MICROCOMPUTER EXPERIMENTATION WITH THE MOS TECHNOLOGY KIM-1

LANCE A. LEVENTHAL

MICROCOMPUTER EXPERIMENTATION WITH THE MOS TECHNOLOGY KIM-1

LANCE A. LEVENTHAL

Emulative Systems Company
San Diego, California

PRENTICE-HALL, INC., *Englewood Cliffs, New Jersey 07632*

Library of Congress Cataloging in Publication Data

Leventhal, Lance A., 1945-
Microcomputer experimentation with the MOS technology KIM-1.

Includes bibliographies and index.
1. KIM-1 (Computer)—Laboratory manuals. I. Title.
QA76.8.K15L48 001.64'04 81-19249
ISBN 0-13-580779-4 AACR2

Editorial/production supervision: Barbara Bernstein
Interior design: Virginia Huebner
Manufacturing buyer: Joyce Levatino
Cover design: Frederick Charles, Ltd.

This book is dedicated to Karl Amatneek, who first introduced me to the KIM.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-580779-4

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Contents

Preface

The aim of this manual is to provide experimental training in the use of microcomputers for students of engineering, engineering technology, computer science, the physical sciences, the health sciences, and related fields. The emphasis throughout is on the design of controllers for industrial and laboratory applications. The experiments, examples, and problems were adapted from applications in instrumentation, test equipment, communications, computers and peripherals, industrial control, process control, business equipment, aerospace and military systems, and consumer products. The manual illustrates the use of the microcomputer in performing tasks that are essential in all of these applications—responding to switches, controlling displays, encoding and decoding data, collecting and processing data, executing arithmetic functions, interfacing simple handshaking peripherals (such as terminals and printers), timing and scheduling operations, and implementing serial communications.

First, the manual describes the operation of the microcomputer and then introduces assembly language programming, shows how to perform simple controller functions, discusses hardware/software tradeoffs, describes the design and development of programs, demonstrates alternative approaches to input/output and timing, presents the advantages and uses of programmable LSI devices, and describes communications methods. The final experiment provides a brief introduction to hardware design and development. Included are numerous examples drawn from actual applica-

casual users can find programs of specific interest to them and students can readily find material for review, reference, or further information.

Each experiment in the manual is itself self-contained. Each includes a list of goals, definitions of new terms, references (with page numbers), descriptions of instructions that are being introduced, a list of required equipment (with diagrams), and a key point summary. Each laboratory exercise contains numerous problems that are linked closely to the discussion. The problems illustrate points made in the discussion and extend that discussion in areas that are important in actual applications; there are no "make-work" problems or rote tasks. I have tested all the problems and have provided sample data, hints, and discussions.

I would like to give a special note of acknowledgment to two monthly magazines that specialize in microcomputers based on the 6502 microprocessor. Both are excellent sources for articles about the KIM-1 and related computers and both feature advertisements from suppliers of KIMs and accessories, news about clubs and users' groups, and reviews of books, hardware, software, and other materials. One of these magazines is *Micro*, published by Robert Tripp (P.O. Box 6502, Chelmsford, MA 01824); the other is *Compute*, published by Robert Lock (P.O. Box 5406, Greensboro, NC 27403). I have enjoyed reading both magazines and recommend both to the users of this manual.

Many people contributed to the writing of this manual. Irvin Stafford of Burroughs Corporation constructed the hardware, checked the examples and problems, and suggested many improvements and corrections. Carter Stafford provided the photographs of the KIM-1 and the other laboratory apparatus. Others who helped include Gary Hankins of Sorrento Valley Associates and Winthrop Saville of Sunrise Software. Michael Tomczyk of Commodore Business Machines provided both encouragement and materials. Special thanks are due to Charles Peddle, the originator of the KIM.

The reviewers of the first volume in this series, all of them anonymous except for Sol Libes of Union County Technical Institute (New Jersey), provided many useful suggestions. My editor, Bernard Goodwin, encouraged this project as did Stephen Cline, Karl Karlstrom, and Walter Welch of Prentice-Hall. My wife, Donna, and my daughter, Elizabeth (Amanda Catherine), were patient and understanding, particularly as this project neared completion. Of course, I am responsible for all remaining errors and I would appreciate the users of this manual taking the time to inform me of any errors that they find or suggestions that they may have for improvements.

LANCE A. LEVENTHAL

San Diego, California

Laboratory 0

Basic Operations

PURPOSE

To learn how to operate the microcomputer.

PARTS REQUIRED

A KIM-1 microcomputer with a 5-V power supply.

REFERENCE MATERIALS

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 11-12 (hexadecimal number system), 70-72 (semiconductor memories—types), 92-94 (basic computer system organization).

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 2 and 3.

WHAT YOU SHOULD LEARN

1) How to reset the computer.
2) How to examine the contents of a memory location.

3) How to change the contents of a memory location.
4) How to enter and execute a simple program.

TERMS

Byte—the smallest grouping of bits that a computer can process at one time, usually consists of 8 bits.

Central processing unit (CPU)—the control section of the computer; the part that controls its operations, fetches and executes instructions, and performs arithmetic and logical functions.

Hexadecimal (or hex)—number system with base 16. The digits are the decimal numbers 0 through 9, followed by the letters A through F (see Table 0-1).

Microcomputer—a computer that has a microprocessor as its central processing unit.

Microprocessor—a complete central processing unit for a computer constructed from one or a few integrated circuits.

Monitor—a program that allows the computer user to enter programs and data, run programs, examine the contents of the computer's memory and registers, and utilize the computer's peripherals.

Nonvolatile memory—a memory that retains its contents when power is removed.

Read-only memory (ROM)—a memory that can be read but not changed in normal operation.

Read/write (random-access) memory (RAM)—a memory that can be both read and changed (written) in normal operation.

Reset—a control signal that forces the computer into a known initial (or startup) state.

Scratchpad—an area of memory that is especially easy and quick to use for storing variable data or intermediate results.

Volatile memory—a memory that loses its contents when power is removed.

Word—the basic grouping of bits that the computer can process at one time. The 6502 microprocessor has an 8-bit word.

6502 INSTRUCTIONS

BRK (00 hex)—force break; on the KIM microcomputer, this instruction transfers control to the address the user stores in memory locations 17FE and 17FF (hex).

OVERVIEW

The MOS Technology KIM-1 (or Keyboard Input Monitor 1; Figure 0-1) is an inexpensive microcomputer based on the widely used MOS Technology 6502 microprocessor. Chapter 2 of the *KIM-1 User Manual* describes the initial unpacking and setup of the microcomputer; no assembly is necessary, other than the attachment of a connector and a power supply (see pp. 6-9 of the *User Manual*). The microcomputer consists of the following components:

- A 6502 microprocessor, the central processing unit or "brain."
- Read/write memory or RAM (eight 6102 devices into which the user can enter data and programs). Each 6102 RAM contains 1K 1-bit words (1K = 2^{10} = 1024). The eight 6102s connected in parallel contain 1K 8-bit words (or bytes).
- Two 6530 Peripheral Interface/Memory devices. Each of these devices contains 1K 8-bit words of read-only memory or ROM, 64 8-bit words of RAM, two 8-bit input/output (I/O) ports, and an 8-bit timer. The read-only memory contains an operating program which we will refer to as the KIM monitor. The

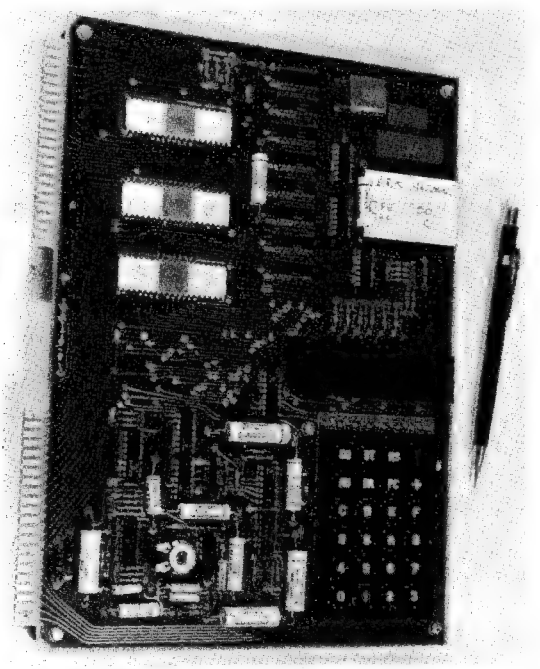


FIGURE 0-1. The KIM-1 microcomputer. (Photo courtesy of Carter Stafford.)

I/O ports on one 6530 device are used to interface the KIM's keyboard and display; the I/O ports on the other 6530 are left available for user-defined input or output.

- A keyboard with 23 keys and one slide switch. The 16 keys at the bottom are used to enter hexadecimal digits. The 7 keys at the top and the slide switch are used to operate the microcomputer. Not all KIMs have the slide switch in the same position; some have it in the leftmost column, whereas others have it in the rightmost column. We will allow for either possibility in our discussion.
- Six-digit seven-segment LED display.
- An audio cassette interface so that the user can record programs and data on cassettes and load them from cassettes. The *KIM-1 User Manual* explains how to do this on pp. 12-16; we will not discuss cassettes again, but they let the user save completed or partially completed programs and thus reduce the amount of repetitive keyboard input that is required.
- A teletypewriter interface that lets the user attach a standard teletypewriter or terminal.

The *MCS6500 Microcomputer Hardware Manual* (Commodore/MOS Technology, Norristown, PA, 1975) contains complete descriptions of the various devices. Appendix 3 contains a summary of those descriptions.

RESETTING THE COMPUTER

To start using the KIM, you must reset it. The reset key (marked RS) is the rightmost key in the top row. Press and release the RS key. The LED display should light and show something in all six digits; unfortunately, the KIM does not provide a special prompting message, so exactly what you see will vary.

The microcomputer is now executing the KIM monitor program stored in the ROM on the 6530 devices. This program allows you to control the microcomputer from the keyboard. You can place programs and data in read/write memory, execute programs, examine and change the contents of memory and registers, and perform other functions which we will describe later.

EXAMINING MEMORY

The basic KIM microcomputer contains 1K bytes of read/write memory which occupy addresses 0000 through 03FF hexadecimal, and 128 bytes of read/write memory (in the 6530 devices) which occupy addresses

17C0 through 17FF hexadecimal. The KIM monitor (in ROM) occupies memory addresses 1800 through 1FFF hexadecimal. Some of the RAM addresses are either special or reserved, so we will either not use them or leave them for purposes that we will discuss later. In particular, we should note the following (see Figures A5-1 and A5-2):

- The KIM monitor uses addresses 00EF through 00FF and 17E7 through 17FF for its own purposes. We will not use those addresses, since we would interfere with the monitor and it would probably change whatever we put there anyway.
- The 6502 microprocessor uses addresses 0100 through 01FF for its stack. We will discuss the stack in Laboratory A, but for now we will simply not use those addresses.
- Addresses 0000 through 00FF (note that the KIM monitor uses some of these) are generally reserved as a scratchpad. We will see why this is done in Laboratory 1. We will use addresses 0000 through 00EE as a data area only; we will not place any programs there.

Overall, we will utilize the following addresses:

- 0000 through 00EE for data
- 0100 through 01FF for the stack
- 0200 through 03FF for programs and data
- 1780 through 17E6 for data

Note that each memory location has a 16-bit address (four hexadecimal digits) and contains 8 bits of data (two hexadecimal digits). Table 0-1 is a list of the hexadecimal digits and their binary and decimal equivalents. Use this table if you need help converting numbers to and from the hexadecimal representation.

Important Note: Before you proceed, be sure that the slide switch in the top row of the keyboard is in its off position (the on position is marked). We will explain the use of this switch in Laboratory 8.

To examine the contents of memory, you must first press the Enter Address Mode or AD key (second row from the top, leftmost column). You must then enter the four-digit address of the memory location that you want to examine. Remember that the digits are hexadecimal (see Table 0-1) and note that the digits B and D are shown as lowercase letters (b and d, respectively) because of the limitations of the inexpensive calculator-like displays. If you get lost or confused, start over again by pressing the RS (reset) key.

Table 0-1

HEXADECIMAL-TO-DECIMAL CONVERSION TABLE

HEXADECIMAL DIGIT	DECIMAL VALUE	BINARY VALUE
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B or b	11	1011
C	12	1100
D or d	13	1101
E	14	1110
F	15	1111

For example, enter the four-digit address 0, 2, 0, 0. Note that the digits appear on the fourth display from the left and move to the left. This is more obvious if you enter an address in which all the digits are different, such as 0, 2, 3, 5. The two rightmost displays may change value. Remember that all the displays are in hexadecimal and that addresses (shown on the leftmost displays) are four digits long, whereas data entries (shown on the rightmost displays) are two digits long.

In general, the two rightmost displays show the contents of the address on the four leftmost displays. If the address is in RAM, the value is arbitrary, because RAM loses its contents when power is removed and could start in any state whatsoever. Such a memory is said to be *volatile*. To demonstrate this volatility, simply unplug the KIM's power supply and repeat the examination procedure. Try this several times if you are not easily convinced.

The following procedure thus allows you to examine the contents of a memory location:

1) (if necessary) Reset the computer with the RS key.
2) Press the Enter Address Mode (AD) key.
3) Enter the address as four hexadecimal digits starting with the most significant digit.

4) Observe the contents of the location on the two rightmost digits of the display.

If you enter the address incorrectly, simply keep entering more digits until you get it right. Note that each digit you enter appears on the fourth display from the left and moves to the left as you enter more digits. If you get totally confused, press RS and start over.

PROBLEM 0-1

Examine the contents of memory location 0038 (hex).

PROBLEM 0-2

Examine the contents of memory location 1CA2 (hex). Its value should be F0. Disconnect the power supply and examine this location again. The result will be the same, since this memory location is in the *nonvolatile* read-only memory.

Note the following special features of the KIM displays:

- The digits B and D are shown as lowercase letters (b and d, respectively).
- The digit 6 appears with a bar at the top so that you can differentiate it from "b."

Be careful; these special features can lead to errors until you get used to them.

Once you have examined a memory location, you can examine the next higher address by pressing the Address Increment (+) key (second row from the top, rightmost column). Try examining memory locations 0200 (hex) through 0210 (hex). Note that you can go forward but not backward. Note also the sequence of the hexadecimal digits (remember Table 0-1).

CHANGING MEMORY

Once you have examined a memory location, you can change its contents by pressing the Enter Data Mode (DA) key (second row from the top, second column from the left) and then entering two digits. For example, to change the contents of memory location 0200 to 6F, first examine that location and then press

DA	(displays do not change)
6	(rightmost display now contains a 6)
F	(rightmost displays now contain 6F)

To verify that the data is there, reset the computer and repeat the examination procedure. Note that in the data mode the digits start on the rightmost display and move left, but do not affect the address.

So the following procedure allows you to change the contents of a memory location (after examining it):

5) Press the Enter Data Mode (DA) key.
6) Enter the data as two hexadecimal digits, starting with the more significant digit.

If you enter the data incorrectly, simply enter more digits until you get it right. Each digit you enter will appear on the rightmost display and move left as you enter more digits. If you want to proceed to the next higher address, press the Address Increment (+) key. This key adds 1 to the address but does not change the mode; you do not have to press the DA key again to change the next location. Of course, if would be nice if the KIM provided an Address Decrement key so that you could correct mistakes. An obvious problem with having two modes is that you always end up in the wrong one (Murphy's Law applies). Unfortunately, the KIM does not indicate which mode it is in; the only way you can find out is by pressing a digit key and seeing where the digit appears.

PROBLEM 0-3

Enter the following data into memory locations 0200 through 0202:

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)
0200	A9
0201	6A
0202	00

Verify the values after you enter them.

