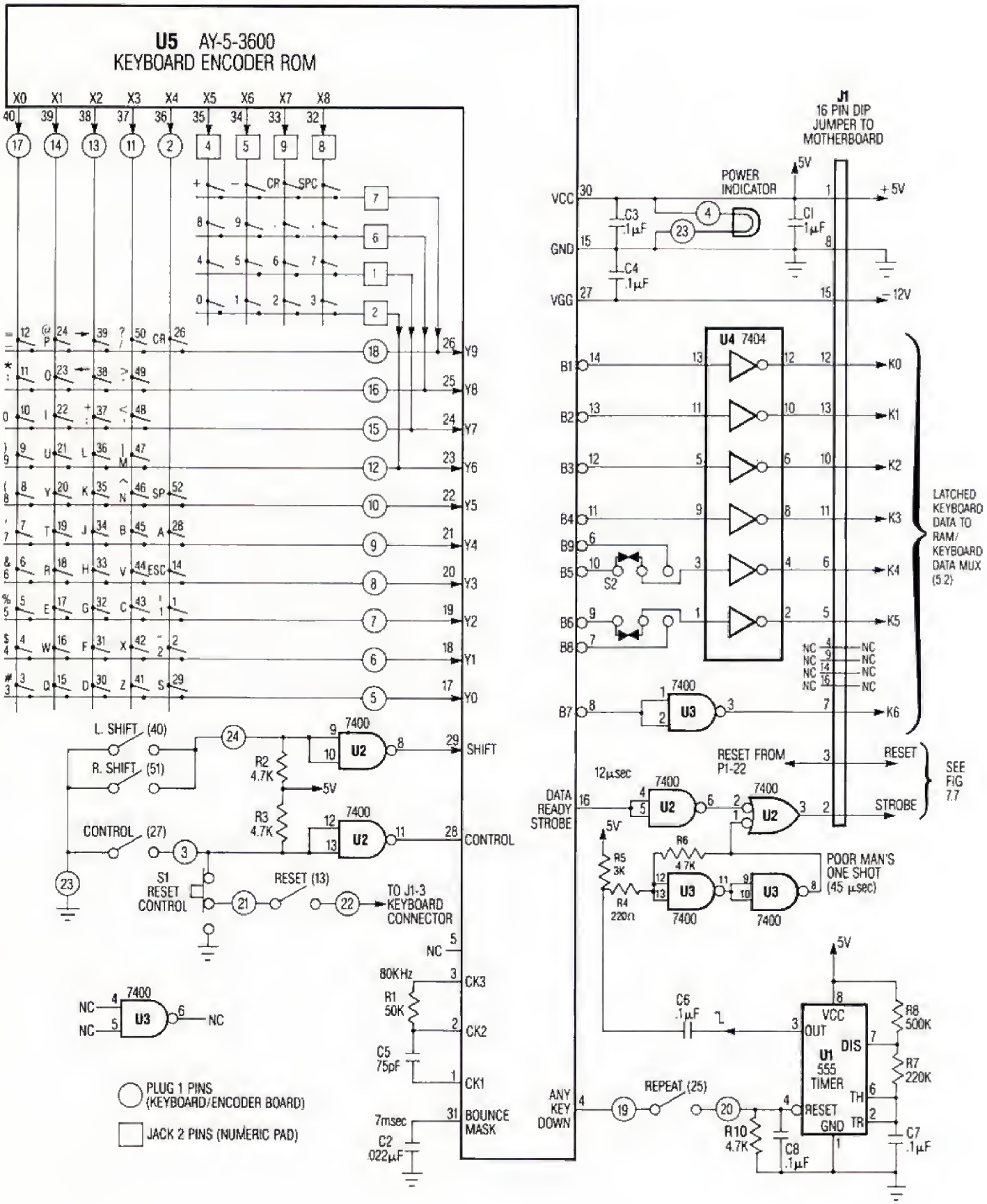
Generation of Address Decoded Signals (Figure 7.2)



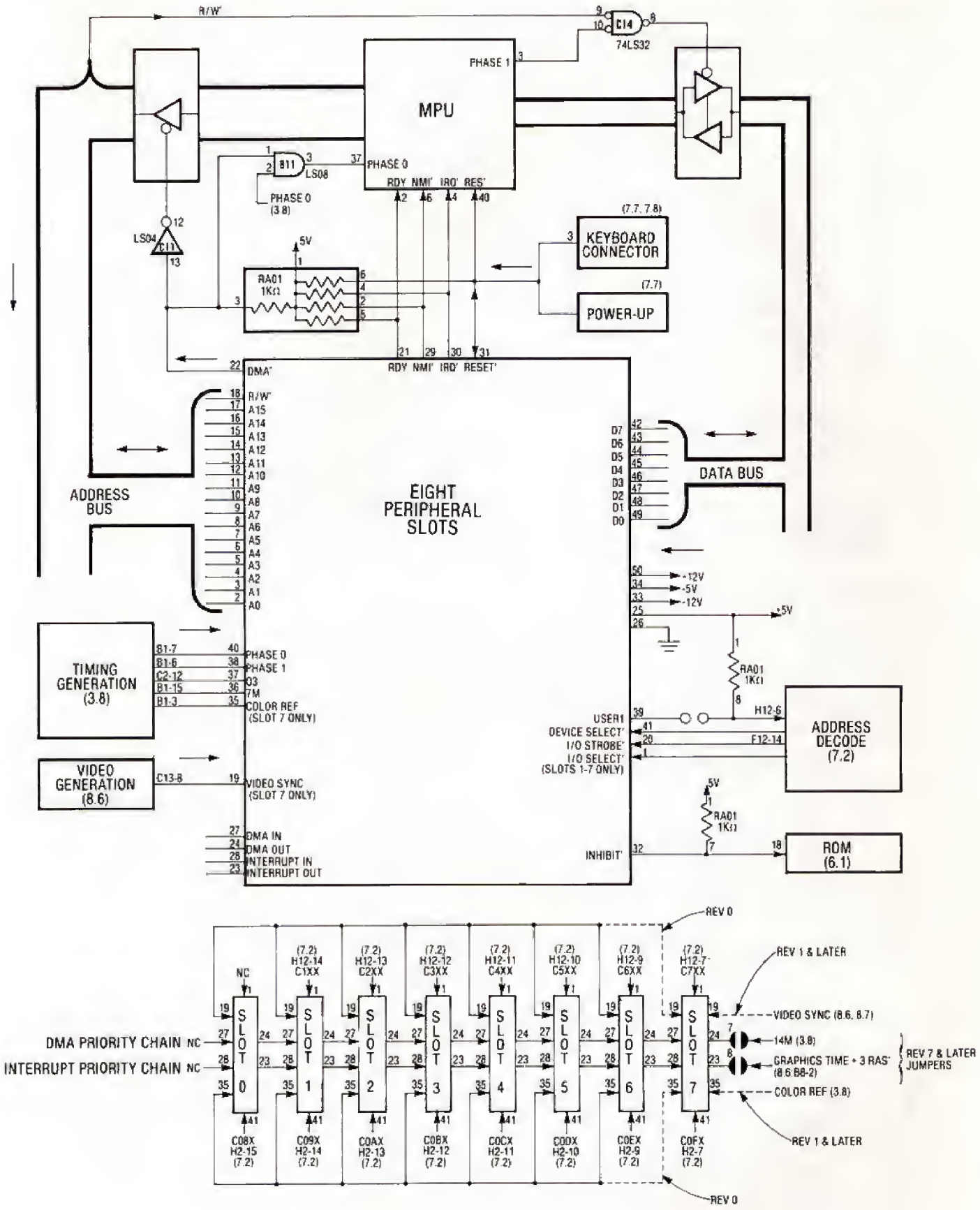
Serial I/O Devices (Figure 7.5)



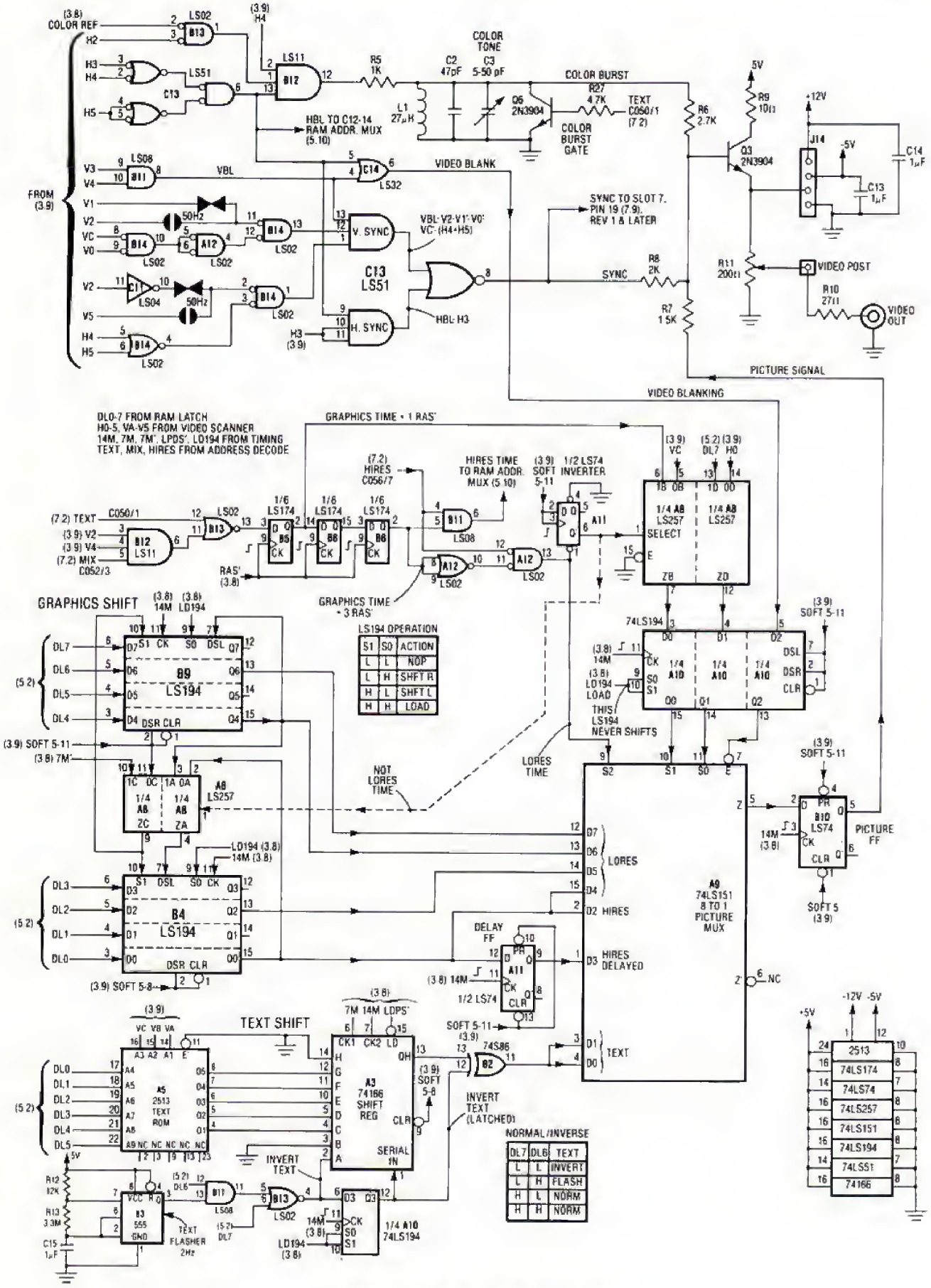
The Apple II Keyboard (Figure 7.7)