PROBLEM 0-4

Try changing the contents of memory location 1CA2 to A9 (hex). Do the rightmost displays change? What do you find when you examine the location again? Remember that this address is in the read-only memory, not in the read/write memory.

EXECUTING A PROGRAM

To execute a program, return to the address mode by pressing the AD key, enter the four-digit address at which you want the computer to start, and then press the GO key (the leftmost key in the top row). A simple program consists of the single instruction BRK (FORCE BREAK or TRAP), which causes the computer to transfer control to an address you specify.

You can select BRK's destination (generally called its *return address*) by loading that address into memory locations 17FE and 17FF. The only special thing to remember is that you must load the address upside down, since this is the way the 6502 microprocessor expects it. That is, you will have to load the less significant half of the return address into memory location 17FE and the more significant half into memory location 17FF. Although this convention seems awkward, it is used by many manufacturers, including Intel and Digital Equipment.

The return address we will use is 1C00, the normal entry point for the KIM monitor. You can place this address in the assigned locations as follows:

1) Press RS to reset the computer.
2) Press AD to enter the address mode.
3) Press 1, 7, F, E to examine memory location 17FE (hex).
4) Press DA to enter the data mode.
5) Press 0,0 to enter the less significant half of 1C00 into memory location 17FE.
6) Press + to examine memory location 17FF.
7) Press 1,C to enter the more significant half of 1C00 into memory location 17FF.

We can now enter and execute our simple program as follows:

1) Press AD to enter the address mode.
2) Press 0, 2, 0, 0 to access memory address 0200. This is the location in which we will place the BRK instruction.
3) Press DA to enter the data mode.
4) Press 0,0. This is the hexadecimal code for the BRK instruction; you can look it up on your programming card or in Table A1-1 of this book.
5) Press GO.

Since we never left address 0200, we do not have to enter it before pressing GO. What happens?

All the computer does is display address 0202 and its contents. We have no way of knowing whether anything actually happened, except that the computer did not wander off aimlessly.

PROBLEM 0-5

Enter and execute the same program in memory location 022A.

PROBLEM 0-6

Enter and execute the same program in memory location 1CA2. What happens and why?

KEY POINT SUMMARY

1) The KIM-1 microcomputer has a monitor program stored in read-only memory (ROM) in addresses 1800 through 1FFF. This memory is nonvolatile and the user cannot change it.

2) You can reset the computer and transfer control to the KIM-1 monitor by pressing the RESET (RS) key.

3) The KIM-1 microcomputer has read/write memory (RAM) in addresses 0000 through 03FF and 1780 through 17FF. This memory is volatile (its contents change when power is lost) and the user can change it.

4) Some of the read/write memory is reserved for special purposes, either by the monitor or by the microprocessor. We will use memory addresses 0200 through 03FF for programs and data and addresses 0000 through 00EE and 1780 through 17E6 for data only.

5) Each memory location is characterized by a 16-bit address (four hexadecimal digits); its contents are an 8-bit number (two hexadecimal digits).

6) You can examine a KIM-1 memory location by pressing the Enter Address Mode (AD) key and entering its address. You can then change the contents by pressing the Enter Data Mode (DA) key and entering two hexadecimal digits as the new data. You can proceed to the next higher address without changing modes by pressing the Increment Address (+) key. This procedure allows you to see the contents of memory and to enter programs and data into memory.

7) You can have the KIM-1 microcomputer execute a program by pressing the Enter Address Mode (AD) key, entering the starting address, and pressing the GO key. All programs should end with a BRK

instruction (00 hex) so that control returns to the monitor when the program is finished. The user can specify the return address for BRK by placing it upside down in memory locations 17FE and 17FF. The normal value for the return address is 1C00 (00 in memory location 17FE, 1C in memory location 17FF), the entry point for the KIM monitor.

Laboratory 1

Writing and Running Simple Programs

PURPOSE

To learn how to write, load, and run simple programs.

PARTS REQUIRED

A KIM-1 microcomputer with a 5-V power supply.

REFERENCE MATERIALS

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 40-63, 72-104.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 4.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 18-19, 22-24, 27-35, 50-53, 64-68.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 95-99 (computer words), 99-101 (instruction words), 112-119 (hardware and software), 163-164 (program counter), 165-166 (accumulator), 310-311 (microprocessor instruction sets), 311-316 (6502 registers), 317-322 (6502 instruction set and addressing modes), 341 (memory/register transfers).

W. J. Weller, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapters 2 through 5.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 2 and 3.

WHAT YOU SHOULD LEARN

1) How to load programs into memory.
2) How to determine the length of instructions.
3) How to place addresses in instructions.
4) How to examine the results of programs.
5) How to make the computer perform simple processing operations.
6) How to examine registers.
7) How to change the contents of registers.

TERMS

Absolute addressing—an addressing mode in which the instruction contains the actual address required for its execution. In 6502 terminology, absolute addressing refers to a type of direct addressing in which the instruction contains a full 16-bit address occupying two bytes of memory.

Accumulator—a register that is the implied source of one operand and the destination of the result for most arithmetic and logical operations.

Addressing modes—the methods for specifying the addresses to be used in executing an instruction. Common addressing modes include direct, immediate, indexed, and relative.

Arithmetic shift—a shift operation that preserves the value of the sign bit (most significant bit). Be careful—the 6502's ARITHMETIC SHIFT LEFT instruction is actually a logical shift.

Assembler—a computer program that converts assembly language programs into a form (machine language) that the computer can execute directly. The assembler translates mnemonic operation codes and names into their numerical equivalents and assigns locations in memory to data and instructions.

Assembly language—a computer language in which the programmer can use mnemonic operation codes, labels, and names to refer to their numerical equivalents.

Comment—a section of a program that has no function other than documentation. Comments are neither translated nor executed; they are simply copied into the program listing.

Direct addressing—an addressing mode in which the instruction contains the address required for its execution. The 6502 processor has two types of direct addressing: zero page addressing (requiring only an 8-bit address on page zero) and absolute addressing (requiring a full 16-bit address in two bytes of memory).

Index register—a register that can be used to modify memory addresses.

Logical shift—a shift operation that moves zeros in at the end as the original data is shifted.

Low-level language—a language in which each instruction or statement is translated into a single machine language instruction.

Machine language—the programming language that the computer can execute directly with no translation other than numeric conversions.

Mnemonic—symbolic name for an instruction, register, or memory location that suggests its actual purpose or function.

Operation code (op code)—the part of an instruction that specifies the operation to be performed.

Page—a subdivision of the memory. In 6502 terminology, a page is a 256-byte section of memory in which all addresses have the same page number (i.e., the same 8 most significant bits). For example, page C6 consists of memory addresses C600 through C6FF.

Page number—the identifier that characterizes a particular page of memory. In 6502 terminology, the 8 most significant bits of a memory address.

Paged address—the identifier that characterizes a particular memory address on a known page. In 6502 terminology, the 8 least significant bits of a memory address.

Program counter—a register that contains the address of the next instruction to be fetched from memory.

Register—a storage location inside the CPU.

Stack—a section of memory that can be accessed only in a last-in, first-out manner. That is, data can be added to or removed from the stack only through its top; new data is placed above the old data and the removal of a data item makes the item below it the new top.

Stack pointer—a register that contains the address of the top of a stack.

Status register—a register that defines the current state of a computer, often contains various bits indicating internal conditions.

Zero page addressing—an addressing mode in which the instruction contains only the address on page zero that is required for its execution. That is, page zero is implied and need not be specified in the instruction. In 6502 terminology, zero page addressing refers to a type of direct addressing in which the instruction contains only an 8-bit address on page zero.

6502 INSTRUCTIONS

AND—logical AND; logically AND the contents of the accumulator with the contents of the specified memory location. The result is placed in the accumulator.

ASL—arithmetic shift left; shift each bit of the accumulator or a memory location left 1 position and clear the least significant bit (see Figure 1-1).

EOR—logical EXCLUSIVE OR; logically EXCLUSIVE OR the contents of the accumulator with the contents of the specified memory location.

LDA—load accumulator; load the accumulator from the specified memory address.

LDX—load index register X; load index register X from the specified memory address.

LDY—load index register Y; load index register Y from the specified memory address.

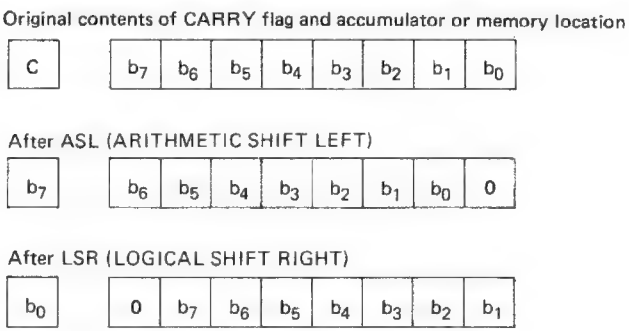


FIGURE 1-1. 6502 shift instructions ASL and LSR.

LSR—logical shift right; shift each bit of the accumulator or a memory location right 1 position and clear the most significant bit (see Figure 1-1).

ORA—logical (INCLUSIVE) OR; logically (INCLUSIVE) OR the contents of the accumulator with the contents of the specified memory location.

STA—store accumulator; store the accumulator in the specified memory address.

STX—store index register X; store index register X in the specified memory address.

STY—store index register Y; store index register Y in the specified memory address.

DATA TRANSFER PROGRAM

The first program that we will write simply moves the contents of memory location 0040 (hex) to memory location 0041. The computer here does exactly what we could do with a direct electrical connection between the two locations. Of course, the computer can provide such a connection between any of its memory locations. The program is

```
LDA     $40     ;GET DATA
STA     $41     ;MOVE IT
BRK             ;RETURN TO MONITOR
```

We are using the format of the MOS Technology assembler (see Figure 1-2), in which a $ before a number means "hexadecimal." The comments (separated from the rest of the line by semicolons) are intended solely for documentation and do not affect the program that the computer executes. Figure 1-3 is a programming model of the 6502 microprocessor, showing the registers that the programmer can use.

Let us now look at each instruction in detail:

1) LDA $40 loads the accumulator (or A register) with the contents of memory location 0040 (hex). The $ means hexadecimal and the leading zeros can be omitted as in common practice. Remember, the address is four hexadecimal digits (16 bits) long but the data stored at that address is two digits (8 bits) long.

2) STA $41 stores the contents of the accumulator in memory location 0041 (hex). Here again, the address is four digits long, whereas the data is two digits long.

Before a number:

$ - hexadecimal
% - binary
@ - octal

The default case (i.e., unmarked) is decimal.

Other symbols:

- immediate addressing
, - between a base address and the designation of an index register (X or Y)
' - before an ASCII character
; - before a comment

A space is required after a label and after an operation code. Parentheses around an address indicate that it is to be used indirectly (i.e., it contains an address rather than the actual data).

FIGURE 1-2. Format for the MOS Technology 6502 assembler.

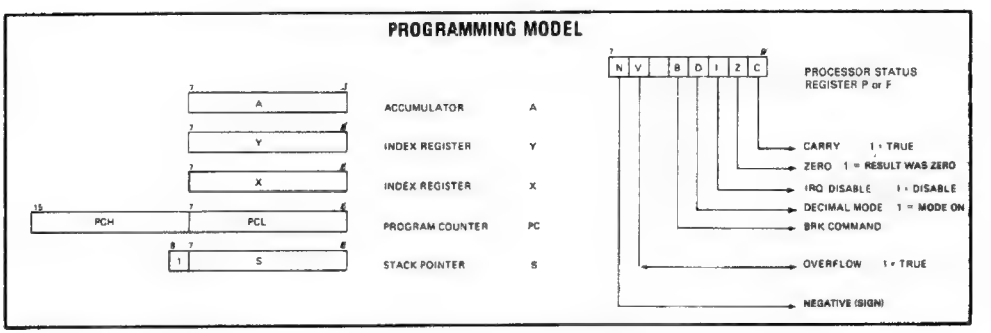


FIGURE 1-3. Programming model of the 6502 microprocessor.

3) BRK (FORCE BREAK) returns control to the KIM monitor (assuming that you have loaded 1C00 into addresses 17FE and 17FF). You should put BRK at the end of every program you write. The computer will then return to the monitor rather than wandering aimlessly after it has finished your program.

To enter the program into the computer's memory, you must look up the hexadecimal operation codes on the Instruction Set Summary or Programming Reference Card. Table A1-1 of this book contains a summary of the 6502 instruction set, but you will probably find the self-

contained card more convenient to use. Note that most instructions have several different operation (or op) codes, depending on which addressing mode you want. Each operation code is followed by two numbers: one (under N) indicates how many clock cycles the CPU needs to execute the instruction and the other (under #) indicates how many bytes of memory the instruction occupies. We will explain the various addressing modes as we use them.

Originally, we write the program in a form in which we can refer to the instructions by name. This form is called *assembly language*. However, the KIM-1 microcomputer does not allow us to enter names; it accepts only hexadecimal numbers. The form that involves only hexadecimal numbers is called *machine language*. Converting assembly language to machine language is a simple (but highly repetitive) matter of looking up operation codes on a card or in a table; we can do the conversion by hand or we can let the computer do this rote task by using a program called an *assembler*. Assembly language is much easier for the programmer to write than is machine language, since assembly language is based on meaningful names or *mnemonics* rather than on arbitrary numbers.

Program 1-1 is the hexadecimal (machine language) version of the data transfer program.

PROGRAM 1-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A5	LDA	$40
0201	40		
0202	85	STA	$41
0203	41		
0204	00	BRK	

Note the following:

1) We have used the zero page (direct) addressing mode for both LDA and STA; the byte of memory following the operation code contains the address to be used in the instruction. This address is actually 16 bits long, but if its 8 most significant bits are all zeros, we can omit them and use the zero page mode. This is like the common practice of referring to intervals of less than a minute as, for example, "20 seconds" rather than as "zero minutes and 20 seconds." If the 8 most significant bits of the address are not all zeros, we cannot drop them and must use the absolute (direct) addressing mode.

2) In the zero page addressing mode, the second byte of the instruction contains an address. Note that LDA $40 means "load the accumulator with the contents of memory location 0040." That location could contain any 8-bit number; it need not contain 40.

3) The instructions vary in length—LDA $40 and STA $41 require two bytes of memory, whereas BRK requires only one.

Let us now run this program with the data 6C in memory location 0040. The answer should be 6C in memory location 0041. You should clear memory location 0041 before running the program to convince yourself that the computer is actually doing something.

ENTERING AND RUNNING THE DATA TRANSFER PROGRAM

Enter and run Program 1-1 as follows:

ENTER PROGRAM

1) Press RS if necessary.
2) Examine memory location 0200 with the key sequence

```
AD
0
2
0
0
```

3) Enter the hexadecimal program with the key sequence

```
DA
A
5
+
4
0
+
8
5
+
4
1
+
0
0
```

You can verify that the program has been entered correctly by first examining memory location 0200 and then using the + key to examine locations 0201 through 0204.

ENTER DATA

1) Examine memory location 0040 (hex) with the key sequence

AD
0
0
4
0

2) Enter the data (6C) with the key sequence

DA
6
C

You may also want to clear memory location 0041 with the sequence +, 0, 0.

RUN PROGRAM

You can now execute the program with the key sequence

AD
0
2
0
0
GO

Remember, the program starts in memory location 0200. The final GO transfers control from the monitor to the program that you have entered. Control will return to the monitor when the computer executes the BRK instruction.

EXAMINE RESULTS

Finally, you can examine the result (after running the program) with the key sequence

```
AD
0
0
4
1
```

Remember, the program stores the result in memory location 0041. The computer does not tell you the answer by itself (regardless of what some fiction writers think). All the computer does is execute the program (which takes about 6 μs) and return control to the monitor (since you put a BRK instruction at the end).

PROBLEM 1-1

Run Program 1-1 again with the following data:

a) F0

b) 28

PROBLEM 1-2

Make Program 1-1 do the following:

a) Store the data in memory location 0042.

b) Move the contents of memory location 0041 to memory location 0040.

PROBLEM 1-3

Revise Program 1-1 so that it uses index register X instead of the accumulator. What changes are required in the binary operation codes for the load and store instructions? Remember that you can use Table 0-1 to convert hexadecimal numbers to binary. What changes are required if you revise the program to use index register Y?

PROBLEM 1-4

Revise Program 1-1 so that it transfers data from memory location 1780 (hex) to 1781 (hex). Now you must use the absolute addressing mode instead of the zero page mode. What are the advantages of the zero page mode? Be careful to load the two-byte absolute addresses in the upside-down order in which the 6502 processor expects them.

PROBLEM 1-5

Write and run a program that moves the contents of memory location 0040 to 0042 and the contents of memory location 0041 to 0043.

Sample Problem:

Data: (0040) = C6
(0041) = 5E
Result: (0042) = C6
(0043) = 5E

How much longer is the program if the memory addresses are 1780 through 1783 instead of 0040 through 0043?

PROCESSING DATA

Of course, we usually want to process the data rather than just move it from one place in memory to another. For example, the following program shifts each bit of the data left one position and clears the least significant bit before storing the result in memory location 0041. The computer here does exactly what an 8-bit shift register would do. The program is

```
LDA     $40     ;GET DATA
ASL     A       ;SHIFT DATA LEFT
STA     $41     ;STORE RESULT
BRK             ;RETURN TO KIM MONITOR
```

The only new instruction is ASL A, which shifts the contents of the accumulator left one position and clears the least significant bit. Note in Figure 1-1 that the standard procedure is to number bit positions starting with 0 at the right (least significant bit) and ending with 7 at the left (most significant bit). ASL A is a one-byte instruction; the processor knows what to do from the operation code alone.

Run this program (Program 1-2 is the hexadecimal version) with the data 01 (00000001 binary) in memory location 0040. The answer should be 02 (00000010 binary) in memory location 0041. Why? Remember, you can use Table 0-1 to convert hexadecimal numbers to binary, and vice versa. For example, 01 hexadecimal is 00000001 binary since 0 is 0000 binary and 1 is 0001 binary. Going the other way, 00000010 binary is 02 hexadecimal since 0000 binary is 0 in hexadecimal and 0010 binary is 2 in hexadecimal. Note that you must split the 8-bit byte down the middle to form two hexadecimal digits.

PROGRAM 1-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A5	LDA	$40
0201	40		
0202	0A	ASL	A
0203	85	STA	$41
0204	41		
0205	00	BRK	

PROBLEM 1-6

Run Program 1-2 again with the following data:

a) 40. The answer should be 80.

b) C7. The answer should be 8E. What happens to the 1 that is originally in bit 7?

PROBLEM 1-7

Can you revise Program 1-2 so that it uses index register X or index register Y? Are there instructions that shift the index registers? The accumulator is the only 6502 register that we can use for data processing; we will discuss the special uses of the index registers later.

PROBLEM 1-8

Make Program 1-2 shift the data right instead of left. **Hint**: Replace ASL A with LSR A.

Sample Problems (the parentheses around a memory address indicate "contents of"):

a) (0040) = 02
Result: (0041) = 01

b) (0040) = C7
Result: (0041) = 63

What happens to the 1 that is originally in bit 0?

PROBLEM 1-9

Revise Program 1-2 so that it shifts the contents of two successive memory locations and stores the results in the next two locations. That is, the new program should store the shifted version of location 0040 in 0042 and the shifted version of 0041 in 0043.

Sample Problem:

(0040) = 01

(0041) = 08

Result:

(0042) = 02

(0043) = 10

LOGICALLY ANDING TWO VALUES

We can easily change the left shift program to a logical AND program. The task now is to logically AND the contents of memory locations 0040 and 0041 and place the result in memory location 0042. Here the computer is doing what we could do in TTL logic with two 7408 quadruple two-input AND gates. Note, however, that the microcomputer contains the logic required to implement a variety of tasks; we do not have to add more circuits to perform different or additional functions. The logical AND program is

```
LDA     $40     ;GET FIRST OPERAND
AND     $41     ;LOGICALLY AND SECOND OPERAND
STA     $42     ;STORE RESULT
BRK
```

Program 1-3 is the hexadecimal version. We have started this program in memory location 0210 so that we can leave Program 1-2 in memory for later use. Remember that the logical AND function is defined as follows on a bit-by-bit basis:

INPUT 1	INPUT 2	INPUT 1 AND INPUT 2
0	0	0
0	1	0
1	0	0
1	1	1

PROGRAM 1-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0210	A5	LDA	$40
0211	40		
0212	25	AND	$41
0213	41		
0214	85	STA	$42
0215	42		
0216	00	BRK	

Enter Program 1-3 into memory and execute it for the following sample cases. Remember to start execution at address 0210, *not* 0200. Use Table 0-1 to convert the hexadecimal values into binary in order to check your results.

a) Data: (0040) = 23
(0041) = 34
Result: (0042) = 20

b) Data: (0040) = F0
(0041) = 3B
Result: (0042) = 30

c) Data: (0040) = 0C
(0041) = 77
Result: (0042) = 04

PROBLEM 1-10

Make Program 1-3 logically OR the two data entries and store the result. How would you implement a logical EXCLUSIVE OR? Refer to Table 3-3 if you cannot remember how the logical functions work.

Sample Problem:

Data: (0040) = 23
(0041) = 34
Result: (0042) = 37 for logical OR
= 17 for logical EXCLUSIVE OR

EXAMINING REGISTERS

One way to see what is actually happening in a program is to examine the processor's registers. You can examine the program counter by pressing

the PC key (second row from the top, third column from the left). On returning to the monitor, the processor stores all the user registers in certain assigned memory locations (see Table 1-1). You can thus examine any register by simply examining the corresponding memory address. Unfortunately, there is no way of telling which register you are examining except by referring to Table 1-1.

We have not yet discussed some of the registers. We will describe the status register in Laboratory 2 and the stack pointer in Laboratory A. The program counter (PC) contains the address of the next instruction that the CPU will fetch from memory. Each time the CPU uses the program counter, it adds one to its contents. Thus the computer will execute instructions sequentially unless it is specifically told to do otherwise. Since the program counter is 16 bits long, it can hold a complete memory address. Note, however, that it takes two memory locations (see Table 1-1) to hold the program counter's value.

> **Warning:** If you want to examine the current contents of the registers, do not press RS. This key reinitializes the CPU and may change the registers.

Since Table 1-1 is difficult to remember and a nuisance to look up repeatedly, we have simply taped a copy of it to our KIM. We recommend strongly that you do the same.

Table 1-1

KIM REGISTER STORAGE AREA

MEMORY ADDRESS (HEX)	LABEL	FUNCTION
00EF	PCL	Low-order byte of program counter
00F0	PCH	High-order byte of program counter
00F1	P	Status register
00F2	SP	Stack pointer
00F3	A	Accumulator
00F4	Y	Index register Y
00F5	X	Index register X

Note: Tape a copy of this table to your computer or next to it.

CHANGING REGISTERS

You can change the contents of the registers by changing the memory locations listed in Table 1-1. For example, let us assume that we want to place 4C in index register Y. All that we must do is:

1) Examine memory address 00F4.
2) Change its contents to 4C.

When you next press GO, the processor will start executing your program with the value 4C in index register Y. Note that you must use this approach to change the program counter; there is no way to change its value by using the PC key.

Run Program 1-2 with (0040) = C7. What are the final contents of the accumulator and the program counter? Does it make any difference if you clear the accumulator initially (i.e., load it with 00)?

A peculiar feature of BRK is that it makes the computer return to the monitor with a program counter value two higher than what you might expect. That is, after your program has been executed, the program counter will contain the address of the final BRK instruction plus 2. Besides being confusing, the extra 2 also makes it difficult to resume a program that you may have interrupted for debugging purposes (see Laboratory 8).

PROBLEM 1-11

Run Program 1-3 with (0040) = 23 and (0041) = 34. What are the final contents of the accumulator and program counter? Does it matter if you clear the accumulator initially? What happens to values you load into the index registers before executing Program 1-3? Try several different values (e.g., 00, FF, AA, 55) and see if the results vary. What happens if you reset the computer after loading the index registers? What happens if you enter 6F into the stack pointer (memory location 00F2) and then reset the computer?

COMMON OPERATING ERRORS

By now, you have undoubtedly discovered some annoying errors that plague KIM users. The most common mistakes are the following:

1) Forgetting to initialize the KIM properly. Before you start working with the KIM, be sure that you have moved the slide switch in the top row to the off position and that you have placed 1C00 in memory locations 17FE and 17FF (the return address for BRK) and in memory locations 17FA and 17FB (the return address for the ST key, which we will discuss later in this section). Since the return addresses are stored in volatile memory, you must reload them whenever you power up the KIM.

2) Using the wrong entry mode. Since the KIM does not indicate whether it is in the address or the data mode, this is a particularly common error. Be careful that you do not change important memory

locations (such as the return addresses) inadvertently. You can tell which mode the KIM is in by noting where new digits appear.

3) Executing the data instead of the program. For example, you may press AD, 0, 0, 4, 0, GO instead of AD, 0, 2, 0, 0, GO. This mistake causes the KIM to execute the data as if it consisted of instructions. One way to limit the damage is to place 1 or 2 BRK instructions (00) at the end of the data. The computer will usually encounter one of these and return control to the monitor.

4) Forgetting to run the program. That is, entering the program and the data and waiting for something to happen. This is comparable to entering data into your calculator and waiting for it to produce a result. Neither a computer nor a calculator will produce an answer until it has been directed to execute a program.

5) Starting program execution at the wrong address. The computer will execute whatever instructions it finds at the address you specify. This is a particularly annoying problem if you have several programs in memory or if you vary your starting addresses. One partial solution is to place BRK instructions at the addresses that you might enter accidentally.

Every once in a while, you will make an error that causes the computer to lose its way and never return to the monitor. If this happens, press RS (reset) or ST (stop). Pressing the ST key causes the computer to transfer control to the address in memory locations 17FA and 17FB. If you place 1C00 in those locations (00 in 17FA and 1C in 17FB), ST will transfer control back to the monitor. The advantage of ST over RS is that ST does not reinitialize the system or change the registers. One way to keep the computer from getting lost is to place one or two extra BRK instructions at the end of each program (the computer occasionally skips the first BRK).

Forgetting to initialize the KIM is a common and annoying error. You can remind yourself of this requirement by taping a copy of the initialization procedure (Figure 1-4) to the computer.

```
INITIALIZATION PROCEDURE

SINGLE-STEP SWITCH OFF
(17FA) = 00
(17FB) = 1C
(17FE) = 00
(17FF) = 1C
```

FIGURE 1-4. Listing of the procedure for initializing a KIM (to be taped to your computer or next to it).

KEY POINT SUMMARY

1) Most simple 6502 programs use the accumulator as the center of operations. The programs begin by loading the accumulator from memory and end by storing the result (from the accumulator) in memory.

2) The easiest addressing mode to use for loading and storing data is the direct mode, in which the instruction contains the address the CPU needs to perform the operation. This address follows the operation code in memory.

3) If the 8 most significant bits of a direct address are all zeros, we can omit them from the instruction by using the zero page addressing mode. This makes page zero of memory special, since instructions that use addresses on that page occupy less memory and execute faster than instructions that use addresses on other pages.

4) If the 8 most significant bits of a direct address are not all zeros, we cannot omit them and must use the absolute addressing mode. An absolute address occupies 2 bytes of memory with the 8 most significant bits in the second byte. That is, the address is stored upside down.

5) To run a program on the KIM-1 microcomputer, you must enter the program and data into memory, execute the program, and examine the results.

6) The accumulator is special in the 6502 microprocessor, because it is the only register that can be used in most processing operations.

7) You can examine the microprocessor's registers by examining the memory locations assigned by the KIM monitor. These locations are addresses 00EF through 00F5. You can examine the program counter (a 16-bit register) by pressing the PC key. You can change the registers by changing the appropriate memory locations after examining them.

8) Common errors in operating the KIM include failing to initialize it properly (particularly forgetting to load the return addresses in memory locations 17FA and 17FB and 17FE and 17FF), using the wrong entry mode, executing the data instead of the program, forgetting to run the program, and starting program execution at the wrong address.

Laboratory 2

Simple Input

PURPOSE

To learn how to use the input ports on the KIM microcomputer.

PARTS REQUIRED

Eight switches or pushbuttons attached to the Application Connector as shown in Figure 2-1. Table 2-1 contains a list of the pin assignments.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, Chapter 3.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 72-104, 369-370.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11-1 through 11-12, 11-39 through 11-49.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 19-20, 33-35, 40, 44-48, 54-68, 222-225.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 26 (hexa-

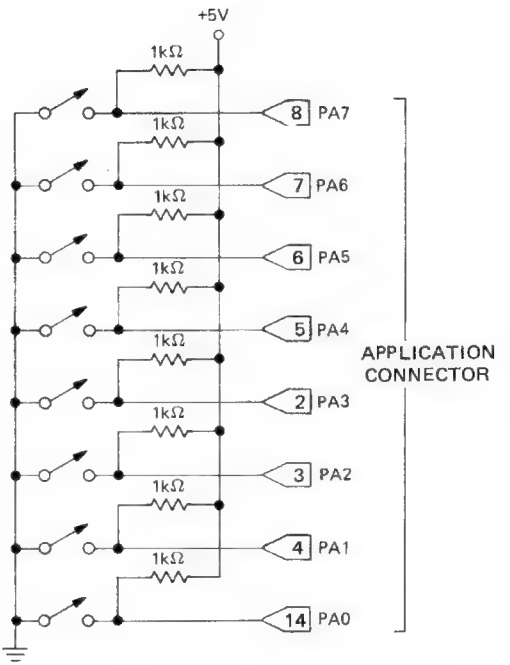


FIGURE 2-1. Attachment of switches to the Application Connector. The user 6530 device is referred to as 6530-003 in the KIM manuals.

decimal arithmetic), 31-32 (AND function), 62-63 (arithmetic circuits), 169-171 (status register), 174-180 (arithmetic/logic unit), 203-206 (simple input port), 312-316 (6502 flags), 319 (immediate addressing), 338-340 (compare and bit test instructions), 346-355 (conditional branch instructions).

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 3 and 5.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 3, 4, 5, 11.

WHAT YOU SHOULD LEARN

1) What memory-mapped input/output is and which 6502 instructions are commonly used for input and output.
2) How to use the LDA instruction to read data from an input port.
3) How to determine whether a switch is open or closed.

4) How to examine single bits of data.
5) How to use conditional branch instructions.
6) How to calculate relative offsets for branch instructions.
7) Which bit positions can be accessed most easily.
8) How to use shift and bit test instructions to examine single bits of data.
9) What flags are available on the 6502 CPU.
10) How to examine the flags.
11) How 6502 instructions affect the flags.
12) How to handle a series of switch closures.
13) How to recognize a starting (synchronization) character in communications.

Table 2-1

APPLICATION CONNECTOR PIN ASSIGNMENTS FOR PORT A OF THE USER 6530 DEVICE (6530-003)

ASSIGNMENT	PIN
Bit 0 (PA0)	14
Bit 1 (PA1)	4
Bit 2 (PA2)	3
Bit 3 (PA3)	2
Bit 4 (PA4)	5
Bit 5 (PA5)	6
Bit 6 (PA6)	7
Bit 7 (PA7)	8

TERMS

Arithmetic-logic unit (or **ALU**)—a device that can perform any of a variety of arithmetic or logical functions; function inputs select which function is performed during a particular cycle.

Branch instruction—*see* Jump instruction.