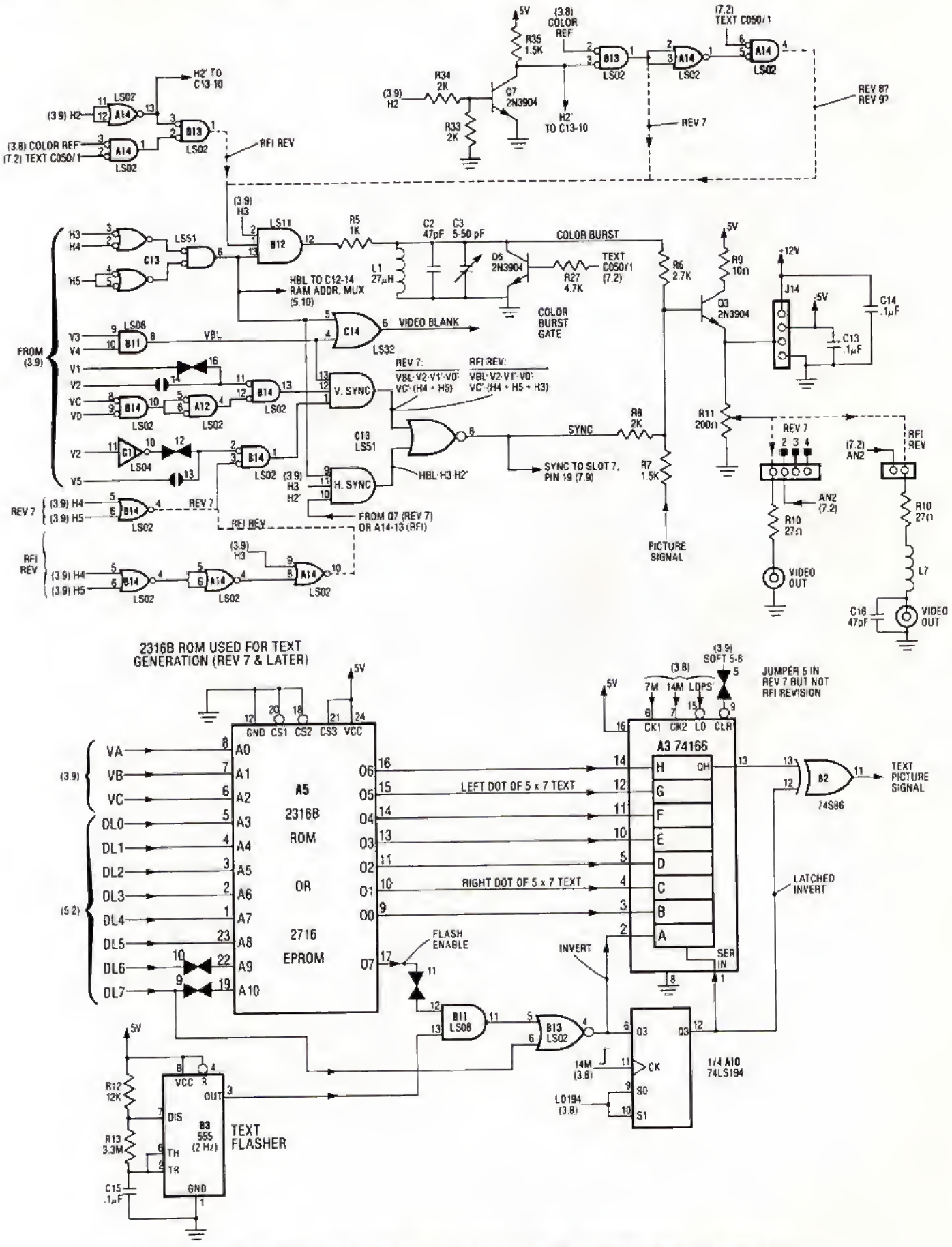
The Apple II Plus Keyboard and Encoder Board (Figure 7.8)



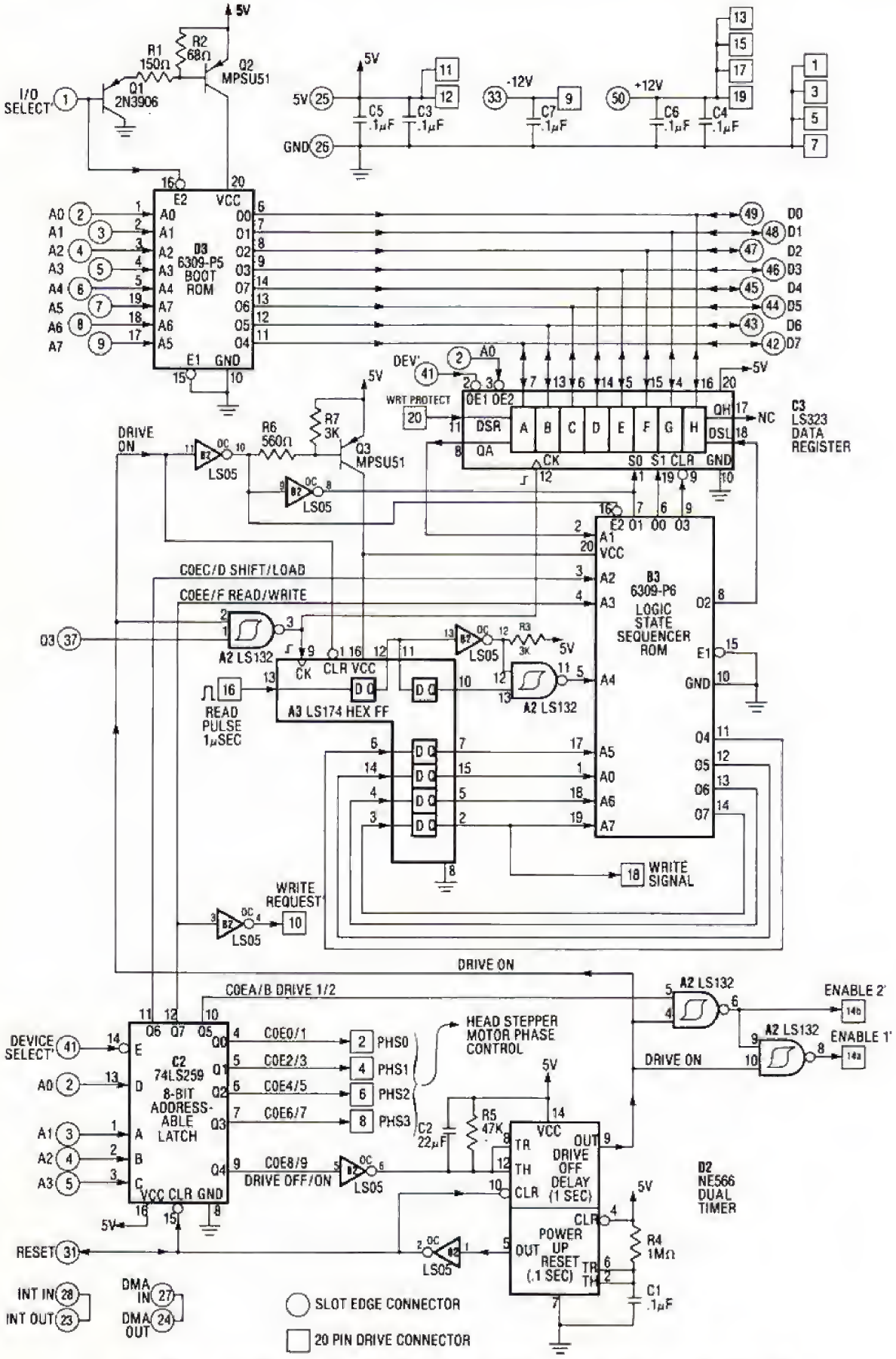
Peripheral Slot Connections (Figure 7.9)



Video Generation (Figure 8.6)



Revision-7 and RFI Version Changes to Video Generation (Figure 8.7)



The Disk II Controller (Address References Assume Slot 6) (Figure 9.8)

Index

FO1 and FO2 refer to the foldouts.