Carry flag—a flag that is 1 if the last operation generated a carry from the most significant bit and 0 if it did not.

Flag (or **condition code** or **status bit**)—a single bit that indicates a condition within the computer, often used to choose between alternative instruction sequences.

Flip-flop—a digital electronic device with two stable states that can be made to switch from one state to the other in a reproducible manner.

Floating—not tied to any logic level. TTL and MOS devices usually interpret a floating input as a logic 1.

Immediate addressing—an addressing method in which the data required by an instruction is part of the instruction, usually immediately following the operation code in memory.

Inverted borrow—a bit that is 0 if a subtraction required a borrow and 1 if it did not.

Isolated input/output—an addressing method for I/O ports that uses a decoding system distinct from that used by the memory section. I/O ports do not occupy memory addresses.

Jump instruction (or **branch instruction**)—an instruction that places a new value in the program counter, thus departing from the normal one-step incrementing. Jump instructions may be conditional; that is, the new value may be placed in the program counter only if a condition is true.

Label—a name attached to an instruction or statement in a program that identifies the location in memory of the machine language code or assignment produced from that instruction or statement.

Mask—a bit pattern that isolates 1 or more bits from a group of bits.

Memory-mapped input/output—an addressing method for I/O ports that uses the same decoding system used by the memory section. The I/O ports thus occupy memory addresses.

Negative flag—*see* sign flag.

Parallel interface—an interface between a CPU and input or output devices that handle data in parallel (more than one bit at a time).

Peripheral interface—one of the 6500 family versions of a parallel interface; examples are the 6520, 6522, 6530, and 6532 devices.

Port—the basic addressable unit of the computer's input/output section.

Relative addressing—an addressing method in which the address specified in the instruction is the offset from a base address. Program relative addressing (in which the base address is the program counter) makes it easy to move programs from one place in memory to another.

Relative offset—the difference between the actual address to be used in an instruction and the current value of the program counter.

Relocatable—can be placed anywhere in memory without changes: that is, a program that can occupy any set of consecutive memory addresses.

Serial—one bit at a time.

Shift instruction—an instruction that moves all the bits of the data by a certain number of bit positions, just as in a shift register.

Sign flag—a flag that contains the most significant bit of the result of the previous operation.

SPST switch—single-pole, single-throw switch with one common line and one output line.

Status register (or **status word** or **condition code register**)—a register that contains bits describing the current state of the computer. This register usually holds all the flags.

Synchronization (or **sync**) **character**—a character that is used only to synchronize the transmitter and the receiver. The character does not contain any actual information.

Two's complement—a binary number that, when added to the original number in a binary adder, produces a zero result. The two's complement of a number may be obtained by subtracting the number from zero or by logically complementing the number (replacing each 0 bit with a 1 and each 1 with a 0, thus forming the *one's complement*) and adding 1.

Zero flag—a flag that is 1 if the last operation produced a result of zero and 0 if it did not.

6502 INSTRUCTIONS

ADC—add with carry; add the contents of the specified memory location and the contents of the CARRY flag to the accumulator. The result is placed in the accumulator.

BCC—branch if carry clear; jump over the specified number of memory locations if the CARRY flag is 0; otherwise, proceed to the next instruction in sequence.

BCS—branch if carry set; jump over the specified number of memory locations if the CARRY flag is 1; otherwise, proceed to the next instruction in sequence.

BEQ—branch if equal to zero; jump over the specified number of memory locations if the ZERO flag is 1; otherwise, proceed to the next instruction in sequence.

BIT—bit test (logical AND with no result saved): logically AND the contents of the accumulator with the contents of the specified memory location but leave the accumulator unchanged. This instruction affects only the flags. BIT has the special feature that it sets the NEGATIVE flag from bit 7 of the specified memory location and the OVERFLOW flag from bit 6 without considering the value in the accumulator. The only flag that depends on the logical AND operation is the ZERO flag. BIT allows only zero page and absolute (direct) addressing.

BMI—branch if minus; jump over the specified number of memory locations if the NEGATIVE (SIGN) flag is 1; otherwise, proceed to the next instruction in sequence.

BNE—branch if not equal to zero; jump over the specified number of memory locations if the ZERO flag is 0; otherwise, proceed to the next instruction in sequence.

BPL—branch if plus; jump over the specified number of memory locations if the NEGATIVE (SIGN) flag is 0; otherwise, proceed to the next instruction in sequence.

BVC—branch if overflow clear; jump over the specified number of memory locations if the OVERFLOW flag is 0; otherwise, proceed to the next instruction in sequence.

BVS—branch if overflow set; jump over the specified number of memory locations if the OVERFLOW flag is 1; otherwise, proceed to the next instruction in sequence.

CMP—compare memory and accumulator; subtract the contents of the specified memory location from the contents of the accumulator but leave the accumulator unchanged. This instruction affects only the flags.

SBC—subtract with carry; subtract the contents of the specified memory location and the complemented contents of the CARRY flag from the contents of the accumulator. The result is (A) = (A) − (M) − (1-CARRY), where M is the specified memory address. Note that the CARRY flag acts as an inverted borrow on the 6502 microprocessor, the opposite of its role on such processors as the 8080, 6800, and Z-80.

6502 INPUT/OUTPUT OPERATIONS

The 6502 microprocessor has no specific input/output (I/O) instructions. Instead, it treats I/O ports as if they were memory locations. (This approach is referred to as *memory-mapped input/output*, as opposed to *isolated input/output*, in which I/O ports and memory locations are

addressed separately.) Note that the processor really cannot tell memory from I/O; all the processor does is produce addresses and control signals and transfer data.

In memory-mapped input/output, any instruction that transfers data to or from memory can serve as an I/O instruction. The 6502 instructions that are most commonly used to perform I/O are:

- LDA (LOAD ACCUMULATOR) transfers 8 bits of data from the specified memory address (actually an input port) to the accumulator.
- STA (STORE ACCUMULATOR) transfers 8 bits of data from the accumulator to the specified memory address (actually an output port).
- BIT (BIT TEST) sets the flags as if the data from an input port had been logically ANDed with the contents of the accumulator. The contents of the accumulator are not changed.
- CMP (COMPARE MEMORY AND ACCUMULATOR) sets the flags as if the data from an input port had been subtracted from the contents of the accumulator. The contents of the accumulator are not changed.

One drawback to memory-mapped input/output is that it makes programs difficult to understand. A load instruction, for example, may be retrieving data from memory or fetching new data from an input device. Processors that allow isolated I/O generally have special operation codes (e.g., IN and OUT or READ and WRITE) for input and output. The resulting programs are easy to read, since one can readily distinguish input and output from operations on memory locations. Programs that use memory-mapped I/O always require thorough documentation.

SIMPLE INPUT

The KIM has two spare I/O ports in one of its 6530 Peripheral Interface/Memory devices (the user 6530 device or 6530-003 as it is denoted in the *KIM User Manual*). The ports occupy memory addresses 1700 (port A) and 1702 (port B). We will use port A for input and port B for output throughout this book.

Starting from reset, the following program will load the accumulator with the data from port A:

```
LDA     $1700
BRK
```

The hexadecimal version is Program 2-1. Before you load this program into memory, be sure that you have initialized your KIM properly; that is,

1) The slide switch in the top row is turned off.
2) 1C00 is in memory locations 17FE and 17FF (the return address for BRK) and in memory locations 17FA and 17FB (the return address for the ST key).

We will not mention initialization again, but you should check the slide switch and the return addresses each time you start working with your KIM.

Open all the switches attached to port A, reset the computer, and execute Program 2-1. What are the final contents of the accumulator (memory location 00F3, see Table 1-1)? What happens if you replace LDA $1700 with LDA $1701?

Close the switch attached to bit position 5 of port A and execute Program 2-1 again. What does the accumulator contain now?

PROGRAM 2-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	AD	LDA	$1700
0201	00		
0202	17		
0203	00	BRK	

PROBLEM 2-1

The computer interprets an open switch as a logic _____ and a closed switch as a logic _____ . How do you think the computer would interpret an unconnected or floating input?

PROBLEM 2-2

Determine the value that Program 2-1 will read from address 1700 if:

a) The switch attached to bit position 2 of port A is closed.
b) Switches attached to bit positions 2 and 5 are closed.
c) Switches attached to bit positions 0, 6, and 7 are closed.

Assume that all other switches are open.

PROBLEM 2-3

What happens if you replace LDA $1700 with LDA $1702? Does opening or closing switches attached to port A affect the input? Explain the result.

Remember the following points which we mentioned in Laboratory 1:

1) The standard procedure in the computer industry is to number bit positions starting with zero at the right. Thus, in an 8-bit word, the bits are numbered 0 through 7 from right to left, with bit 0 being least significant and bit 7 most significant. Figures 2-2 and 2-3 contain examples of the standard numbering. Be careful—switches and other input and output devices are often numbered differently from the computer convention (e.g., 1 to 8 or left to right).

2) Since address 1700 is not on page zero, we must use absolute addressing when referring to it. As you can see in Program 2-1, the 16-bit address occupies the two bytes of memory immediately following the operation code. The least significant bits always come first; this is the standard 6502 method for storing 16-bit addresses or data. The bytes appear to be stored upside down, but note that you need the least significant bits first if you are performing 16-bit addition or subtraction.

FLAGS AND CONDITIONAL BRANCHES

To have the computer determine if a switch is open or closed we must use:

1) The flags (also called *condition codes* or *status bits*).
2) The conditional branch instructions.

Instructions that transfer or process data also set the flags. A conditional branch (or jump) instruction allows the computer to choose between alternative paths depending on the current value of a flag.

The most important 6502 flags are:

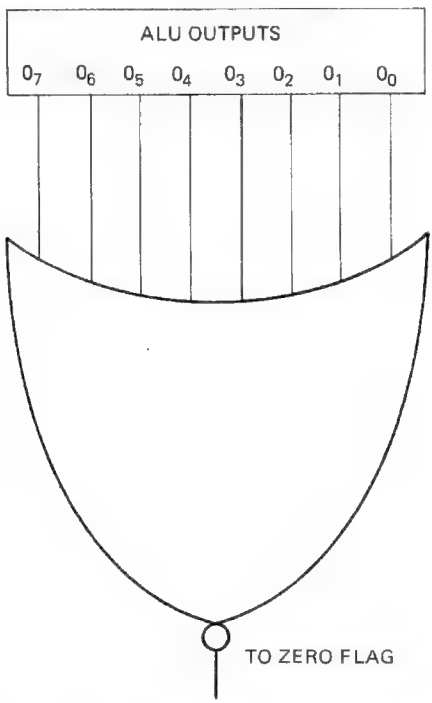
C (CARRY)—1 if the last arithmetic or shift instruction produced a carry, 0 if it did not.

N (NEGATIVE or SIGN)—1 if the result of the last instruction had a 1 in its most significant bit, 0 if it did not.

Z (ZERO)—1 if the result of the last instruction was zero, 0 if it was not zero.

These flags are simply flip-flops inside the 6502 processor. Figure 2-2 shows, for example, how the ZERO flag could be implemented on the 6502 chip with an eight-input logical NOR gate. The NEGATIVE (SIGN) and CARRY flags can be derived directly from the outputs of the accumulator or an arithmetic-logic unit (ALU).

The 6502 conditional branch instructions place a new value in the program counter if the specified flag has the specified value. If the flag has the opposite value, the computer simply proceeds to the next instruction in sequence. Conditional branch instructions make the computer "smart," that is, capable of making decisions based on current information. The computer thus becomes an intelligent controller. Table 2-2 lists the 6502's conditional branch instructions.



The output of the NOR gate is 1 if and only if all the ALU outputs are zero.

FIGURE 2-2. Possible implementation of a ZERO flag.

Table 2-2

6502 CONDITIONAL BRANCH INSTRUCTIONS

INSTRUCTION	FLAG USED	VALUE ON WHICH BRANCH OCCURS
BCC	CARRY	0
BCS	CARRY	1
BNE	ZERO	0
BEQ	ZERO	1
BPL	NEGATIVE (SIGN)	0
BMI	NEGATIVE (SIGN)	1
BVC	OVERFLOW	0
BVS	OVERFLOW	1

WAITING FOR A SWITCH TO CLOSE

Let us concentrate for now on the switch attached to bit 5 of address 1700 (switch 5, for short). The following program waits for you to close switch 5 and then returns control to the monitor. Remember that an open switch is a logic 1 and a closed switch is a logic 0 (see Figure 2-1).

```
WAITC   LDA   $1700          ;GET INPUT DATA
        AND   #%00100000     ;IS SWITCH 5 CLOSED?
        BNE   WAITC          ;NO, WAIT
        BRK
```

Let us now look at each instruction:

1) LDA $1700 loads the accumulator with the contents of the port to which the eight switches are attached. Note that the CPU must fetch 8 bits even though we are only interested in one of them.

WAITC is a name that we have given to the memory address in which the instruction LDA $1700 begins. Such a name is called a *label;* its sole purpose is to make the program easier to read and understand. People find names easier to remember than hexadecimal numbers; however, the KIM only accepts numbers, so the programmer (or the assembler program) must replace all labels with the actual addresses to which they refer. For example, if the last program starts in memory location 0200, the label WAITC refers to address 0200 and the instruction BNE WAITC must actually cause a branch to that address. Labels are convenient because

they are easy to find and change in a program listing. The name WAITC is arbitrary; we selected it because it suggests the idea of *wait*ing for a closure.

2) AND #%00100000 logically ANDs the contents of the accumulator with the binary number 00100000. The % means "binary" and the # before the number means "immediate" (i.e., the data is in the next byte of program memory). The result of the logical AND is 0 if the switch is closed (an 8-bit zero, remember) and 00100000 if the switch is open. (Verify this!) This procedure of singling out part of a group of bits is called *masking.*

3) BNE WAITC causes the processor to execute the instruction in memory location WAITC next if the ZERO flag is zero. Otherwise, the processor proceeds to the next instruction in sequence (BRK in this case). The ZERO flag is a flip-flop inside the 6502 CPU which is set to 1 if an operation produces a zero result. Watch out—the ZERO flag is *1* if the result was zero.

Program 2-2 is the hexadecimal version. Note the following features:

PROGRAM 2-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	AD	WAITC	LDA	$1700
0201	00			
0202	17			
0203	29		AND	#%00100000
0204	20			
0205	D0		BNE	WAITC
0206	F9			
0207	00		BRK	

1) LDA $1700 uses absolute addressing. This mode requires a 16-bit address stored upside down in the 2 bytes of memory following the operation code.

2) AND #%00100000 uses immediate addressing. This mode requires the data (00100000 binary or 20 hexadecimal) in the byte of memory following the operation code. Thus an instruction using immediate addressing contains the actual data, not the address of the data. Explain how immediate and direct addressing differ? Can STA be used with immediate addressing? Why not?

Although we have written the data in binary to make its purpose clearer, we must convert it to hexadecimal before we can enter it into the KIM's memory. The procedure is to split the binary version down the middle and use Table 0-1 to convert the two 4-bit sections to hexadecimal —0010 is 2 in hexadecimal and 0000 is 0.

3) BNE requires an 8-bit relative address or offset in the byte following the operation code. This offset tells the computer how many locations it should jump over (going backward or forward) from the end of the instruction (address 0207 in this case). A positive offset (most significant bit = 0) is added to the final address (e.g., an offset of 02 would be added to 0207 to make the destination 0209); the maximum positive offset is 7F or +127 decimal. A negative offset (most significant bit = 1) tells the computer how many locations backward to go (one back is FF, two back is FE, etc.). You can calculate the offset by subtracting the final address from the destination address; in our case, the subtraction is

```
  0200    (destination address)
- 0207    (final address at the end of the BNE instruction)
------
  FFF9
```

Only the F9 is significant; the largest negative offset is 80 hex or −128 decimal.