Aalto, Jim 9-34
active-when-high E-3
active-when-low E-3, 1-4 to 1-5
address bus 1-3, 2-1 to 2-13, FO1
  and address decoding 2-6 to 2-12, 7-1 to 7-8
  and DMA 2-9, 4-5, 4-13 to 4-14
  and MPU 4-2 to 4-5, 4-7 to 4-9
  and peripheral slots 7-17 to 7-20
  and RAM address multiplexor 2-9 to 2-11, 5-1 to 5-9, 5-21
  and ROM 6-2 to 6-4
  and serial I/O 2-6 to 2-12, 7-2 to 7-10
  and 6502 instructions 4-20 to 4-23
address decoding 2-6 to 2-12, 7-1 to 7-8, FO1
  and R/W' 7-8
  and 6502 1-3, 4-6
  list of functions 7-5
address fields, DOS 9-3, 9-4, 9-26 to 9-28, 9-39 to 9-42
  (also see DOS data formats)
AND gate E-1 to E-3, gl-1
AND instruction E-1 to E-5
ANIMATRIX 8-30
annunciator 1-8, 2-12, 7-2 to 7-3, 7-6, 7-10, FO1
Apple Computer, Inc. vi, 3-9, 4-1, 5-26, 5-28, 6-2, 6-4, 6-6, 6-8, 6-13, 7-20, 7-34, 9-3, 9-4, 9-9, 9-11, 9-17, 9-34, 9-41, 10-3, 10-5, 10-7, B-1, G-1 to G-3, H-1 to H-2, I-1 to I-3
Apple II
  history H-1 to H-2, I-1 to I-4
  overview 1-2 to 1-10
Apple II Circuit Description G-1, Schematics-1
Apple II Plus 1-4, 6-6, G-2
  keyboard 7-13, 7-15 to 7-18
Apple II Reference Manual v, 1-4, 3-9, 3-18, 4-11, 4-12, 4-15, 5-9, 5-11, 6-1, 6-4, 6-5, 7-12, 7-13, 7-15, 7-22, 7-24, A-1, E-4, G-1, G-2, Schematics-1
Apple II Reference Manual Addendum 3-9, 7-21
Apple II Reference Manual for IIe Only 1-4, E-4
Apple IIe v, 1-7, 4-26, 7-13, 10-5, I-2 to I-3
Applesoft Basic 1-4, 4-12, 4-20 to 4-21, 6-6, 6-8, G-2, gl-1
Applesoft Tutorial v
Applesoft II BASIC Programming Reference Manual v
Application Notes (see Hardware, Software Applications)
ASCII v, 1-6, 1-7, 7-17 to 7-18, 8-30 to 8-32, gl-1
aspect ratio 8-28
assembler 4-11 to 4-12
assembly language vi, 4-11 to 4-12, gl-1
Atari, Inc. 4-1, H-2
audio (see Speaker)
Autostart ROM 6-6
  and Apple II Plus 6-6, G-2
  and Apple II reliability 10-3
  and disk boot 4-15, 9-1, 9-11
  and interrupts 4-15 to 4-18
  and I/O links 7-22
  and monitor modification 6-16 to 6-17
  and RESET' 4-15, 6-16 to 6-17, 9-11
  and 16K RAM card 5-31
  on firmware card 6-11, 6-18 to 6-21
  (also see monitor)
AX signal 3-3 to 3-10, 5-2, 5-6, 5-23 to 5-25

bandwidth 8-6, 8-33 to 8-34
bank switching
  motherboard RAM 5-34 to 5-35, I-2 to I-3
  motherboard ROM 6-1, 7-19 to 7-20
  RAM card bank 1/bank 2 5-26 to 5-30
  (also see INHIBIT')
baseplate 1-2
  removal J-1
BASIC vi, 1-1 to 1-2, 6-5 to 6-6, gl-1, I-2 to I-4
  and disk I/O 9-34, 9-37
  and paddle programming 7-25
  and Page 0/Page 1 4-5
  and 6502 instruction details 4-20 to 4-21
  in ROM 1-1 to 1-4, 2-6, 6-1, 6-5 to 6-6
  programming 4-12
  (also see Applesoft BASIC; Integer BASIC)
Baum, Alan 7-8, H-1, I-2
Baum, Peter I-2
Beneath Apple DOS 9-3, 9-37
bidirectional bus drivers 4-2 to 4-3, 4-5, G-3
  and DMA 4-13
  and write cycle 5-24 to 5-25, 7-9
  (also see MPU; transceiver)
binary information 1-3, F-1 to F-3
binary number system vi, F-1 to F-3, gl-2
Bishop, Bob 5-36
bit 1-3, F-1, gl-2
blanking 3-10 to 3-12, 8-3 to 8-5, 8-28
  (also see HBL; VBL; video signals)
Boole, George E-4
Boolean algebra E-4 to E-5
Bootstrap ROM (also see ROM) 9-1, 9-10 to 9-12, gl-2
BREAK
  flag 4-4, 4-18
  handler 4-17 to 4-18
  instruction 4-17 to 4-19
  vector 4-18
breakpoint
  hardware 4-29 to 4-32
  software 4-17, 6-16

bus 2-1 to 2-3, gl-2, FO1
address (see address bus)
Apple 2-3 (see peripheral slots)
data (see data bus)
drivers 2-1 to 2-3, 4-2 to 4-3, 4-5, gl-2
fights 6-2 to 6-4, 6-8, 6-11, I-3
multiplexed RAM address (see RAM)
peripheral 2-3 (see peripheral slots)
RAM 2-3
ROM 2-3
secondary buses 2-9 to 2-12
byte 1-3, gl-2
BYTE FLAG 9-26, 9-29 to 9-34, gl-2
BYTE Magazine 5-32, 6-5
card cage 1-4, gl-2
(also see peripheral slots)
CAS' 3-3 to 3-10, 5-2 to 5-6, 5-20 to 5-35
cassette I/O 1-8, 7-10 to 7-12, FO1
and address decoding 7-2 to 7-6
and bus structure 2-6 to 2-8
and D MAnual Controller 4-27
read/write routines 6-16
schematics 7-6, 7-10
cathode ray tube (CRT) gl-2 (also see video)
character sets 8-9, 8-30 to 8-32
chip select, ROM 6-1 to 6-3, 6-13, 6-21
chrominance signal v, 8-5 to 8-6, 8-33 to 8-34
circuit symbols 2-2 to 2-3, E-1 to E-5
clockpulse 1-2 to 1-3, 3-2 to 3-8, 4-2, 4-6 to 4-10, E-3
clockpulse jitter 1-2, 3-18, 9-24 (also see long cycle)
cold start reset 4-15
collector-OR (see wire-OR)
COLOR BURST 3-9, 8-3 to 8-6, 8-8 to 8-13
Color Burst Killer 8-11 to 8-13, 8-29, G-2 to G-3
COLOR DELAY' 3-5 to 3-7
color graphics 1-5 to 1-6, 8-6 to 8-7, 8-18 to 8-25
COLOR REFERENCE 3-3 to 3-9, 8-5, 8-15, 8-18 to 8-25
color signals 8-5 to 8-7, 8-15, 8-18 to 8-25, 8-33 to 8-34
color subcarrier 8-33 to 8-34
colors 1-6, 8-6 to 8-7, 8-18 to 8-25
COLUMN address 2-11, 3-9, 5-1 to 5-6, 5-23 to 5-24
COLUMN limited RAM access 5-32
command decoder 9-2, 9-10 to 9-14
(also see disk controller)
Commodore 4-1
complement gl-2
complementary colors gl-2, 8-15, 8-18 to 8-19
compilers 4-12, gl-2
composite video 8-3, 8-6, gl-2 (also see video)
COUT1 7-21 to 7-22
CP/M (Control Program for Microprocessors) 4-13, 5-26
CSW (Character output SWitch) 7-21 to 7-23
cursor 8-31 to 8-32
cursor moves 1-7, 6-6, 6-11, 6-16
cycle stealing 4-13 to 4-14, 4-24
data bus 1-3, 2-1 to 2-13, FO1
and DMA 2-9, 4-13
and MPU 2-4 to 2-10, 4-2 to 4-5, 4-7 to 4-10
and peripheral slots 7-17 to 7-20
and ROM 6-2 to 6-4, 6-6 to 6-8
and serial inputs 2-7 to 2-12, 7-2 to 7-10
and 6502 instructions 4-20 to 4-23
driver (see bidirectional bus driver)
management 2-3, 4-7 to 4-10
management gates 7-3
RAM/keyboard connection 2-11 to 2-13, 5-3 to 5-5, 7-2
(also see timing diagrams; RAM SELECT')
data fields, DOS 9-3 to 9-4, 9-26 to 9-28, 9-39 to 9-42
debounce 4-24 to 4-25, 6-18, 7-13 to 7-17
debug 4-12, 4-17, gl-2
decimal number system F-1, gl-2
DEVICE SELECT' 4-8 to 4-10, 7-2 to 7-9, 7-19 to 7-21
Digital Research 4-13, B-1
DIP (Dual In line Package) 1-8, gl-3
disassembler 4-11, gl-2
disk controller 9-10 to 9-34
and bus structure 2-7 to 2-9
Bootstrap ROM 9-1, 9-10 to 9-12
command decoder 9-11 to 9-14
data register 9-17 to 9-24, 9-29 to 9-31
drive select 9-13
head positioning commands 9-13
logic state sequencer (see logic state sequencer)
power-up reset 9-13
read pulse processing 9-15 to 9-16, 9-29 to 9-35
READ/WRITE 9-13 to 9-14
SHIFT/LOAD 9-14
WRITE PROTECT signal 9-8, 9-12, 9-17
WRITE REQUEST' 9-7, 9-8, 9-13 to 9-14
WRITE signal 9-7, 9-22 to 9-24
disk drive 9-5 to 9-11
analog card 9-3, 9-6, 9-17, I-1
apparent momentum 9-13
enabling 9-5, 9-13
erase head 9-6 to 9-8
motor speed up time 9-38
power supply 9-5, 9-38
read interface chip 9-9 to 9-11
read pick up signal 9-9
read pulse 9-9 to 9-11
read/write head 9-5 to 9-7
reliability and repair 10-2, 10-5
stepper motor 9-2 to 9-7
stepper motor response time 9-7, 9-38 to 9-39
write protect switch 9-8 to 9-9
write protect switch bypass installation 9-43 to 9-45
writing to disk 9-7 to 9-8
disk I/O 2-7 to 2-9, 9-1 to 9-45
bypassing write protection 9-43 to 9-45
controller (see disk controller)
data formats (see DOS data formats)
data paths 9-1 to 9-5
DOS (see DOS)
drive (see disk drive)
hard sector 9-3
head positioning 9-5 to 9-7, 9-11, 9-13, 9-38 to 9-39
programming 9-11 to 9-15, 9-17 to 9-24, 9-29, 9-34 to 9-42
read process 9-4 to 9-5
RWTS (see RWTS)
soft sector 9-3 to 9-4
write process 9-5
write protection 9-6, 9-8 to 9-9, 9-12, 9-17 to 9-22
display, video (see video; screen)
DMA (Direct Memory Access) 1-5, 2-9, 4-13 to 4-15, gl-3
and MPU 1-5, 2-9, 4-2 to 4-5, 4-12 to 4-14
and maximum PHASE 0 hold off 4-13 to 4-14
and READY 4-14
cycle stealing 4-13, 4-24 to 4-26
direct bus access 1-5, 2-9, 4-14
D MAnual Controller 4-24 to 4-27
DMA IN/OUT signals 7-19, 7-21
DMA' signal 2-9, 4-2 to 4-5, 4-12 to 4-14, 7-19, 7-20
from video scanner 4-1, 8-1 to 8-2
priority chain 4-14 to 4-15, 4-25 to 4-27, 5-31, 6-8 to 6-10, 7-19, 7-21
simultaneous DMA 2-11, 4-13
(also see video scanning)