The usual way to perform hexadecimal subtraction is to calculate the two's complement of the number to be subtracted and add it to the other number. Table 2-3 contains the two's complements of all two-digit hexadecimal numbers and Table 2-4 is a hexadecimal addition table. Using these tables, we can calculate the offset as follows:

```
  0200  (destination address)
- 0207  (address immediately following the end of the BNE instruction)
```

The two's complement of 07 is F9 from Table 2-3. So the subtraction is equivalent to

```
   00
+ F9
----
   F9
```

Clearly this method is only a slight improvement over counting on one's fingers (counting is substantially easier if you happen to have 16 fingers or are a whiz at counting backward in hexadecimal). There are two ways out of this predicament:

1) Use an assembler, which will perform this rote task automatically.

Table 2-3

TWO'S COMPLEMENTS OF TWO-DIGIT HEXADECIMAL NUMBERS

	MSD															
LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	F0	E0	D0	C0	B0	A0	90	80	70	60	50	40	30	20	10
1	FF	EF	DF	CF	BF	AF	9F	8F	7F	6F	5F	4F	3F	2F	1F	0F
2	FE	EE	DE	CE	BE	AE	9E	8E	7E	6E	5E	4E	3E	2E	1E	0E
3	FD	ED	DD	CD	BD	AD	9D	8D	7D	6D	5D	4D	3D	2D	1D	0D
4	FC	EC	DC	CC	BC	AC	9C	8C	7C	6C	5C	4C	3C	2C	1C	0C
5	FB	EB	DB	CB	BB	AB	9B	8B	7B	6B	5B	4B	3B	2B	1B	0B
6	FA	EA	DA	CA	BA	AA	9A	8A	7A	6A	5A	4A	3A	2A	1A	0A
7	F9	E9	D9	C9	B9	A9	99	89	79	69	59	49	39	29	19	09
8	F8	E8	D8	C8	B8	A8	98	88	78	68	58	48	38	28	18	08
9	F7	E7	D7	C7	B7	A7	97	87	77	67	57	47	37	27	17	07
A	F6	E6	D6	C6	B6	A6	96	86	76	66	56	46	36	26	16	06
B	F5	E5	D5	C5	B5	A5	95	85	75	65	55	45	35	25	15	05
C	F4	E4	D4	C4	B4	A4	94	84	74	64	54	44	34	24	14	04
D	F3	E3	D3	C3	B3	A3	93	83	73	63	53	43	33	23	13	03
E	F2	E2	D2	C2	B2	A2	92	82	72	62	52	42	32	22	12	02
F	F1	E1	D1	C1	B1	A1	91	81	71	61	51	41	31	21	11	01

Table 2-4

HEXADECIMAL ADDITION TABLE

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

2) Buy a calculator that does hexadecimal arithmetic. The Texas Instruments Programmer is a popular model.

Henceforth, we will occasionally show how to obtain some relative offsets, but we will not emphasize hexadecimal arithmetic. In practice, one should automate this annoying task. If you must perform it by hand, always check your arithmetic (assuming that you are no better at it than we are).

Enter and run Program 2-2. What happens if you leave switch 5 open? What happens if you close other switches (i.e., switches 0, 1, 2, 3, 4, 6, or 7)?

PROBLEM 2-4

Make Program 2-2 wait for you to close switch 4 (i.e., the switch attached to bit position 4 of memory location 1700). Next try switch 2 and then switch 6. How easily could you change from one bit position to another if the system were implemented entirely in TTL logic?

PROBLEM 2-5

Change Program 2-2 so that it starts in memory location 0210. A program that can be placed anywhere in memory without any changes is called *relocatable.* Is Program 2-2 relocatable? Explain why the use of relative addressing in branch instructions is the key to this program's relocatability. Would the program be relocatable if the BNE instruction actually specified the complete destination address? Suggest some reasons why relocatable programs are desirable.

PROBLEM 2-6

What happens if you replace LDA $1700 and AND #%00100000 with LDA #%00100000 and BIT $1700? Close switch 5 and open all the other switches. What is in the accumulator before and after the processor executes BIT $1700? What happens if you replace BIT $1700 with AND $1700? What is the advantage of BIT over AND?

SPECIAL BIT POSITIONS

Since most instructions operate on 8 bits of data at a time, there is little to differentiate one bit position from another. However, some instructions and flags make certain bit positions more accessible than others. For example:

1) The instruction ASL (see Figure 1-1) shifts each bit left one position. The old value of bit 6 is placed in the NEGATIVE

or SIGN flag; it can then be used as a branch condition by either BMI or BPL.

2) The instruction LSR (see Figure 1-1) similarly places the value of bit 0 in the CARRY. It can then be used as a branch condition by either BCC or BCS.

3) The instruction LDA places bit 7 in the NEGATIVE or SIGN flag; it can then be used as a branch condition by either BMI or BPL.

So the following program will exit if you close switch 7:

```
WAITC   LDA   $1700   ;GET INPUT DATA
        BMI   WAITC   ;WAIT UNTIL SWITCH 7 IS CLOSED
        BRK
```

The hexadecimal version (Program 2-3) is much shorter than Program 2-2. Enter and run Program 2-3; try the following variations.

PROBLEM 2-7

Write two programs that wait for you to close switch 0, one using AND and one using LSR. Which program is shorter? Which will be executed faster?

PROBLEM 2-8

Write two programs that wait for you to close switch 6, one using AND and one using ASL. Which program is shorter? Which will be executed faster?

PROBLEM 2-9

A special feature of the 6502's BIT instruction is that it sets the NEGATIVE (SIGN) and OVERFLOW flags according to the values of bits 7 and 6, respec-

PROGRAM 2-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	AD	WAITC	LDA	$1700
0201	00			
0202	17			
0203	30		BMI	WAITC
0204	FB			
0205	00		BRK	

tively, of the memory location without even considering the value in the accumulator. Write a program that uses BIT to wait for you to close switch 6.

Hint: Use the instruction BVS (BRANCH IF OVERFLOW SET). This is the most common use of the 6502's OVERFLOW flag and it has nothing whatsoever to do with overflow.

The relative offset in Program 2-3 (memory location 0204) is given by

$$\begin{array}{rcr} 0200 & & 00 \\ -\,0205 & = & +\,\underline{FB} \\ & & FB \end{array}$$

where we obtained the two's complement of 05 from Table 2-3.

If you have only one or two switches (or other serial inputs) to attach to a port, which bit positions should you use and why? Which bit positions should you use for the switches that are read most frequently or that have the highest priority?

EXAMINING FLAGS

You can determine the current states of the 6502's flags by examining the processor status (P) register. This register is sometimes called the condition code or flag register.

Figure 2-3 shows the organization of the status register. We will describe the seldom used OVERFLOW (V), BREAK (B), DECIMAL

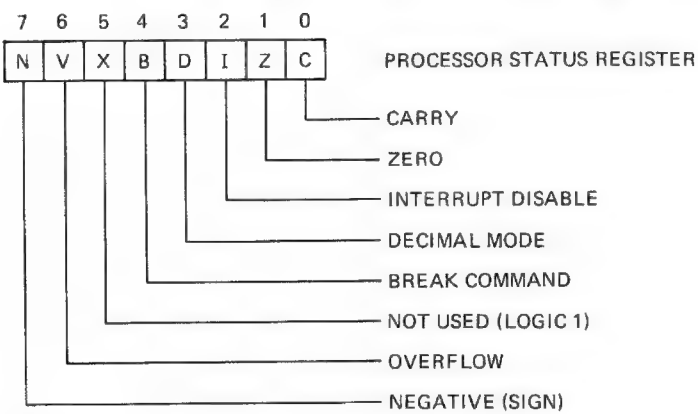


FIGURE 2-3. Organization of the 6502 Processor Status register (referred to as the P register). Bit 5 is not used and always appears as a logic 1.

MODE (D), and INTERRUPT DISABLE (I) flags later. Here the hexadecimal notation is a nuisance, because only the binary values are meaningful. You can use Table 0-1 to convert hexadecimal to binary.

The designers of the microprocessor decide how its instructions will affect its flags. The only way you can determine what will happen is by consulting your instruction card or Table A1-1 of this book. Fortunately, the designers of the 6502 generally chose the effects of instructions on flags to be what a reasonable person would expect. You can see how an instruction affects the flags by initializing the status register, specifying the operands, letting the computer execute the instruction, and then examining the final value of the status register. The results will depend on the instruction and on the operands. Remember not to reset the computer because that may change the flags.

For example, if we start with (A) = 80 hex and (P) = 04 hex (leaving I equal to 1 to disable interrupts), executing the instruction AND #$80 makes (P) = B4 hex = 10110100 binary (see Table 0-1). The values of the major flags are thus:

NEGATIVE (SIGN)	= 1 (bit 7 of P)
ZERO	= 0 (bit 1 of P)
CARRY	= 0 (bit 0 of P)—this is unchanged.

These values reflect the fact that logically ANDing 80 hex (10000000 binary) with itself produces a result of 80 hex (10000000 binary). The ZERO flag is set to 0 since the result is not zero. The NEGATIVE flag is set to 1 since bit 7 of the result is 1. As you might expect, a logical operation does not affect the CARRY flag. Remember, you must use the procedure described in Laboratory 1 (see Table 1-1) to initialize the status register (in memory location 00F1) and the accumulator (in memory location 00F3).

PROBLEM 2-10

Determine the final values of the NEGATIVE, ZERO, and CARRY flags after the processor executes AND #$80 for the following initial conditions:

a) (P) = (00F1) = FF
   (A) = (00F3) = 80

b) (P) = (00F1) = 04
   (A) = (00F3) = 7F

c) (P) = (00F1) = FF
   (A) = (00F3) = 7F

Hint: Use the program

```
AND     #$80
BRK
```

Do the final values of the flags depend on their initial values? What happens to information stored in the flags if the processor executes the instruction AND #$80? Remember, the NEGATIVE flag is bit 7 of the status register, the ZERO flag is bit 1, and the CARRY flag is bit 0.

PROBLEM 2-11

Arithmetic and shift instructions affect the CARRY flag. For example, consider the instruction ADC (ADD WITH CARRY) which adds the contents of the specified memory location and the contents of the CARRY flag to the accumulator. Determine the final values of the NEGATIVE, ZERO, and CARRY flags after the processor executes ADC #$80 for the following initial conditions:

a) (P) = (00F1) = 04
   (A) = (00F3) = 80

b) (P) = (00F1) = F7 (leaving the DECIMAL MODE flag cleared, see Laboratory 9)
   (A) = (00F3) = 80

c) (P) = (00F1) = 04
   (A) = (00F3) = 7F

d) (P) = (00F1) = F7
   (A) = (00F3) = 7F

Which flags does ADC affect and how? What happens to information stored in the flags if the processor executes the instruction ADC #$80? What is the final value of the accumulator in each case?

WAITING FOR TWO CLOSURES

You can easily extend Program 2-2 to wait for more than one closure. The following program will wait for switches 2 and 5 to be closed in that order, assuming that you start with all the switches open.

```
WAIT1   LDA     $1700           ;GET INPUT DATA
        AND     #%00000100      ;IS SWITCH 2 CLOSED?
        BNE     WAIT1           ;NO, WAIT
WAIT2   LDA     $1700           ;GET INPUT DATA
        AND     #%00100000      ;IS SWITCH 5 CLOSED?
        BNE     WAIT2           ;NO, WAIT
        BRK
```

PROGRAM 2-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	AD	WAIT1	LDA	$1700
0201	00			
0202	17			
0203	29		AND	#%00000100
0204	04			
0205	D0		BNE	WAIT1
0206	F9			
0207	AD	WAIT2	LDA	$1700
0208	00			
0209	17			
020A	29		AND	#%00100000
020B	20			
020C	D0		BNE	WAIT2
020D	F9			
020E	00		BRK	

Enter and run this program; the hexadecimal version is Program 2-4. What happens if you close switch 2 and then switch 5? What happens if you reverse the order? Explain the result. Does the result change if you allow only one switch to be closed at a time?

PROBLEM 2-12

Make Program 2-4 wait for you to close switch 3 followed by switch 1. What happens if you leave one of the switches closed all the time?

Write a program that waits for a particular sequence of switch closures and let someone else try to guess which sequence you chose. What happens if the other person simply closes all the switches? You can defeat this strategy by using the instruction CMP instead of AND. CMP (COMPARE MEMORY AND ACCUMULATOR) subtracts the specified memory location from the accumulator and sets the flags appropriately, but does not change the accumulator. Thus CMP will set the ZERO flag only if its operands are equal. Since the outcome depends on the positions of all eight switches, a subsequent BNE instruction will cause a branch unless all eight switches are set correctly. For example, after the sequence

```
LDA     $1700           ;GET INPUT DATA
CMP     #%00100000
```

the ZERO flag will be set to 1 only if all eight switches are in the positions specified by CMP's operand (0 = closed, 1 = open). That is, CMP #%00100000 will set the ZERO flag only if all the following conditions hold:

- Switch 7 is closed.
- Switch 6 is closed.
- Switch 5 is open.
- Switch 4 is closed.
- Switch 3 is closed.
- Switch 2 is closed.
- Switch 1 is closed.
- Switch 0 is closed.

Note how different these eight conditions are from the single condition needed to make AND #%00100000 set the ZERO flag. Remember that AND is a bit-by-bit operation, whereas CMP is a byte-wide operation.

PROBLEM 2-13

Explain the differences between SBC and CMP. What are the final values of NEGATIVE, ZERO, CARRY, and the accumulator after the processor executes SBC #$80 and after CMP #$80 for the following initial conditions?

a) (P) = (00F1) = 04
   (A) = (00F3) = 80

b) (P) = (00F1) = F7 (leaving the DECIMAL MODE flag cleared)
   (A) = (00F3) = 80

c) (P) = (00F1) = 04
   (A) = (00F3) = 81

d) (P) = (00F1) = F7
   (A) = (00F3) = 81

What are the advantages of CMP over SBC?

PROBLEM 2-14

Write a program that waits for switch 0 to be closed and then for switch 7 to be closed. Write one version that ignores the states of other switches and one that only works if all the other switches are open. What happens in the second version if you reverse the order (i.e., close switch 7 first and then switch 0) or leave either switch 0 or switch 7 closed all the time?

PROBLEM 2-15

Write a program that waits for switches 2 and 5 to be closed at the same time and then for switches 0 and 7 to be closed at the same time.

PROBLEM 2-16

Write a program that waits for either switch 2 to be closed followed by switch 5 or switch 5 followed by switch 2. Allow only one switch to be closed at a time.

SEARCHING FOR A STARTING CHARACTER

In communications applications, the input data will be the most recent character received from the channel. Of course, if the transmitter is not sending anything, that character will be meaningless. Assume that the transmitter precedes every message with the hexadecimal value 7F (a so-called synchronization or sync character since it is not part of the actual information).

PROBLEM 2-17

Write and run a program that waits for 7F to appear in memory location 1700. An easy way to produce 7F is to first open all the switches (producing FF) and then close switch 7.

If the input data has a completely random value, how often will the computer think that it has found a message? That is, what is the probability that the random value will be 7F? How often would the computer find a message erroneously if the synchronizing pattern were two 7F characters? How about three 7F characters?

Clearly, a longer synchronizing pattern results in fewer false alarms. On the other hand, noise in the communications channel could cause a 7F character to be received as something else and a real message would then be missed.

PROBLEM 2-18

Write and run a program that will accept the input as 7F regardless of the value of bit 2. How often would this program find a message erroneously (i.e., what is its probability of receiving a random value that looks just like a synchronization character)?

KEY POINT SUMMARY

1) The 6502 microprocessor has no specific input/output instructions. Instead, input and output ports are addressed as memory loca-

tions (memory-mapped I/O), and any instruction that transfers data to or from memory can be used as an I/O instruction.