Dokay Computer Products 5-32
DOS 9-1 to 9-7, I-4
and firmware card 5-29, 6-8 to 6-10
and I/O links 7-22 to 7-23
and RAM card 5-28 to 5-30
TOOL KIT 8-30 to 8-32
3 9-3, 9-26
3.2 9-26 to 9-34
3.3 9-26 to 9-42
DOS data formats 9-24 to 9-27, 9-34
address field 9-3, 9-4, 9-27 to 9-28
bootstrap incompatibility 9-26, 9-34
checksum 9-26, 9-42
data field 9-3, 9-27 to 9-28
data field misalignment 9-40 to 9-41, 9-42
field identifiers 9-26 to 9-28, 9-34
read syncing leaders 9-24, 9-27 to 9-28, 9-30
restrictions 9-26 to 9-27
sector 9-3 to 9-4
sector interleaving 9-39 to 9-42
track 9-3, 9-7
track to track synchronization 9-40
write tables 9-26, 9-27
(also see RWTS programming examples)
dynamic RAM 1-4, 5-1 to 5-4, gl-3 (also see RAM)
EPROM 6-13 to 6-17
adaptor 4-29, 6-13 to 6-15
and NMI STEPPER 4-29
and Revision 7 G-2
configuring firmware card for 4-29, 6-9, 6-11, 6-21
creation for system monitor 6-16 to 6-17
manufacturers of 6-14
products related to 6-13 to 6-14
programming screen character sets 8-30 to 8-32
Espinosa, Chris H-2, I-3
Eurapple jumpers 1-5, 3-12, 3-14, 8-12 to 8-14, G-2
Eurocolor 1-5, 8-14
exclusive-OR gate gl-3, E-2
expansion ROM 6-1 to 6-2 (also see seventh ROM)

fan, cooling 10-3 to 10-4
FCC regulations 8-3, G-1, G-3
figures (see appendix K)
firmware 1-1 to 1-4, 2-6, 6-4 to 6-6, 9-11
(also see Applesoft; Autostart; BASIC; Integer; monitor; ROM)
firmware peripheral card 1-8, 6-6 to 6-8, G-2, I-2
and DMA Controller 4-26 to 4-27
and DMA priority chain 4-15, 4-26, 5-31, 6-10
jumpers 6-11, 6-18 to 6-21
modifications 6-18 to 6-21
timing 6-11 to 6-12
flashing text 1-6, 8-7 to 8-8, 8-12 to 8-13, 8-17, 8-30 to 8-32
flip-flop 9-24, E-3, gl-3
floating bus 4-8 to 4-10, 5-25 to 5-26, 5-36, 7-8, I-2
(also see timing diagrams)
floating point routines 6-5 to 6-6
floppy disks 9-3, 9-24
Fourth Dimension 9-45
frequencies, Apple 3-3 to 3-4, 3-18
front panel 4-29
Fujitsu 5-32
F8 ROM 4-15, 6-5, 6-16
and firmware card 6-11, 6-18 to 6-21
and 16K RAM card 5-31
(also see monitor; ROM)

game I/O socket 1-8, 7-2 to 7-4, 7-9 to 7-11
and SHIFT key mod 7-36 to 7-37
extending 1-8 to 1-9, 7-28 to 7-33
gates (logic) 2-6, E-1 to E-5, gl-3
Gayler, Winston G-1, Schematics-1
General Instrument 6-1, 7-15, 8-16
GETLN 7-21 to 7-22
GRAPHICS mode 1-5 to 1-7, 8-7 to 8-9
(also see LORES; HIRES)
GRAPHICS/TEXT soft switch 1-7, 7-2 to 7-6

hacker 6-6, gl-3
Hardware Applications gl-3
Bank switching the motherboard RAM 5-34
D MAnual Controller 4-24 to 4-27
Detecting television SYNC 3-19 to 3-26
Eliminating colored shadows from text 8-29
EPROM in the Apple 6-13 to 6-15
Extending the game I/O socket 7-28 to 7-32
Installing volume control on Speaker 7-39 to 7-40
Installing WRITE PROTECT switch on Disk II drive 9-43 to 9-45
Making the shift key modification 7-36 to 7-38
Modifying the firmware card for independent selection of the F8 ROM 6-18 to 6-21
Modifying the keyboard so control and RESET must be pressed to cause a RESET 7-34 to 7-36
Modifying the system monitor 6-16 to 6-17
Multiple RAM card configurations 5-42
NMI' based single stepper 4-28 to 4-32
Programming screen character sets in EPROM 8-30 to 8-32
Upgrading Apples to 48K RAM 5-32 to 5-33
Using paddles and joysticks not designed for the Apple 7-33
HBL (Horizontal BLanking) 8-3 to 8-5, 8-9 to 8-13
and memory scanning 5-9 to 5-20, 5-36 to 5-41
head positioning (see disk I/O)
Hertz (Hz) 1-2, gl-3
hexadecimal number system vi, F-2 to F-3, gl-3
high frequency rejection filter g-3
high level language 1-2, 4-10 to 4-12, gl-3
HIRES graphics 1-5 to 1-7, 8-18 to 8-22
character sets 8-30 to 8-32
colors 1-6, 8-6 to 8-7, 8-18 to 8-22
delayed video 1-6, 8-18 to 8-22
distance ratios 8-28
generation 8-8, 8-12, 8-15, 8-18 to 8-19
interference 8-20 to 8-22
memory scanning 5-12 to 5-21, 5-36 to 5-38
resolution 1-6, 8-19
(also see LORES; video)
Holt, Rod 9-21, H-2, I-1, I-2
Homebrew Computer Club H-1
horizontal blanking (see HBL)
horizontal counter 3-11, 3-14 to 3-15
horizontal PERIOD 8-9 to 8-14
horizontal retrace 3-11, 5-13, 8-4, 8-10
horizontal scan 3-10 to 3-12, 8-4 to 8-5, 8-10
horizontal sync 3-10, 5-11, 5-15 to 5-18, 8-3 to 8-5, 8-9 to 8-13
HRCG (HIRES Character Generator) 8-30

I/O (Input/Output) 1-4 to 1-9, 2-6 to 2-12, Chapters 7-9
and address decoding 2-6 to 2-12, 7-1 to 7-8
and bus structure 2-6 to 2-12
and firmware 6-5, 7-21 to 7-23
cassette 7-2 to 7-12
disk 9-1 to 9-45
game socket 7-2 to 7-11
keyboard 7-12 to 7-18
links (CSW and KSW) 7-21 to 7-23
port 1-8, 2-7, 7-9, gl-4
peripheral slots 7-19 to 7-23
serial I/O 2-12, 7-2 to 7-12
speaker 7-1 to 7-12
timing 7-8 to 7-9
video 3-10 to 3-16, 8-1 to 8-34

I/O SELECT' 6-4, 7-2 to 7-9, 7-19 to 7-21
and disk controller 9-10 to 9-12
I/O STROBE' 6-2 to 6-4, 7-2 to 7-9, 7-19 to 7-21, I-2
protocol 6-4
(also see seventh ROM)
IC (see integrated circuits)
impedance 2-1 to 2-3, gl-3, E-3
indirect addressing 4-5
INHIBIT' 6-2 to 6-4, 6-9 to 6-12, 7-19 to 7-20
input buffer, GETLN 6-5, 7-21
Input/Output (see I/O)
Integer BASIC 1-4, 4-12, 5-38, 6-5, 6-8, I-3 to I-4, G-2, gl-4
integrated circuits 1-2, 1-8, gl-4, E-3
location FO2
troubleshooting 10-3 to 10-10
2316/9316 ROM 6-1 to 6-3, 6-6 to 6-8
2513 text ROM 8-9, 8-12, 8-16 to 8-17, I-2
2716 EPROM 6-13 to 6-15
3470 floppy read interface 9-9 to 9-11
3600 keyboard encoder 7-15 to 7-17
4116 RAM 5-1 to 5-3, 5-32 to 5-33
558 quad timer 7-10 to 7-11
5740 keyboard encoder 7-13 to 7-15
6309 PROM 9-22 to 9-23
6502 MPU 4-1 to 4-10, 4-22, C-1 to C-7
74LS138 decoder 6-2 to 6-3, 6-9, 7-4
74LS148 priority encoder 4-24 to 4-25
74LS251 multiplexor 6-4, 7-11 to 7-12
74LS74 dual D flip-flop E-3 to E-5
741 op amp 7-10 to 7-12
8T28 quad transceiver 4-3, 4-5, 6-2, 6-4
8T97 hex tri-state driver 4-3, 4-5
8304 octal transceiver 4-3, 4-5, 6-2, 6-4
Intel 4-14, 6-13
interlacing
frequency 8-6, 8-34
television scan 3-12, 8-5, gl-4
interpreter 4-12, 6-5, gl-4
interrupts, 6502 4-3 to 4-4, 4-15 to 4-19
acknowledge 3-19 to 3-24, 4-16
BREAK 4-17 to 4-19
handlers 4-16 to 4-19
INTERRUPT IN/OUT 7-19, 7-21
IRQ' 4-3 to 4-4, 4-16 to 4-19, 7-19, 7-20
NMI STEPPER 4-28 to 4-32
NMI' 4-3 to 4-4, 4-16 to 4-19, 7-19 to 7-20
polling 4-17
priority among interrupts 4-19
priority chain 4-17, 7-19, 7-21
RESET' 4-3 to 4-4, 4-15 to 4-16, 4-19
stacked interrupts 4-17
vectors 4-15 to 4-18
video scan interrupting 3-19 to 3-24
(also see IRQ'; NMI'; RESET')
inverse text 1-6, 8-7 to 8-8, 8-12 to 8-13, 8-17, 8-30 to 8-32
INVERT TEXT signal 8-12 to 8-13, 8-16 to 8-17
IRQ' (Interrupt ReQuest) 4-3 to 4-4, 4-16 to 4-19, 7-19, 7-21
handler 4-16 to 4-19
hard vector 4-16 to 4-18
soft vector 4-17

Jameco Electronics 5-32, 7-33
JDR Microdevices 5-32
Jobs, Steve vi, H-1 to H-2
John Bell Engineering 6-13, 6-14
joystick 1-8, 7-9 to 7-11, gl-4 (also see paddles; timers)
jumpers FO2
Eurapple 1-5, 3-12, 3-14, 8-12 to 8-14
firmware card 6-9, 6-11, 6-18 to 6-21
keyboard 7-16, 7-17, 7-37 to 7-38
RAM card 5-27, 5-31
TEXT ROM 8-13
USER1 7-6, 7-8, 7-19, 7-20
7 and 8 7-19, 7-21, 7-22, G-3

Kane, Gerry 4-13
keyboard 1-7, 2-11, 7-12 to 7-18, FO1
alphabetic shifting 1-7, 7-16 to 7-17, 7-36 to 7-38
and address decode 7-2 to 7-6
Apple II Plus 7-13, 7-15 to 7-18, G-2
ASCII 1-7, 7-17 to 7-18
CTRL required for RESET 7-15 to 7-17, 7-34
encoder board 7-15 to 7-16
encoder ROM 7-13 to 7-17
input buffer 6-5, 7-21
keybounce mask 7-13 to 7-17
numeric keypad 7-15 to 7-16
parity bit 7-13
RAM/keyboard multiplexor 2-11, 5-4, 7-2, FO1
repeat oscillator 7-13 to 7-17
SHIFT key mod 7-36 to 7-38
special function keys 1-7, 7-17 to 7-18
STROBE 7-13 to 7-17
strobe flip-flop 7-14 to 7-15
KEYIN 1-7, 7-21
Kraul, Doug H-2
KSW (Keyboard input SWitch) 7-21 to 7-23

Language card (see RAM card)
Language system 5-26 to 5-28, 6-6
LDPS' 3-3 to 3-10, 8-14 to 8-16
LD194 3-3 to 3-10, 8-14 to 8-16
Lechner, Pieter 9-3, 9-37
LOGIC DATABOOK E-3
logic equations (Boolean algebra) E-4 to E-5
logic levels 1-3 to 1-5, 1-8, E-1
logic state sequencer 9-12, 9-15 to 9-25, 9-29 to 9-35, I-2
commands 9-15
decoding the contents 9-16 to 9-19
listings 9-20, 9-21
P6 PROM 9-12, 9-14 to 9-17
read pulse input 9-12, 9-15 to 9-16, 9-32
READ sequence 9-5, 9-20, 9-21, 9-26, 9-29 to 9-35
sequencing flip-flops 9-12, 9-14 to 9-15
WRITE PROTECT sequence 9-17, 9-20, 9-21
WRITE sequence 9-15, 9-17 to 9-25
logic symbols E-1 to E-5
long cycle 3-2 to 3-6
and disk I/O 9-24
and timing loops 3-18
and 6502 communication 4-6
detection 3-21 to 3-23
reason for 3-10, 3-12
LORES graphics 1-5 to 1-7, 8-22 to 8-25
circular patterns 8-23 to 8-25
colors 1-6, 8-6 to 8-7, 8-23 to 8-25
distance ratios 8-28
generation 8-8, 8-12, 8-15, 8-22 to 8-25
memory scanning 5-7 to 5-13, 5-36 to 5-38
resolution 1-6
(also see HIRES; video)
LORES TIME 8-12, 8-26
LORES/HIRES Soft switch 1-7, 7-2 to 7-6
LSTTL 1-8, 3-15, 5-42, 7-11, gl-4 (also see TTL)
luminance signal 8-5 to 8-6, 8-33 to 8-34

machine cycles 3-3 to 3-9, 4-6 to 4-10, gl-4
(also see long cycle)
machine language 1-2, 4-10 to 4-12, F-2, gl-4
maintenance and care 10-1 to 10-10
Markkula, Mike vi, 9-3, H-2
Mazur, Jeffrey 6-13, 9-3

memory 1-3 to 1-4, 2-6
cell 2-9, 5-2, gl-4
display areas 1-5 to 1-7
location 2-9, 5-2
pages 4-5, gl-5
scanning 1-5, 2-11, 3-10, 5-6 to 5-20, 5-36 to 5-41, 8-1 to 8-2
scanning maps 5-10 to 5-19, 5-37
6502 usage 1-3, 4-5 to 4-6
(also see RAM; ROM; DMA)
memory mapped I/O 4-6, gl-4 (also see address decoding)
memory mapped video 8-1 to 8-2, gl-4
Microcomputing 8-30
microprocessing unit (see MPU)
Microproducts 6-13, 6-14
Microsoft 4-14, 4-26, 6-5, B-1, I-3
microsecond 3-4, gl-4
millisecond (msec.) gl-4
Mini-Assembler 4-11 to 4-12, 6-5 to 6-6
MIXED mode 1-7, 8-11
scanning 5-5, 5-13, 5-19
switching 5-5, 8-11 to 8-14, 8-25 to 8-27
modulation 1-5, 8-3, 8-33, gl-4 (also see RF modulator)
monitor, system 1-4, 4-15, 6-5 to 6-6,
Autostart 4-15, 6-6, 7-22
in ROM 1-4, 2-6, 6-5
listing 6-5
old monitor 4-15, 6-5, 7-21
monitor, video 8-3, 8-6, 8-17, 8-19 to 8-20
Monitor ROM (old) 6-5
and firmware card 6-11, 6-18 to 6-21
and I/O links 7-21
and interrupts 4-15 to 4-18
and RESET' 4-15
(also see Autostart ROM)
Moore, Robin 8-30
MOS integrated circuit 1-8, 6-1, gl-5
MOS Technology 4-1, 4-7 to 4-8, 4-13, 5-5, 6-1, C-1, C-7, I-3
most significant bit (MSB) 2-7, 2-11, 9-14, gl-5
(also see BYTE FLAG)
motherboard 1-2, gl-4
I/O 1-8, 7-1 to 7-12
part numbers G-1 to G-2
removal J-1
revisions G-1 to G-3
(also see revision)
Motorola 4-6, 4-14, 6-13, 9-9, I-1
Mountain Computer 6-14
MPU (Microprocessing Unit) 1-2 to 1-3, 2-5 to 2-6, gl-4
MPU, 6502 4-1 to 4-32, C-1 to C-8, FO1
advantages/disadvantages 4-11
and Apple I H-1
and bus structure 2-5 to 2-13, FO1
and DMA 4-13 to 4-15
and peripheral slots 1-4 to 1-5, 4-3, 4-5, 7-19 to 7-20
clock pulses 3-5 to 3-9, 4-2, 4-6 to 4-10
connections 4-2 to 4-5, 7-19, FO1
data sheet C-1 to C-8
instruction details 4-20 to 4-23, C-8
internal registers 4-10 to 4-11
interrupts 4-4, 4-15 to 4-19
machine cycle 3-3 to 3-9, 4-6 to 4-10
manufacturers 4-1, C-1
maximum clock holdoff 4-13 to 4-14, I-3, C-7
memory usage 1-3, 4-5 to 4-6
programming 4-10 to 4-12, F-2 to F-3
related signals 4-2 to 4-4
stack 4-5, gl-6
timing 4-6 to 4-10, C-1 to C-8, I-1 to I-2
(also see DMA; interrupts; timing diagrams)
multiplexed RAM address (RA0-RA6) 2-9 to 2-11, 5-5 to 5-7, 5-21, EO1
and RAM card 5-28, 5-31, 5-42, I-3
reflections on 5-22, 5-31, I-2 to I-3
termination resistors 5-21, 5-22, I-2
multiplexing gl-5
PICTURE signal 8-7, 8-8, 8-12
RAM address 2-9 to 2-11, 5-6
RAM/keyboard data 2-11, 5-4
serial inputs 2-12, 7-2
NAND gate E-2, E-3, gl-5
nanoseconds (nsec.) 3-4, gl-5
National Semiconductor 7-13, B-1, E-3, H-2
NEC 5-32
NMI' (Non-Maskable Interrupt) 4-3 to 4-4, 4-16 to 4-19, 7-19, 7-21
handler 4-16 to 4-19
hard vector 4-16
NMI STEPPER 4-28 to 4-32
soft vector 4-17, 4-29
NMOS integrated circuits 6-1
Nintendo . B-1
NOMIX/MIX soft switch 1-7, 7-2 to 7-6
NOR gate gl-5, E-2
normal text 1-6, 8-7 to 8-8, 8-12 to 8-13, 8-17, 8-30 to 8-32
NTSC television 8-3 to 8-6, 8-14
number systems vi, F-1 to F-3
numeric keypad 7-15, 7-16
object program 4-11 to 4-12, gl-5
octal number system F-2, gl-5
Ohio Scientific 4-1
op code 4-10 to 4-11, 4-20 to 4-23, gl-5
open collector 4-5, 7-11
operand 4-10 to 4-11, gl-5
OR gate E-3 to E-5, gl-5
Osborne, Adam 4-13
Osborne 4 & 8 Bit Microprocessor Handbook 4-13
output enable 2-2 to 2-3, E-2 to E-3
(also see tri-state; data bus management)
paddles 1-8, 7-9 to 7-11
and game socket extender 7-28 to 7-32
calibration 7-33
non standard 7-33
programming 7-24 to 7-27
quad timer 1-8, 7-2 to 7-11, FO1
pages, display 1-7, gl-5
pages, memory 4-5, gl-5 (also see memory scanning)
Page 1/Page 2 soft switch 1-7, 7-2 to 7-6
parallel data transfer 1-8, gl-5
PEEK 4-20 to 4-21
peripheral card check 10-6
peripheral card failures 10-8
peripheral slots 1-4 to 1-5, 7-17 to 7-23, FO1
and address decoded signals 7-2 to 7-9, 7-19 to 7-21
and bus structure 1-4 to 1-5, 2-3, 7-17 to 7-20, F01
and I/O links 7-21 to 7-23
connections 4-3, 7-2, 7-19 to 7-21, FO1
reliability 10-2
phase relationships, color 3-9, 8-15, 8-18 to 8-25
PHASE 0 3-3 to 3-9, 4-2 to 4-10
(also see timing diagrams)
PHASE 1, Apple 3-2 to 3-9, 4-7, 7-3
PHASE 1, 6502 4-2 to 4-3, 4-6 to 4-7, 5-24 to 5-25
PHASE 2 4-2 to 4-3, 4-6 to 4-10
(also see timing diagrams)
phases, stepper motor 9-5 to 9-7, 9-12 to 9-13
PICTURE signal 8-3, 8-6 to 8-8, 8-12, 8-15