2) The KIM microcomputer has two I/O ports (in one of its 6530 devices) that are available for user-defined input/output. These ports occupy memory addresses 1700 and 1702. We will use port A (address 1700) for input and port B (address 1702) for output.

3) The 6502 microprocessor has three major flags, which are set according to the results of certain instructions. These are the CARRY, ZERO, and NEGATIVE (or SIGN) flags. Almost all instructions affect the ZERO and NEGATIVE flags, whereas only arithmetic and shift instructions affect the CARRY flag.

4) A conditional branch instruction forces a jump if the specified condition is true. If the condition is false, the mieroprocessor continues executing instructions in their normal sequence. Conditional branch instructions are the keys to computer decision making.

5) The processor can determine the value of a specific bit in a register or memory location by logically ANDing the contents with a mask. The mask has a 1 in the specified bit position and 0's elsewhere. The result is zero if and only if the specified bit position contains zero. Bit positions at either end of a byte can be handled by using the SIGN or CARRY flag and the load, shift, or bit test instructions.

6) The processor can determine whether a register or memory location contains a specified value by subtracting the value from the contents. The result is zero if and only if the operands are equal (i.e., if the register or memory location contains the specified value).

7) The processor performs logical operations (AND, OR, EXCLUSIVE OR, NOT) bit by bit, 8 bits at a time. However, arithmetic operations (ADD, SUBTRACT) involve carries or borrows, so the bit positions are not independent.

Laboratory 3

Simple Output

PURPOSE

To learn how to use the output ports on the KIM microcomputer.

PARTS REQUIRED

Seven single LEDs (light-emitting diodes) attached to the user 6530 device (6530-003 in the KIM schematic) as shown in Figure 3-1. Table 3-1 contains a list of the pin assignments for the Application Connector. Bit 6 of user 6530 port B is not available externally.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, pp. 50-80, 111-112.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 72-104, 376-377, 413, 414.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, pp. 11-1 through 11-12, 11-39 through 11-42, 11-61 through 11-64.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 48-54, 83-91.

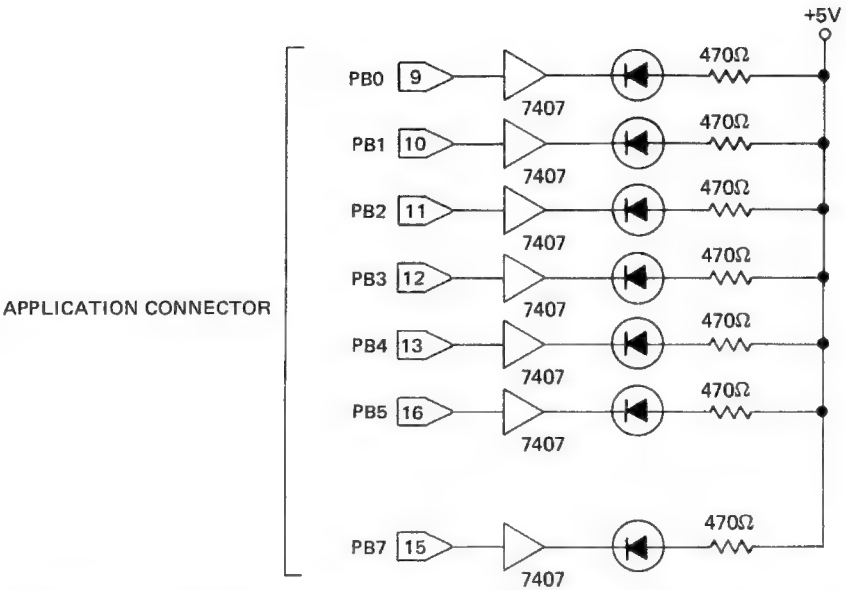


FIGURE 3-1. Attachment of LEDs to the Application Connector. (**Note:** PB6 is not available externally.)

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.*, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 31-33 (logic gates), 33-34 (logic equivalences), 200-202 (LED displays), 329-334 (logical instructions), 337-338 (decrement/increment instructions), 368-373 (timing loops).

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 3 and 5.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 3, 4, 5, 11.

WHAT YOU SHOULD LEARN

1) How to make 6530 I/O lines into inputs or outputs.
2) How to use a 6530 I/O port for output.
3) How to turn LEDs on and off.
4) How to write a delay routine.
5) How to produce long delays by nesting routines.
6) How to turn individual LEDs on and off by using logical functions.
7) How to complement (invert) data.

8) How to operate LED displays according to a specific duty cycle.
9) How to use software to control the appearance of displays.

Table 3-1

APPLICATION CONNECTOR PIN ASSIGNMENTS FOR USER 6530 PORT B

ASSIGNMENT	PIN
Bit 0 (PB0)	9
Bit 1 (PB1)	10
Bit 2 (PB2)	11
Bit 3 (PB3)	12
Bit 4 (PB4)	13
Bit 5 (PB5)	16
Bit 6 (PB6)	Not available
Bit 7 (PB7)	15

TERMS

Active-high—the active state is a high logic level.

Active-low—the active state is a low logic level.

Anode—positive terminal.

Cathode—negative terminal.

Complement—*see* One's complement.

Data direction register—a register that determines whether I/O lines are inputs or outputs.

Duty cycle—the period of time during which a device is active as part of a total period of continuous operation.

Light-emitting diode (LED)—a semiconductor device that emits light when biased in the forward direction.

Multiplex—to use one functional unit for several different purposes on a shared basis.

Negative logic—circuitry in which a logic zero is the active or "on" state.

Nesting—constructing programs in a hierarchical manner with one level contained within another, etc. The nesting level is the number of transfers of control required to reach a particular part of a program without returning to a higher level.

One's complement—a bit-by-bit logical inversion of a binary number, replacing each 0 bit with a 1 and each 1 bit with a 0.

Software delay—a program that has no function other than to waste time.

Turn-on time—the time required for a device to enter the "on" state after the signal to do so has been received.

6502 INSTRUCTIONS

DEC—decrement by 1; subtract one from the contents of the specified memory location. DEC cannot be applied to the accumulator.

DEX—decrement index register X by 1; subtract one from the contents of index register X.

DEY—decrement index register Y by 1; subtract one from the contents of index register Y.

EOR—logical EXCLUSIVE OR; logically EXCLUSIVE OR the contents of the accumulator with the contents of the specified memory location. Logically EXCLUSIVE ORing the accumulator with the all 1's byte (FF hex) complements the accumulator, changing each 1 bit into a 0 and each 0 bit into a 1.

INC—increment by 1; add one to the contents of the specified memory location. INC cannot be applied to the accumulator.

INX—increment index register X by 1; add one to the contents of index register X.

INY—increment index register Y by 1; add one to the contents of index register Y.

JMP—jump unconditionally; jump (transfer control) to the specified memory address. JMP can be used only with absolute (direct) or indirect addressing. You may want to use JMP rather than a branch instruction because JMP does not require the calculation of a relative offset. Note that JMP really means "load the program counter."

ATTACHING THE LEDS

Attach seven single LEDs to port B of the user 6530 device as described in Table 3-1 and Figure 3-1. Note that there is a gap in the port; bit 6 is not available because the KIM uses it internally.

An LED will light when its cathode is sufficiently negative with respect to its anode. The computer can therefore light an LED either by grounding its cathode (if the anode is tied to +5 V) or by applying +5 V to its anode (if the cathode is grounded). The 6530 output port (like most MOS or TTL devices) can drive the cathodes of LEDs better than the

anodes, so we will use the connections shown in Figure 3-1. Note that a logic 0 from the computer lights the LED; that is, the LED circuit is active-low or uses *negative logic.*

6530 INPUT/OUTPUT PORTS

In Laboratory 2, we used a 6530 I/O port for input. In fact, the programmer can make each bit of the two I/O ports either an input or an output by placing either a 0 (input) or a 1 (output) in the corresponding bit position of the port's data direction register.

Thus the data direction register determines which way data flows, much as a directional signal controls which way traffic moves on a highway or railroad track. The data direction registers themselves occupy memory addresses (see Table 3-2).

Table 3-2

MEMORY ADDRESSES FOR THE I/O PORTS IN THE USER 6530 DEVICE

ADDRESS (HEX)	FUNCTION
1700	Port A
1701	Data direction register for port A
1702	Port B*
1703	Data direction register for port B

*Bit 6 of port B is not available externally.

Typical examples of assigning bits as inputs or outputs are:

1) Storing zero in address 1701 makes all the bits of port A (address 1700) inputs. The required instructions are

```
LDA     #0
STA     $1701
```

Note the use of immediate addressing to load the accumulator with a specific value. What would the instruction LDA 0 do?

2) Storing FF (hex) in address 1703 makes all the bits of port B (address 1702) outputs. The required instructions are

```
LDA     #$FF
STA     $1703
```

We will initialize port B in this way throughout Laboratory 3.

3) Storing 0F (hex) in address 1703 makes bits 4 through 7 of port B (address 1702) inputs and bits 0 through 3 outputs. The required instructions are

```
LDA     #$0F
STA     $1703
```

4) Storing AA (hex)—10101010 binary—in address 1701 makes bits 1, 3, 5, and 7 of port A (address 1700) outputs and bits 0, 2, 4, and 6 inputs. The required instructions are

```
LDA     #$AA
STA     $1701
```

Of course, the assignment of inputs and outputs is clearer if we specify the data direction register's contents in binary rather than in hexadecimal. As with the status register, a hexadecimal value is difficult to interpret, since each bit has a separate meaning. You can use Table 0-1 to convert numbers from binary to hexadecimal for entry into the KIM's memory.

The 6530 device has the following important features:

1) RESET clears the data direction registers, thus making all the bits of both I/O ports inputs. You can check this by resetting the KIM and examining memory addresses 1701 and 1703. This initial state allowed us to perform the experiments in Laboratory 2 without worrying about the data direction registers, as long as we started from reset.

2) The I/O ports can consist of any combination of input and output bits. Thus KIM users can select the numbers and arrangements of input and output lines rather than having to modify their designs to satisfy a fixed arrangement. This is a major advantage of programmable I/O devices; users can modify a fixed board design with software to handle a variety of applications. This flexibility is advantageous to manufacturers as well, since it lets them design, test, and market a single board that users can modify with software for their applications.

3) In a particular application, the initialization routine (starting from reset) must assign the required arrangement of input and output bits. The main program will usually transfer data to or from the I/O ports as if the arrangement were fixed.

PROBLEM 3-1

Write a program that makes port A of the user 6530 device an input port and port B an output port. How could you check this program to see if it had executed correctly? What happens when you reset the computer and run the program? What happens if you then load 00 into memory location 1703? Change memory location 1703 to AA and see what happens.

PROBLEM 3-2

Revise the initialization routine from Problem 3-1 so that it makes bit 0 of port B an output and all other bits of port B inputs. What happens when you reset the computer and run this program?

LIGHTING AN LED

The following program will light the LED attached to bit 3 of port B of the user 6530 device (LED 3, for short).

```
LDA     #$FF            ;MAKE PORT B OUTPUT
STA     $1703
LDA     #%11110111      ;LIGHT LED 3
STA     $1702
BRK
```

Since the LEDs are attached to the I/O port by their cathodes, a 0 turns an LED on and a 1 turns it off. We must assign the I/O lines of port B as outputs before the LEDs will light. Program 3-1 is the hexadecimal version; enter and run it. What happens if you reset the computer afterward?

PROGRAM 3-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A9	LDA	#$FF
0201	FF		
0202	8D	STA	$1703
0203	03		
0204	17		
0205	A9	LDA	#%11110111
0206	F7		
0207	8D	STA	$1702
0208	02		
0209	17		
020A	00	BRK	

Try the following variations:

PROBLEM 3-3

Write a program that lights the LED attached to bit 4 of port B (LED 4) and turns off all the other LEDs. Make the program light only LEDs 2 and 5.

PROBLEM 3-4

Write a program that shows the data from port A on the LEDs attached to port B. Does a switch have to be open or closed for the corresponding LED to light?

PROBLEM 3-5

Write a program that lights every other LED, starting with LED 0. Change the program so that it produces the opposite pattern of lights.

IMPLEMENTING A TIME DELAY

Of course, in real applications we seldom want to turn an output on and leave it on. Usually, we want to leave it on for a specific amount of time. The microprocessor can provide the delay if we make it perform a simple time-wasting procedure such as the following:

1) Load a register with an initial value.
2) Decrement the register until it contains zero.

The program that performs these steps using index register X is

```
        LDX     #COUNT
DLY     DEX
        BNE     DLY
```

This procedure works like telling someone to count down from 10 before opening his or her eyes.

We can calculate the amount of time wasted from the following information:

INSTRUCTION	NUMBER OF TIMES EXECUTED	CLOCK CYCLES PER EXECUTION
LDX # (IMMEDIATE)	1	2
DEX	COUNT	2
BNE	COUNT	2 if no branch, 3 if a branch occurs

Note that BNE takes different numbers of clock cycles, depending on whether the program branches. That is, BNE takes two cycles if the ZERO flag is 1 and no branch occurs; BNE takes three cycles if the ZERO flag is 0 and the program branches back to address DLY.

Table A1-1 contains the execution times for 6502 instructions; you can also find them in Appendix C of the *MCS6500 Microcomputer Family Programming Manual.* The total amount of time the computer spends executing the delay program is

$$[5 \times (\text{COUNT} - 1) + 4 + 2] \times t_C$$

where t_C is the clock period of the KIM. The 5 is the execution time for a DEX (2) and a BNE with a branch (3); the extra 4 is for the last iteration in which no branch occurs and hence BNE takes only two cycles. The final 2 is the execution time for the initial LDX #COUNT instruction.

Since the KIM has a 1-MHz clock (see p. 22 of the *KIM-1 User Manual*), $t_C = 1\ \mu s$. If, for example, COUNT = 10, the amount of time wasted is

$$[5 \times (10 - 1) + 6] \times 1\ \mu s = 51\ \mu s$$

The maximum amount of time this program can waste is

$$[5 \times (256 - 1) + 6] \times 1\ \mu s = 1281\ \mu s \text{ or } 1.28\ ms$$

What value of COUNT produces this maximum delay? Note that the program decrements index register X before it branches.

You can add a delay to Program 3-1 as follows:

```
        LDA   #$FF            ;MAKE PORT B OUTPUT
        STA   $1703
        LDA   #%11110111      ;LIGHT LED 3
        STA   $1702
        LDX   #COUNT          ;DELAY
DLY     DEX
        BNE   DLY
        LDA   #%11111111      ;TURN OFF LED 3
        STA   $1702
        BRK
```

Program 3-2 is the hexadecimal version; we still must assign a numerical value to COUNT. Enter and run Program 3-2 using the value 00 for COUNT (memory location 020B).

PROGRAM 3-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#$FF
0201	FF			
0202	8D		STA	$1703
0203	03			
0204	17			
0205	A9		LDA	#%11110111
0206	F7			
0207	8D		STA	$1702
0208	02			
0209	17			
020A	A2		LDX	#COUNT
020B	COUNT			
020C	CA	DLY	DEX	
020D	D0		BNE	DLY
020E	FD			
020F	A9		LDA	#%11111111
0210	FF			
0211	8D		STA	$1702
0212	02			
0213	17			
0214	00		BRK	

PROBLEM 3-6

Run Program 3-2 repeatedly, dividing COUNT (memory location 020B) in half after each execution; use the sequence 00, 80, 40, 20, 10, 08, 04, 02, 01. What is the smallest value of COUNT for which you can see the LED light?