PICTURE signal multiplexor 8-7, 8-8, 8-12
POKE 4-20 to 4-21
positive logic 1-4, E-1, gl-5
potentiometer (pot) 1-8, 7-11, 7-33, 7-39
power supply 1-2, 1-9, gl-5
  reliability 10-2 to 10-3
  to disk drive 9-5, 9-6
  to peripheral slots 7-19, 7-20
  troubleshooting/failures 10-6 to 10-7, 10-9
power-up byte 4-15
power-up reset 2-6, 4-3, 7-14, 7-15, 7-19, G-2
  on disk controller 7-15, 9-12, 9-13
PREAD 7-24 to 7-25
primary I/O device 6-5, 7-21 to 7-23
prime (') notation 1-4, E-3
printer reliability 10-2
priority chains 7-19, 7-21
  DMA 4-14 to 4-15, 4-25 to 4-27, 5-31, 6-8 to 6-10
  interrupt 4-17
processor status register 4-10 to 4-11, 4-16 to 4-18, gl-6
program counter 4-10 to 4-11, 4-16 to 4-18, gl-6
programming 4-10 to 4-13
  (also see memory scanning maps; software applications)
PROM (see EPROM)
propagation delay 3-4, gl-6
  CAS' 5-20
  disk controller clock 9-14, 9-22, 9-23
  in timing generator 3-4 to 3-7
  PHASE 0 to RAM data valid 5-20
  picture flip-flop 8-17, 8-25
  RAS' to RAM data valid 5-5
  (also see timing diagrams)
pull-down resistor 7-9 to 7-11
pull-up resistor 4-5, 4-26, 7-11
pushbutton inputs 1-8, 7-2 to 7-5, 7-9 to 7-11
  and game I/O extension 7-28 to 7-33
  and SHIFT key mod 7-36 to 7-37
P5 PROM (see Bootstrap ROM)
P6 PROM (see logic state sequencer)

quad timer 1-8, 7-2 TO 7-11, 7-24 TO 7-27, FO1
  fixed timer reference 7-33
  (also see paddles; timers)
Quality Software 9-3
Quest Electronics 5-33

R/W' 2-3 to 2-13, 4-2 to 4-3
  and address bus 2-3, 4-2, 4-3
  and address decoding 7-8
  and RAM 5-3, 5-4, 5-24 to 5-25
  and ROM 6-2, I-3
  and write cycle 5-24 to 5-25, 7-8 to 7-9
R.H. Electronics 10-3
radio frequency (RF) 1-5, 8-3, 8-33 to 8-34
Radio Shack 10-6
RAM (read/write memory) 1-3 to 1-4, 5-1 to 5-42, gl-6, FO1
  and Apple bus structure 2-2 to 2-13
  bank switching the motherboard RAM 5-34 to 5-35
  chip organization 2-12 to 2-13, 5-3, 5-4, FO1
  chip suppliers 5-32 to 5-33
  connections 5-3 to 5-5, FO1
  data latch 2-11, 5-3 to 5-5, FO1
  diagnostic program 10-8
  dynamic RAM chip 5-1 to 5-4, 5-32 to 5-33
  latched data (DL0-DL7) 5-4, 8-8, 8-12, FO1
  manufacturers 5-33
  R/W' 5-3 to 5-4, 5-24 to 5-25
  RAM bus 2-3
  RAM/keyboard data multiplexor 2-11, 5-3, 5-4, FO1
  read cycle 5-22 to 5-24
  reading video data from program 5-36 to 5-41
  refreshing of 1-4, 5-2, 5-19 to 5-20
  scanning (see memory scanning)
  timing 5-22 to 5-26
  troubleshooting 10-8, 10-9
  write cycle 5-24 to 5-25
  4K RAM chip 5-1, 5-19, 5-32, G-2
RAM address multiplexor 5-2, 5-5 to 5-22, FO1
  address assignments 5-6, 5-7, 5-19 to 5-20
  and bus structure 2-9 to 2-11, FO1
  hardware 5-6, 5-20 to 5-22
  HIRES scanning 5-12 to 5-21, 5-36 to 5-38
  MIXED mode scanning 5-5, 5-13, 5-19
  offset generation 5-6 to 5-9
  RAM SELECT' 5-5, 5-6, 5-21
  RA0 to RA6 (see multiplexed RAM address)
  TEXT/LORES scanning 5-7 to 5-13, 5-36 to 5-38
  UNUSED 8 5-7 to 5-19, 5-37
RAM card, 16K 5-26 to 5-31, I-2, I-3
  advantages, disadvantages 6-6
  and D MAnual Controller 4-27
  and DOS 5-28 to 5-29
  multiple RAM cards 5-31, 5-42, 6-11
random access memory 1-3, 6-1, gl-6 (also see RAM; ROM)
RAS' 3-3 to 3-10, 5-2 to 5-6, 5-23 to 5-32
  and MIXED mode switching 8-11 to 8-14, 8-26
RAS' only refresh 5-2 to 5-3
RAS'CAS' refresh 5-2 to 5-3
raster 3-10, gl-6
RDKEY 8-17
read cycle 2-3 to 2-5, 5-22 to 5-26, 7-8 to 7-9
read pulse (see disk topics)
READ sequence (see logic state sequencer)
read-modify-write instructions 4-20 to 4-23
read/write control (see R/W')
read/write memory (see RAM)
READY 4-3, 4-4, 4-14
refreshing RAM 1-4, 5-2, 5-19 to 5-20
reliability, Apple 10-1 to 10-4
relocatable program 4-11, gl-6
repair, Apple 10-4 to 10-10
RESET' 2-6, 4-3, 4-4, 4-15 to 4-16
  and Autostart ROM 4-15, 6-16 to 6-17, 9-11
  and disk controller 9-12, 9-13
  and peripheral slots 1-4, 7-19, 7-20
  and RAM card 5-27 to 5-31
  and 6502 4-3, 4-4, 4-15
  handler 4-5
  hard vector 4-15
  and Monitor ROM 4-15
  power-up (see power-up reset)
  priority 4-19
  soft (RAM) vector 4-15 to 4-16
revisions, motherboard G-1 to G-3
  Revision 1 7-15, 8-5, 8-9 to 8-12, 8-14, 8-19, 8-29
  Revision 7 5-22, 7-21, 8-5, 8-11, 8-16, 8-30
  RFI Revision 4-5, 8-11, 8-15
  (also see schematics)
RF leakage 8-3, G-1, G-3
RF modulator 1-5, 8-3, 8-33
Rockwell International 4-1, 4-7 to 4-8, 4-14, C-1 to C-7, I-3
ROM (Read Only Memory) 1-3 to 1-4, 6-1 to 6-21, FO1
  and BASIC 1-1 to 1-4, 2-6, 6-1, 6-5 to 6-6
  and bus fights 6-2 to 6-4
  and bus structure 2-3 to 2-13, FO1
  and I/O SELECT' 7-21
  and monitor 1-4, 2-6, 6-5
  and R/W 6-2, I-3
  and 6502 memory usage 4-5 to 4-6
  Bootstrap (P5) 9-1, 9-10 to 9-12

chip selects 6-1, 6-3, 6-13, 6-21
connections 6-2 to 6-4, FO1
firmware 1-1 to 1-4, 2-6, 6-4 to 6-6, 9-11
inhibiting 6-2 to 6-4, 7-20
on RAM card 5-31
ROM bus 2-3
seventh ROM 6-2 to 6-4, 7-21
write cycle (just kidding)
2316/9316 ROM 6-1 to 6-3, 6-6 to 6-8
2716 EPROM 6-13 to 6-15
6309 PROM 9-22 to 9-23
(also see Autostart ROM; Firmware card; F8 ROM; Monitor ROM)
ROM ENABLE'
firmware card 6-9 to 6-11
I/O SELECT' 7-21
I/O STROBE' 6-2 to 6-4, 7-21
motherboard 6-2, 6-3, 7-2, 7-3
RWTS 9-4
data formats 9-24 to 9-28
flowchart 9-36
RWTS, programming examples from 9-34 to 9-42
checksum 9-26, 9-42
write protect check 9-17 to 9-22
Write Table 9-26, 9-27

scan counter (see video scanner)
schematic diagrams (see Schematic Diagrams section)
Scott, Mike vi, I-3
screen display 1-5 to 1-7
mapping 8-1 to 8-2
memory display areas 1-5
memory maps 5-10 to 5-19, 5-37
modes 1-5 to 1-7
pages 1-5 to 1-7
soft switches 1-7, 7-2 to 7-6
(also see memory; video)
secondary buses 2-9 to 2-12, FO1
serial data transfer 1-8, gl-6
serial I/O 1-8, 2-12, 7-2 to 7-12, FO1 (also see I/O)
serial input multiplexor 2-7 to 2-12, 7-2 to 7-12, FO1
serrations 8-3 to 8-5, 8-11, G-2, G-3
SET OVERFLOW' 4-4
seventh ROM (expansion ROM) 6-2 to 6-4, 7-21
(also see I/O STROBE')
SHIFT key modification 7-31, 7-36 to 7-38, 10-5
short circuit 10-6 to 10-7
Shugart 9-3, 9-17, I-1
Siemens 9-45
SIMULREAD 7-25 to 7-26
simultaneous DMA 2-11, 4-13
Soft CTRL Systems 6-14
soft reset 4-15 to 4-16, 10-3
RAM card 5-30
soft switches
disk controller 9-11 to 9-14
screen mode 1-7, 7-2 to 7-6
SOFT 5 3-14, 3-15, G-3
SOFTALK 5-36, 6-13, 9-3
Software Applications
Apple timing loops 3-18
Aspect ratio in Apple Display 8-28
Modifying the system monitor 6-16 to 6-17
Programming the game paddles 7-24 to 7-27
Reading video data from a program 5-36 to 5-41
Switching screen modes in timed loops 3-16 to 3-17
6502 addressing details 4-20 to 4-23
software interrupt 4-15 (also see BREAK)
Solid State Sales 5-33
source program 4-11 to 4-12, gl-6
speaker 1-8, 7-1 to 7-6, 7-10, 7-12
installing volume control 7-39 to 7-40
programming 4-20, 7-1 to 7-2, 7-12
special function keys 1-7, 7-13 to 7-18
stack, 6502 4-5, 4-15 to 4-18
stack pointer 4-5, 4-10 to 4-11, 4-15
stacked interrupts 4-17
state machine (see logic state sequencer)
status register, 6502 4-10 to 4-11, 4-16 to 4-18
STEP utility 6-6, 6-16, 6-18
stepper motor 9-2 to 9-7
strobe gl-6
CAS' 3-3 to 3-10, 5-2 to 5-6, 5-20 to 5-35
C040 STROBE' 1-8, 2-12, 7-2 to 7-3, FO1
I/O STROBE' 6-2 to 6-4, 7-2 to 7-9, 7-19 to 7-21
keyboard STROBE 7-13 to 7-17
keyboard strobe flip-flop 7-14 to 7-15
RAS' 3-3 to 3-10, 5-2 to 5-6, 5-23 to 5-32
SWEET 16 6-5, 6-6
switch bounce 6-18, 7-13
SYNC, video 8-3, 8-9 to 8-13
SYNC, 6502 4-4, 4-32
Synertek 4-1, 4-7 to 4-8, 4-14, 5-5, 6-1, C-1, C-7, I-3
Synertek Programming Manual 4-20

tables (see Appendix K)
television
aspect ratio 8-28
color killer 8-29
frequency interlace 8-6, 8-34
frequency response 8-6, 8-19 to 8-20, 8-33 to 8-34
input 1-5, 8-3
processing 8-3 to 8-7, 8-33 to 8-34
scan interlace 3-12, 8-5, gl-4
scanning 3-10, 3-11, 8-3 to 8-5, 8-10, 8-14
sync 3-10, 8-3 to 8-5
(also see video)
temperature, operating 10-3 to 10-4
Texas Instruments 6-13
TEXT mode 1-5 to 1-7, 8-16 to 8-17
ASCII 1-6, 8-9, 8-30 to 8-32
characters 8-9
eliminating colored shadows 8-29
memory scanning 5-7 to 5-13, 5-36 to 5-38
norm/inv/flash 1-6, 8-7 to 8-8, 8-12 to 8-13, 8-17, 8-30 to 8-32
size ratio 8-28
video generation 8-8, 8-12, 8-15 to 8-17
time constant 7-4, 7-33
timers
disk controller 9-12, 9-13
fixed timing reference 7-33
paddles 1-8, 7-2 to 7-11, 7-24 to 7-27, FO1
power-up reset 7-14, 7-15
text flasher 8-12, 8-13, 8-17
timing diagrams and descriptions
CAS' gating 5-22
disk controller command decoder switching 9-22 to 9-23
disk read pulse generation 9-9 to 9-10
DMA 4-13 to 4-14
firmware card 6-11 to 6-12
HIRES interference patterns 8-20 to 8-22
I/O 7-8 to 7-9
LORES phase relationships 8-23 to 8-25
MIXED mode switching 8-25 to 8-27
RAM read cycle 5-22 to 5-24
RAM timing signals 5-2
RAM write cycle 5-24 to 5-25
read cycle with no data response 5-25 to 5-26
READ sequence performance 9-32 to 9-33
ROM 6-6 to 6-8
timing generator signals 3-4 to 3-10

video output 8-15 to 8-25
WRITE sequence initialization 9-22 to 9-23
6502 4-6 to 4-10, C-1 to C-8
timing generator 3-1 to 3-15, FO1
and video scanner 3-2
hardware 3-12 to 3-15, FO2
overview 3-2 to 3-3
propagation delay 3-4 to 3-7
repair 10-9
signal descriptions 3-6 to 3-10
signal distribution 3-8
signal frequencies 3-3 to 3-4
timing diagrams 3-4 to 3-6
(also see long cycle; timing diagrams)
timing loop 3-16 to 3-18, 4-27, 9-24, 9-27
toggle outputs 2-7, 2-12, 7-2 to 7-4, 7-10, 7-12
TRACE utility 6-6, 6-16, 6-18
trademarks B-1
transceiver (transmitter/receiver) 2-2 to 2-3, 4-3, 4-5
(also see bidirectional bus driver)
tri-state bus drivers 2-1 to 2-3, 4-2 to 4-5
tri-state logic 2-1 to 2-3, gl-7, E-3
troubleshooting 10-6 to 10-10, gl-7
truth tables E-1 to E-4
TTL (Transistor Transistor Logic) 1-8, gl-7, E-3
LSTTL 1-8, 3-15, 5-42, 7-11, gl-4
STTL 3-15, 5-42
TTL Data Book 5-20
two state logic 1-3, 1-8, 2-1

UNDERLINE program 5-38 to 5-41
UNUSED 8 5-8, 5-10, 5-14, 5-19, 5-36 to 5-38
upgrading to 48K RAM 5-32 to 5-34
USER1 7-6, 7-8, 7-19, 7-20

VBL (Vertical BLanking) 8-4 to 8-5, 8-9 to 8-14
and memory scanning 5-11, 5-13, 5-18, 5-19, 5-36 to 5-41
vectors, interrupt 4-15 to 4-18
vertical counter 3-11 to 3-15
vertical retrace 3-12, 5-13, 8-5, 8-10
vertical scan 3-10 to 3-12, 8-4 to 8-5, 8-10
vertical sync 3-10 to 3-12, 5-11, 5-18, 8-3 to 8-5, 8-9 to 8-14
video
and EURAPPLE 1-5, 3-12, 3-14, 8-12 to 8-14
and FCC 8-3, G-1, G-3
and NTSC 8-3 to 8-6, 8-14
and RF modulator 1-5, 8-3, 8-33
aspect ratio 8-28
black reference 8-3
blanking 3-10 to 3-12, 8-3 to 8-5, 8-28
color burst killing 8-11 to 8-13, 8-29, G-2 to G-3
color signals 8-5 to 8-7, 8-15, 8-18 to 8-25, 8-33 to 8-34
colors 1-6, 8-6 to 8-7, 8-18 to 8-25
composite video 8-3, 8-6, gl-2
display (see screen display)
generation (see video generator)
HIRES graphics (see HIRES)
horizontal PERIOD 8-9 to 8-14
LORES graphics (see LORES)
mapping (see screen mapping)
MIXED mode (see MIXED)
modes 1-5 to 1-7
monitor 8-3, 8-6, 8-17, 8-19 to 8-20
programming 1-5 to 1-6, 8-1 to 8-2
retrace 3-11 to 3-12, 8-4 to 8-5, 8-10
scanning (see memory scanning, television scanning)
soft switches 1-7, 7-2 to 7-6
syncing serrations 8-3 to 8-5, 8-11, G-2, G-3
TEXT mode (see TEXT)
unwanted spike 8-4, 8-5, 8-11
(also see HIRES; LORES; MIXED; screen; television; TEXT)
video data
and peripheral cards 3-23 to 3-26, 5-5
distribution 2-12, 5-5, FO1
reading from program 5-36 to 5-41
(also see RAM latched data)
video generator 8-1 to 8-34, FO1
and Eurapple scanning 8-12 to 8-14
and long cycle 8-16
color burst killer 8-11 to 8-13, 8-29, G-2 to G-3
delayed HIRES 1-6, 8-18 to 8-22
GRAPHICS shifter 8-7, 8-18, 8-23
HIRES generation 8-18 to 8-19
HIRES interference 8-20 to 8-22
LORES cyclical patterns 8-23 to 8-25
LORES generation 8-22 to 8-25
MIXED mode switching 5-5, 8-11 to 8-14, 8-25 to 8-27
mode configuration 8-18, 8-22 to 8-23
norm/inv/flash 8-7 to 8-8, 8-17, 8-30 to 8-32
PICTURE signal generation 8-7
PICTURE signal multiplexor 8-7
revisions 8-5, 8-9 to 8-12, 8-14, 8-16, 8-19, 8-29
TEXT generation 8-7 to 8-9, 8-16 to 8-17
TEXT ROM 8-7 to 8-9, 8-16 to 8-17, 8-30 to 8-32, I-2
TEXT shifter 8-7
timing diagram 8-16
timing signals 8-14 to 8-15
video scanner gating 8-7 to 8-14
video scanner 1-5, 2-11, 3-10 to 3-15, FO1
and Eurapple 3-12, 3-14, 8-12 to 8-14
feedback to video generator 3-2
hardware 3-12, 3-14, FO2
signal distribution 2-12, FO1
(also see video generator)
video signals
chrominance 8-5 to 8-6, 8-33 to 8-34
COLOR BURST 3-9, 8-3 to 8-6, 8-8 to 8-13
COLOR REFERENCE 3-3 to 3-9, 8-5, 8-15, 8-18 to 8-25
HBL 8-3 to 8-5, 8-9 to 8-13
HIRES TIME 8-12, 8-26
luminance 8-5 to 8-6, 8-33 to 8-34
PICTURE signal 8-3, 8-6 to 8-8, 8-12, 8-15
SYNC 8-3, 8-9 to 8-13
VBL 8-4 to 8-5, 8-9 to 8-14
VIDEO output 8-2 to 8-6, 8-8, 8-12
VIDEO BLANKING 8-3 to 8-5, 8-9 to 8-13, 8-21 to 8-22, 8-26
voltage gl-7
(also see logic levels; power supply)
volume control installation 7-39 to 7-40

Waller, Eric 10-10
Wiggington, Randy 9-17, H-2, I-4
Wilbur, Roger 10-10
wire-OR (collector OR) 4-5, gl-7
Word Power 6-14
Worth, Don 9-3, 9-31
Wozniak, Steve vi, 4-14, 5-12, 6-5, 7-8, 9-17, 9-26, H-1 to H-2, I-1 to I-4
write cycle 2-3 to 2-5, 5-24 to 5-25, 7-8 to 7-9
write protect switch 9-6, 9-8 to 9-9
installing on disk drive 9-43 to 9-45
(see also disk I/O)