LENGTHENING THE DELAY

You can lengthen the delay by placing one time-wasting routine inside another (called *nesting* the routines): that is,

```
        LDY   #CT1    ;SET MULTIPLYING FACTOR
DLY1    LDX   #CT2    ;SET DELAY FACTOR
DLY2    DEX
        BNE   DLY2
        DEY
        BNE   DLY1
```

CT1 determines how many times the inner loop (constant CT2) is executed. Determine the execution time for the nested delay. Program 3-3 extends Program 3-2 to provide a nested delay.

PROGRAM 3-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#$FF
0201	FF			
0202	8D		STA	$1703
0203	03			
0204	17			
0205	A9		LDA	#%11110111
0206	F7			
0207	8D		STA	$1702
0208	02			
0209	17			
020A	A0		LDY	#CT1
020B	CT1			
020C	A2	DLY1	LDX	#CT2
020D	CT2			
020E	CA	DLY2	DEX	
020F	D0		BNE	DLY2
0210	FD			
0211	88		DEY	
0212	D0		BNE	DLY1
0213	F8			
0214	A9		LDA	#%11111111
0215	FF			
0216	8D		STA	$1702
0217	02			
0218	17			
0219	00		BRK	

PROBLEM 3-7

If you set CT1 (memory location 020B) to 200 (C8 hex), what value of CT2 (memory location 020D) produces a 10-ms delay? What value of CT2 produces a 100-ms delay?

PROBLEM 3-8

Revise the nested delay so that it counts up instead of down. What value of CT2 produces a 10-ms delay if you set CT1 to -200 (38 hex)?

PROBLEM 3-9

Revise the original delay routine so that it uses memory location 0040 to hold the counter. Calculate how much time the revised routine wastes as a function of the initial value that it stores in the memory location.

CONTROLLING INDIVIDUAL BITS

Often, we want to change the state of one display without affecting unrelated displays that are attached to the same output port. We can do this by taking advantage of the following effects of the common logical functions (see Table 3-3):

1) Logically ANDing a number with a mask clears those bit positions which are 0's in the mask but does not affect those bit positions which are 1's.

2) Logically ORing a number with a mask sets those bit positions which are 1's in the mask but does not affect those bit positions which are 0's.

3) Logically EXCLUSIVE ORing a number with a mask complements (inverts) those bit positions which are 1's in the mask but does not affect those bit positions which are 0's.

In Program 3-2, for example, we could set bit 3 of the accumulator and thus affect only LED 3 by using the instruction ORA #%00001000; that is,

```
020F    09      ORA     #%00001000
0210    08
```

Make this change and run the revised program. Make a similar change in Program 3-3.

PROBLEM 3-10

Write a program that turns LED 4 off, waits for a while, and then turns LED 4 on without affecting any other displays.

PROBLEM 3-11

Write a program that gets the values for the LEDs from memory location 0040, waits for a while, and then complements LEDs 1 and 5 without affecting any other displays.

Table 3-3

EFFECTS OF LOGICAL INSTRUCTIONS

Logical AND ORIGINAL VALUE	MASK VALUE	FINAL VALUE
0	0	0
1	0	0
0	1	0
1	1	1

The final value is 0 if the mask value is 0 and the same as the original value if the mask value is 1.

Logical OR ORIGINAL VALUE	MASK VALUE	FINAL VALUE
0	0	0
1	0	1
0	1	1
1	1	1

The final value is 1 if the mask value is 1 and the same as the original value if the mask value is 0.

Logical EXCLUSIVE OR ORIGINAL VALUE	MASK VALUE	FINAL VALUE
0	0	0
1	0	1
0	1	1
1	1	0

The final value is the complement of the original value if the mask value is 1 and the same as the original value if the mask value is 0.

PROBLEM 3-12

Write a program that gets the values for the LEDs from memory location 0040, turns LEDs 2 and 5 on, waits for a while, turns LED 5 off, waits again, and finally turns LED 2 off and LED 5 on without affecting any other displays.

Remember that you can change a particular bit of the accumulator (bit 5, for example) as follows:

1) Make it 1 with ORA #%00100000.
2) Make it 0 with AND #%11011111.
3) Complement (invert) it with EOR #%00100000.

The last two lines of Table 3-3 show that we can use the EXCLUSIVE OR instruction to complement (invert) the accumulator. All that we must do is make the mask FF (the all 1's byte). The instruction EOR #$FF inverts each bit of the accumulator; that is, it replaces each 0 with a 1 and each 1 with a 0. The common use of inversion is in transferring data to or from I/O devices that operate in negative logic. Many switches, displays (such as the LEDs we have attached to port B of the user 6530 device), and other peripherals operate at inverted logic levels. Most microprocessors have an explicit complement instruction to handle this common situation.

PROBLEM 3-13

Write a program that displays the contents of memory location 0040 on the LEDs attached to port B of the user 6530 device. Make the data appear in the form an observer would expect—that is, an LED should be lit to indicate a 1 and off to indicate a 0.

ESTABLISHING A DUTY CYCLE

We can establish a duty cycle for an LED simply by turning it on and then off for specified periods of time. The following program will do the job:

```
        LDA   #$FF          ;MAKE PORT B OUTPUT
        STA   $1703
CYCLE   LDA   #%11110111    ;LIGHT LED 3
        STA   $1702
        LDY   #CT1          ;DELAY WHILE LIGHT IS ON
DLY1    LDX   #CT2
DLY2    DEX
        BNE   DLY2
        DEY
        BNE   DLY1
        LDA   #%11111111    ;TURN OFF LED 3
        STA   $1702
        LDY   #CT3          ;DELAY WHILE LIGHT IS OFF
DLY3    LDX   #CT4
DLY4    DEX
        BNE   DLY4
        DEY
        BNE   DLY3
        JMP   CYCLE
```

Program 3-4 is the hexadecimal version. Enter and run this program; try various values for the delay constants (CT1, CT2, CT3, and CT4).

PROGRAM 3-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#$FF
0201	FF			
0202	8D		STA	$1703
0203	03			
0204	17			
0205	A9	CYCLE	LDA	#%11110111
0206	F7			
0207	8D		STA	$1702
0208	02			
0209	17			
020A	A0		LDY	#CT1
020B	CT1			
020C	A2	DLY1	LDX	#CT2
020D	CT2			
020E	CA	DLY2	DEX	
020F	D0		BNE	DLY2
0210	FD			
0211	88		DEY	
0212	D0		BNE	DLY1
0213	F8			
0214	A9		LDA	#%11111111
0215	FF			
0216	8D		STA	$1702
0217	02			
0218	17			
0219	A0		LDY	#CT3
021A	CT3			
021B	A2	DLY3	LDX	#CT4
021C	CT4			
021D	CA	DLY4	DEX	
021E	D0		BNE	DLY4
021F	FD			
0220	88		DEY	
0221	D0		BNE	DLY3
0222	F8			
0223	4C		JMP	CYCLE
0224	05			
0225	02			

PROBLEM 3-14

Set CT2 (memory location 020D) = CT4 (memory location 021C) = 00. Start with CT1 (memory location 020B) = CT3 (memory location 021A) = 00 and run

Program 3-4. Then try the following sequence of hexadecimal values for CT1 and CT3: 80, 40, 20, 10, 08, 04, 02, 01. What is the smallest value for which you can see the LED flicker? How many times per second is the LED being turned on and off at this value?

PROBLEM 3-15

Set CT2 = CT4 = 0. Start with CT1 = CT3 = 10 (hex) and run Program 3-4. Try the following pairs of hexadecimal values for CT1 and CT3: (1) CT1 = 1C, CT3 = 04; (2) CT1 = 18, CT3 = 08; (3) CT1 = 08, CT3 = 18; (4) CT1 = 04, CT3 = 1C. Describe how different values affect the brightness and continuity of the LEDs. Compare the effects to those you saw in Problem 3-14.

PROBLEM 3-16

Set CT2 = CT4 = 0 and CT1 = CT3 = 20 (hex). Write a program that flashes the LED on and off for 5 s. Use memory location 0040 as an overall counter and load it initially from the keyboard (before executing the program).

LED displays have the following important characteristics:

1) LEDs have a very short turn-on time, typically only a few microseconds. It is therefore easy to handle many LEDs from one port (i.e., to multiplex them).

2) LEDs dissipate less power and last longer if they are pulsed rather than left on continuously.

3) Display time constants can readily be varied in software to achieve a suitable balance among power dissipation, visibility, and life span. How would you implement such changes in hardware? In many applications the operator may wish to control the displays in order to adapt them to local lighting conditions or to avoid distraction.

4) A microprocessor can easily handle LED displays while performing other tasks, since the LEDs need only be controlled at a very slow rate to satisfy human observers.

KEY POINT SUMMARY

1) The I/O ports in the 6530 Peripheral Interface/Memory device can be either inputs or outputs. Each bit is selected individually to be an input or an output by storing the appropriate value (0 for input, 1 for output) in the corresponding bit position of the data direction register. The data direction registers themselves occupy memory addresses; the user must remember, however, that they are actually located inside the 6530 device and are not connected to any peripherals.

2) By storing the proper values in the data direction registers, the user can vary the numbers and arrangements of inputs and outputs to handle different applications. In most applications, the initialization routine assigns the directions and the rest of the program treats the ports as if the arrangement was fixed.

3) You can implement a delay by having the processor count in a register or memory location. The length of the delay depends on the number of instructions in the program and their execution times. Delay programs can be nested as long as there are registers or memory locations available to hold counters.

4) A particular bit can be cleared, set, or complemented by means of logical operations with appropriate masks. The entire accumulator can be complemented (inverted) by EXCLUSIVE ORing it with the all 1's byte.

5) You can establish a duty cycle by providing appropriate delays after turning the peripheral device on and off.

6) You can easily modify control functions that are implemented in software, since the only changes required are a few constants or instructions in the program.

Laboratory 4

Processing Data Inputs

PURPOSE

To learn how to process data inputs.

PARTS REQUIRED

- Eight switches or pushbuttons attached to port A of the user 6530 device as shown in Figure 2-1. Table 2-1 lists the pins used on the Application Connector. You can replace the single switches with an unencoded rotary or thumbwheel switch. It should have at least eight positions, which can be connected as shown in Figure 4-1.
- Eight switches attached through an encoder to port A of the user 6530 device, as shown in Figure 4-2. This add-on can employ the same switches as the add-on shown in Figure 2-1, since the two are not needed at the same time.
- A 74148 priority encoder (see Table 4-2 and Figure 4-6 for a description of this device).

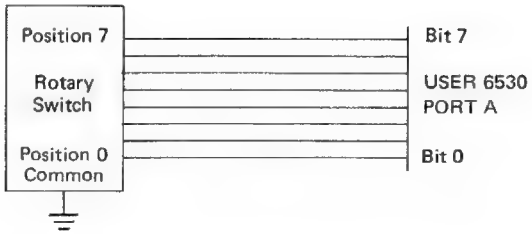


FIGURE 4-1. Connections for an unencoded rotary switch.

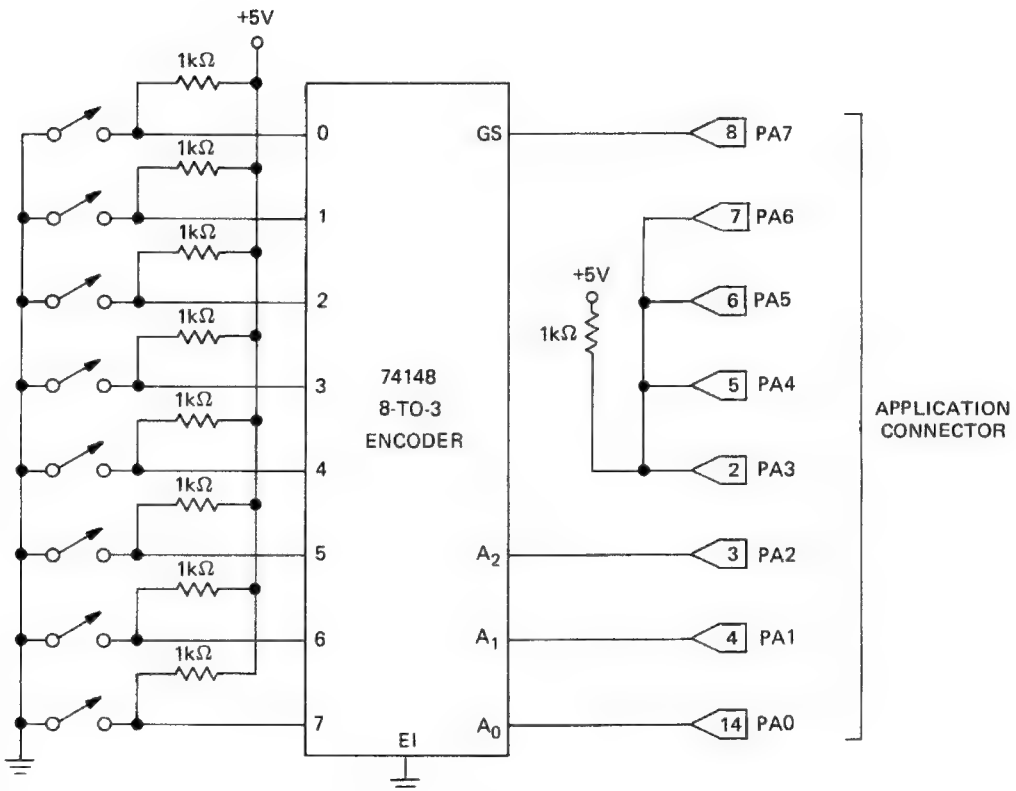


FIGURE 4-2. Attachment of switches and an encoder to port A of the user 6530 device.

REFERENCE MATERIALS

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 369-376.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11-8 through 11-12, 11-39 through 11-60.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 60 (encoders), 274-279 (keyboard input devices), 335-337 (shift and rotate instructions), 346 (unconditional jump instruction).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapter 11.

The TTL Data Book for Design Engineers, Texas Instruments Inc., Dallas, TX, 1976, pp. 6-64 through 6-67 (74121 one-shot), 7-151 through 7-156 (74148 encoder).

WHAT YOU SHOULD LEARN

1) How to wait for a switch to close.
2) How to wait for a switch to close and reopen.
3) How to debounce a switch.
4) How to count switch closures.
5) How to determine the bit position of a switch closure.
6) How to use a TTL encoder.
7) How to make simple tradeoffs between hardware and software.

TERMS

Bounce—move back and forth between states before reaching a final state.

Cross-coupled—when two devices each have their output fed back into the other's input.

Debounce—convert the output from a contact with bounce into a single, clean transition between states.

Debounce time—the amount of time required to debounce a change of state.

Enable—allow an activity to proceed or a device to produce data outputs.

Encoder—a device that produces coded outputs from unencoded inputs. A *priority encoder* accepts only the input with the highest priority if two or more inputs are active simultaneously.

Group select (GS)—a signal that indicates if there is any activity at a particular level or within a group of signals, used to combine levels or groups.

Normal closed (NC)—a switch output that is connected to the common line if the switch is in its marked closed position.

Normal open (NO)—a switch output that is connected to the common line if the switch is in its marked open position.

One-shot (or **monostable multivibrator**)—a device that produces a single pulse of known length in response to a pulse input.

SPDT switch—single-pole, double-throw switch with one common line and two output lines.

6502 INSTRUCTIONS

BEQ—branch if equal to zero; jump over the specified number of memory locations if the ZERO flag is 1; otherwise, proceed to the next instruction in sequence. Note that the ZERO flag is 1 if the last result *was zero.*

CMP—compare memory and accumulator; subtract the contents of the specified memory location from the contents of the accumulator but leave the accumulator unchanged. This instruction affects only the flags.

TAX—transfer accumulator to index register X; transfer the contents of the accumulator to index register X. The accumulator does not change.

TAY—transfer accumulator to index register Y; transfer the contents of the accumulator to index register Y. The accumulator does not change.

TXA—transfer index register X to accumulator; transfer the contents of index register X to the accumulator. Index register X does not change.