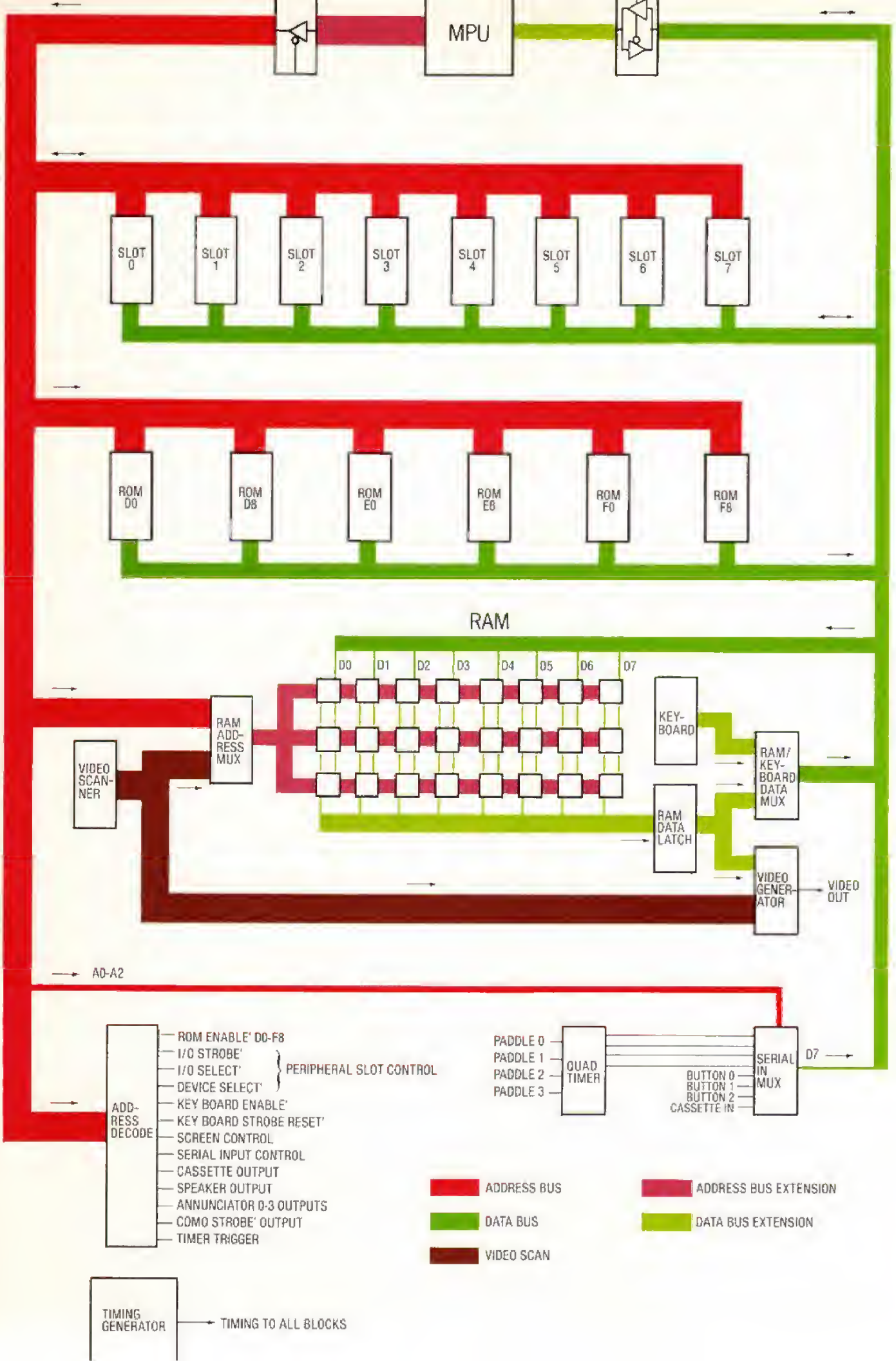
X-register 4-10 to 4-11

Y-register 4-10 to 4-11

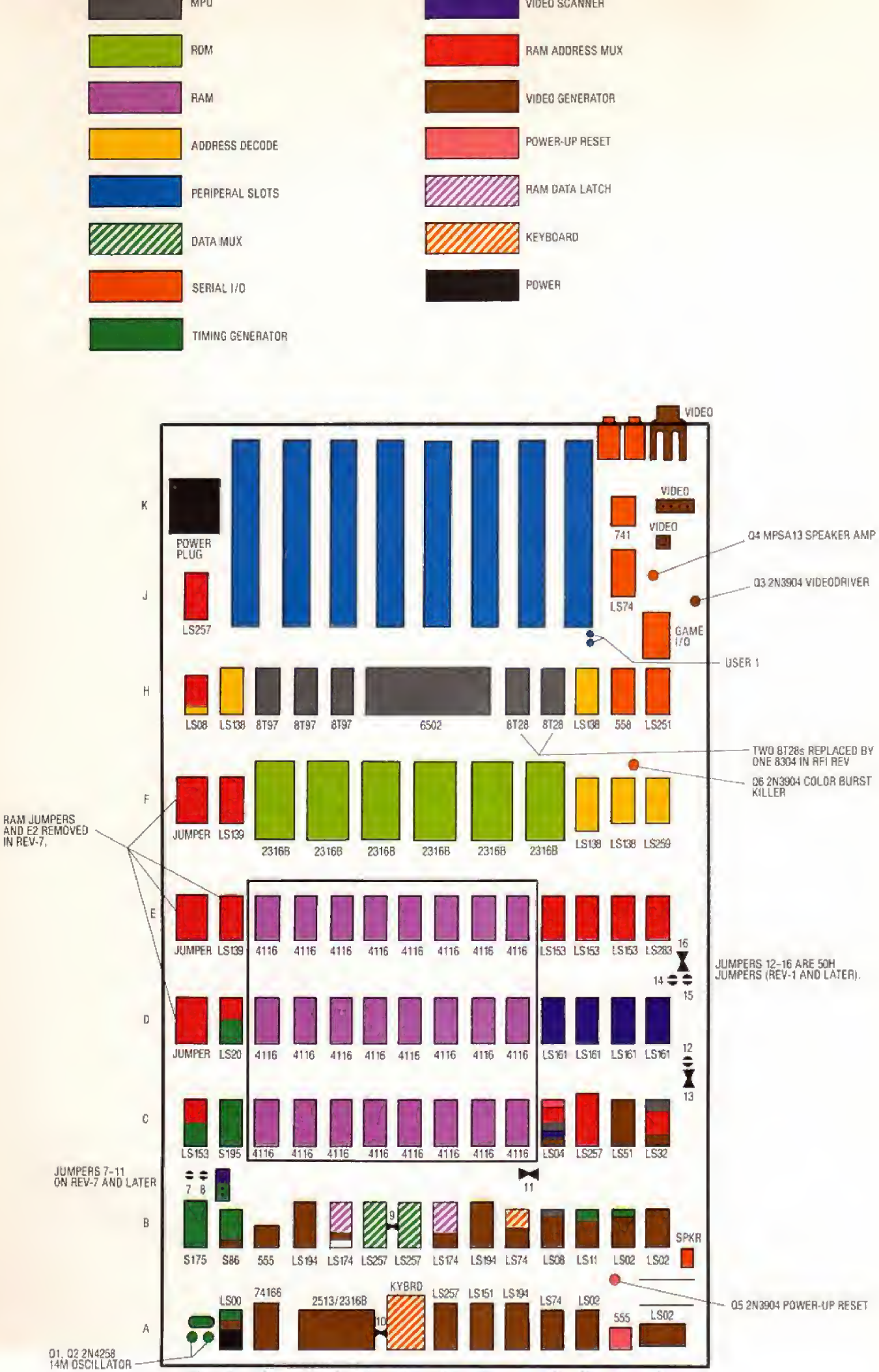
zero page addressing mode 4-5
Zilog 4-14, B-1
Z80 MPU 4-1, 4-13, 5-26, I-3
Z80 softcard 4-14, 4-26, I-3

6502 MPU (see MPU)

The Apple II Bus Structure


MPU
SLOT 0
SLOT 1
SLOT 2
SLOT 3
SLOT 4
SLOT 5
SLOT 6
SLOT 7
ROM D0
ROM D8
ROM E0
ROM E8
ROM F0
ROM F8
RAM
D0
D1
D2
D3
D4
D5
D6
D7
RAM ADD-RESS MUX
VIDEO SCAN-NER
KEY-BOARD
RAM DATA LATCH
RAM/ KEY-BOARD DATA MUX
VIDEO GENER-ATOR
VIDEO OUT
A0-A2
ADD-RESS DECODE
ROM ENABLE' D0-F8
I/O STROBE'
I/O SELECT'
DEVICE SELECT'
PERIPHERAL SLOT CONTROL
KEY BOARD ENABLE'
KEY BOARD STROBE RESET'
SCREEN CONTROL
SERIAL INPUT CONTROL
CASSETTE OUTPUT
SPEAKER OUTPUT
ANNUNCIATOR 0-3 OUTPUTS
COMO STROBE' OUTPUT
TIMER TRIGGER
PADDLE 0
PADDLE 1
PADDLE 2
PADDLE 3
QUAD TIMER
BUTTON 0
BUTTON 1
BUTTON 2
CASSETTE IN
SERIAL IN MUX
D7
ADDRESS BUS
DATA BUS
VIDEO SCAN
ADDRESS BUS EXTENSION
DATA BUS EXTENSION
TIMING GENERATOR
TIMING TO ALL BLOCKS

Motherboard Component Locations


MPU
ROM
RAM
ADDRESS DECODE
PERIPERAL SLOTS
DATA MUX
SERIAL I/O
TIMING GENERATOR
VIDEO SCANNER
RAM ADDRESS MUX
VIDEO GENERATOR
POWER-UP RESET
RAM DATA LATCH
KEYBOARD
POWER
VIDEO
POWER PLUG
LS257
741
LS74
GAME I/O
Q4 MPSA13 SPEAKER AMP
Q3 2N3904 VIDEODRIVER
USER 1
LS08 LS138 8T97 8T97 8T97 6502 8T28 8T28 LS138 558 LS251
TWO 8T28s REPLACED BY ONE 8304 IN RFI REV
Q6 2N3904 COLOR BURST KILLER
RAM JUMPERS AND E2 REMOVED IN REV-7.
JUMPER LS139 2316B 2316B 2316B 2316B 2316B 2316B LS138 LS138 LS259
JUMPER LS139 4116 LS153 LS153 LS153 LS283
JUMPERS 12-16 ARE 50H JUMPERS (REV-1 AND LATER).
JUMPER LS20 LS161 LS161 LS161 LS161
LS153 S195 LS04 LS257 LS51 LS32
JUMPERS 7-11 ON REV-7 AND LATER
S175 S86 555 LS194 LS174 LS257 LS257 LS174 LS194 LS74 LS08 LS11 LS02 LS02
SPKR
LS00 74166 2513/2316B KYBRD LS257 LS151 LS194 LS74 LS02 555 LS02
Q5 2N3904 POWER-UP RESET
Q1, Q2 2N4258 14M OSCILLATOR