TYA—transfer index register Y to accumulator; transfer the contents of index register Y to the accumulator. Index register Y does not change.

HANDLING MORE COMPLEX INPUTS

Normally, we would like the microprocessor to do more than just determine the value of a particular binary input. Rather, we want the processor to handle a series of inputs and convert the input data into useful

forms. The processor should also be able to perform such simple tasks as smoothing the input data and accounting for the time constants of input peripherals. An important consideration is that either software or hardware can perform these tasks. Extra hardware can transform the inputs so that the software can handle them easily. Designers must make tradeoffs based on per-unit cost, development time and cost, reliability, compatibility with other applications, power dissipation, board space, and availability of parts that perform specific functions.

WAITING FOR ANY SWITCH TO CLOSE

Table 4-1 contains the binary inputs resulting from the closure of one of a set of single switches. If all the switches are open, the input data is all 1's (i.e., FF hexadecimal). So the following program will wait until you close any of the switches:

```
WAITC   LDA   $1700    ;GET INPUT DATA
        CMP   #$FF     ;ARE ANY SWITCHES CLOSED?
        BEQ   WAITC    ;NO, WAIT
        BRK
```

Table 4-1

INPUTS RESULTING FROM THE CLOSURE OF INDIVIDUAL SWITCHES

BIT POSITION OF CLOSED SWITCH	INPUT	
	BINARY	HEX
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F
None	11111111	FF

Program 4-1 is the hexadecimal version of the program. Note that the instruction CMP #$FF subtracts FF from the contents of the accumulator and sets the flags appropriately but does not save the result anywhere.

Thus the data from port A (memory address 1700) is still in the accumulator at the end of the program. (Verify this!)

Enter and run Program 4-1. Check to see that it exits if you close any of the switches. What happens if you close several switches at once? What happens if you close switches before executing the program? Note the final contents of the accumulator in each case.

PROGRAM 4-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	AD	WAITC	LDA	$1700
0201	00			
0202	17			
0203	C9		CMP	#$FF
0204	FF			
0205	F0		BEQ	WAITC
0206	F9			
0207	00		BRK	

We can easily add a section that waits until all the switches are open again. All that we must do is branch on the opposite condition. The new section is

```
WAITO   LDA   $1700    ;GET INPUT DATA
        CMP   #$FF     ;ARE ANY SWITCHES CLOSED?
        BNE   WAITO    ;YES, WAIT
```

The complete program now is

```
WAITC   LDA   $1700    ;GET INPUT DATA
        CMP   #$FF     ;ARE ANY SWITCHES CLOSED?
        BEQ   WAITC    ;NO, WAIT
WAITO   LDA   $1700    ;GET INPUT DATA
        CMP   #$FF     ;ARE ANY SWITCHES CLOSED?
        BNE   WAITO    ;YES, WAIT
        BRK
```

Program 4-2 contains the hexadecimal additions to Program 4-1. Enter and run Program 4-2; try the variations in Problems 4-1, 4-2, and 4-3.

PROGRAM 4-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0207	AD	WAITO	LDA	$1700
0208	00			
0209	17			
020A	C9		CMP	#$FF
020B	FF			
020C	D0		BNE	WAITO
020D	F9			
020E	00		BRK	

PROBLEM 4-1

Write a program that waits for switch 5 to be closed and opened while all other switches are open.

PROBLEM 4-2

Write a program that waits for switch 5 to be closed and opened, regardless of the other switches.

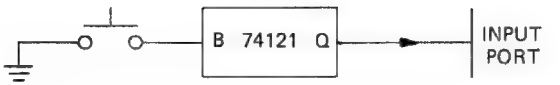
PROBLEM 4-3

Write a program that waits for switch 5 to be closed, opened, and then closed again, regardless of the other switches.

DEBOUNCING A SWITCH

If you run Programs 4-1 and 4-2 several times, you will probably find that the computer often exits before you open the switch. This is because a mechanical switch (or a key on a keyboard) does not open or close cleanly. Instead, the switch bounces back and forth for a while before it settles into its final position. The computer cannot tell the bounce from the opening of the switch, since both result in a logic 1 in the input bit.

The solution to this problem is to debounce the switch. This can be done in hardware with a one-shot (see Figure 4-3) or with cross-coupled NAND gates (see Figure 4-4). But we can also debounce the switch in software at the cost of a few bytes of memory. All we need is a short delay program that waits until the switch stops bouncing. Since the bounce usually lasts less than 1 ms, the following program will do the job:



74121 Function Table

INPUTS			OUTPUTS	
A1	A2	B	Q	$\overline{Q}$
L	X	H	L	H
X	L	H	L	H
X	X	L	L	H
H	H	X	L	H
H	↓	H	⎍	⎍
↓	H	H	⎍	⎍
↓	↓	H	⎍	⎍
L	X	↑	⎍	⎍
X	L	↑	⎍	⎍

X = irrelevant

74121 Pin Assignments

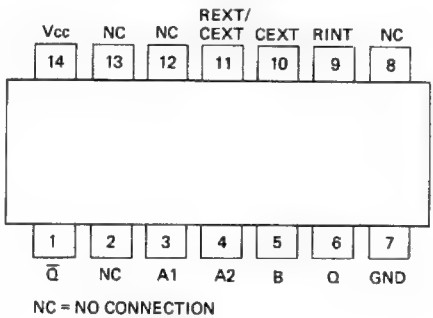


FIGURE 4-3. Debouncing a switch with a one-shot.

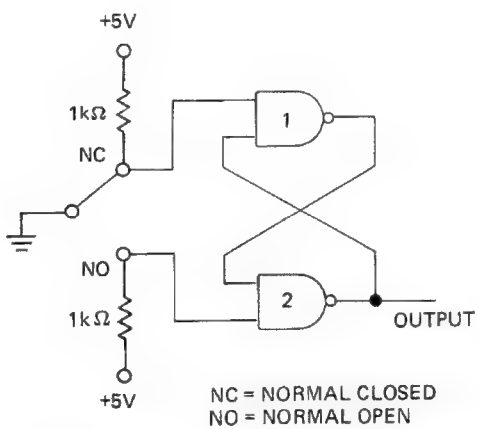


FIGURE 4-4. Debouncing a switch with cross-coupled NAND gates.

```
WAITC   LDA   $1700    ;GET INPUT DATA
        CMP   #$FF     ;ARE ANY SWITCHES CLOSED?
        BEQ   WAITC    ;NO, WAIT
        LDX   #$C8     ;DELAY 1 MS TO DEBOUNCE
DLY     DEX
        BNE   DLY
WAITO   LDA   $1700    ;GET INPUT DATA
        CMP   #$FF     ;ARE ANY SWITCHES CLOSED?
        BNE   WAITO    ;YES, WAIT
        BRK
```

Program 4-3 contains the required hexadecimal additions to Program 4-1. We obtained the value C8 (200 decimal) by calculating the number of iterations required to waste 1 ms (1000 μs). Verify this value from the timing equation in Laboratory 3.

PROGRAM 4-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0207	A2		LDX	#$C8
0208	C8			
0209	CA	DLY	DEX	
020A	D0		BNE	DLY
020B	FD			
020C	AD	WAITO	LDA	$1700
020D	00			
020E	17			
020F	C9		CMP	#$FF
0210	FF			
0211	D0		BNE	WAITO
0212	F9			
0213	00		BRK	

PROBLEM 4-4

Run the combination of Programs 4-1 and 4-3 with shorter delays and determine whether you can still see the bounce. Divide the delay constant (memory location 0208) in half after each trial (an approximate sequence is C8, 64, 32, 19, 0D, 07, 04, 02, 01).

The various debouncing methods represent a tradeoff between hardware and software. The software delay costs very little, since the program is simple and requires only a few bytes of memory. On the other hand, it

occupies the processor completely, preventing it from performing other tasks. Hardware debouncing frees the processor for other work but requires an additional part and more connections. Note that several different tasks may use the same hardware or software at different times.

COUNTING CLOSURES

We can keep a running count (in memory location 0040) of the number of switches closed as follows, assuming that we close only one switch at a time:

1) Add the instructions

```
        INC   $40       ;INCREMENT NUMBER OF CLOSURES
        LDX   #$C8      ;DELAY 1 MS TO DEBOUNCE OPENING
DLY1    DEX
        BNE   DLY1
        JMP   WAITC     ;WAIT FOR NEXT CLOSURE
```

to the end of the program as shown in Program 4-4.

2) Clear memory location 0040 (from the keyboard) before executing the program.

PROGRAM 4-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0213	E6		INC	$40
0214	40			
0215	A2		LDX	#$C8
0216	C8			
0217	CA	DLY1	DEX	
0218	D0		BNE	DLY1
0219	FD			
021A	4C		JMP	WAITC
021B	00			
021C	02			

We could use BEQ WAITC instead of JMP WAITC to conclude this program (why?), but we used JMP because it does not need a relative offset. Instead, JMP uses absolute addressing. Note that this program

never returns control to the monitor. You will have to reset the computer to examine the contents of memory location 0040.

PROBLEM 4-5

Write a program that returns control to the monitor after it counts the number of switch closures initially entered into memory location 0040. Assume that only one switch is ever closed at a time.

PROBLEM 4-6

Write a program that counts the number of times that switch 5 is closed. Use memory location 0041 for the counter.

PROBLEM 4-7

Write a program that counts the number of times that switches 2 and 5 are closed. Use memory location 0040 as the counter for switch 2 and memory location 0041 as the counter for switch 5. Assume that only one switch is ever closed at a time, so that the program simply has to wait for all switches to be open rather than waiting specifically for the opening of the switch that was closed.

If you find that the count is erratic, try lengthening the delay by using either index register Y or one or two memory locations. Remember to delay both after the switch is closed and after it is opened. If you use a 16-bit counter in two memory locations, you can count up to zero with the following sequence of instructions:

```
DLY     INC     LSCNT     ;COUNT UP LESS SIGNIFICANT BYTE
        BNE     DLY       ;DONE IF NO CARRY
        INC     MSCNT     ;CARRY TO MORE SIGNIFICANT BYTE
        BNE     DLY
```

Since INC does not affect the CARRY flag, we cannot use that flag to tell whether incrementing the less significant byte has caused a carry (which must then be added to the more significant byte). However, INC does affect the ZERO flag and so we can recognize a carry by checking to see if the result is zero. It can only be zero if the byte was previously FF (and hence a carry has occurred). Decrementing a 16-bit counter is more difficult, since we can only recognize the need for a borrow by determining when LSCNT is initially zero. Try writing the sequence of instructions that decrements a 16-bit counter stored in two memory locations.

IDENTIFYING THE SWITCH

Remember that Table 4-1 contains the binary inputs formed when individual switches are closed. What are the inputs when several switches are closed at once? In Table 4-1, the bit that is zero identifies the switch (i.e., bit 0 is 0 for switch 0, bit 1 for switch 1, and so on). The problem is how to determine which bit is zero. A simple method is as follows (see Figure 4-5 for a flowchart).

Step 1) SWITCH NUMBER = 0
DATA = input from switches

Step 2) Shift DATA right one bit. If the CARRY is zero, the program is finished.

Step 3) SWITCH NUMBER = SWITCH NUMBER + 1
Go to step 2.

A program to implement this method is

```
        LDY   #0        ;SWITCH NUMBER = ZERO
SRCHS   LSR   A         ;IS NEXT SWITCH CLOSED?
        BCC   DONE      ;YES, DONE
        INY             ;NO, SWITCH NUMBER = SWITCH NUMBER + 1
        JMP   SRCHS
DONE    BRK
```

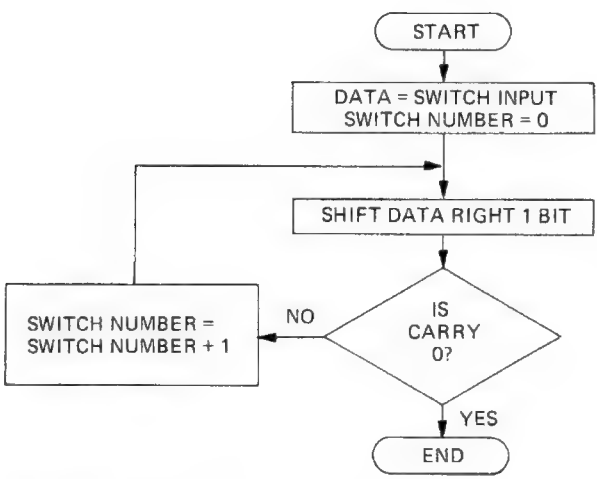


FIGURE 4-5. Flowchart for switch identification.

Index register Y contains the switch number at the end of the program.

An alternative approach uses different initial conditions to eliminate the unconditional jump instruction; that is,

```
        LDY   #$FF      ;SWITCH NUMBER = -1
SRCHS   INY             ;SWITCH NUMBER = SWITCH NUMBER + 1
        LSR   A         ;IS NEXT SWITCH CLOSED?
        BCS   SRCHS     ;NO, KEEP LOOKING
        BRK
```

Which approach do you prefer, and why?

The entire program for identifying a switch consists of the following sections:

1) Wait for any switch to be closed.
2) Wait 1 ms to debounce the switch.
3) Identify the switch by shifting the input and counting until a zero bit is found.

The assembly language program is

```
WAITC   LDA   $1700     ;GET INPUT DATA
        CMP   #$FF      ;ARE ANY SWITCHES CLOSED?
        BEQ   WAITC     ;NO, WAIT
        LDX   #$C8      ;YES, DELAY 1 MS TO DEBOUNCE
DLY     DEX
        BNE   DLY
        LDY   #$FF      ;SWITCH NUMBER = -1
SRCHS   INY             ;SWITCH NUMBER = SWITCH NUMBER + 1
        LSR   A         ;IS NEXT SWITCH CLOSED?
        BCS   SRCHS     ;NO, KEEP LOOKING
        STY   $41       ;YES, SAVE SWITCH NUMBER
        BRK
```

You can see the switch number by examining memory location 0041 (or index register Y) after executing the program. Program 4-5 is a complete hexadecimal version of this program, but note that memory locations 0200 through 020B are the same as in Programs 4-3 and 4-4.

Enter Program 4-5 into memory and try it for each switch individually. What happens if you close more than one switch before executing the program? Which closure does the program identify and why?

PROGRAM 4-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	AD	WAITC	LDA	$1700
0201	00			
0202	17			
0203	C9		CMP	#$FF
0204	FF			
0205	F0		BEQ	WAITC
0206	F9			
0207	A2		LDX	#$C8
0208	C8			
0209	CA	DLY	DEX	
020A	D0		BNE	DLY
020B	FD			
020C	A0		LDY	#$FF
020D	FF			
020E	C8	SRCHS	INY	
020F	4A		LSR	A
0210	B0		BCS	SRCHS
0211	FC			
0212	84		STY	$41
0213	41			
0214	00		BRK	

PROBLEM 4-8

Revise Program 4-5 so that it always identifies the highest numbered switch that is closed. **Hint**: Shift left and decrement the counter instead of shifting right and incrementing, but remember to initialize the counter appropriately.

PROBLEM 4-9

Revise Program 4-5 so that it checks the switches only once, identifies the highest numbered switch that is closed if it finds any closed, and places FF in memory location 0041 if it finds none closed. What would happen if this program checked the switches while one was bouncing? How could you solve this problem? **Hint**: If the program finds all the switches open, have it wait for 1 ms and examine the switches again.

Write a general program that only accepts the input from the switches if it remains the same after a 1-ms delay. That is, the program should keep checking the switches until two readings separated by 1 ms produce the same input.

Table 4-2

FUNCTION TABLE FOR 74148 ENCODER*

INPUTS (X = irrelevant)									OUTPUTS				
EI	0	1	2	3	4	5	6	7	A_2	A_1	A_0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

*H = high or logic 1, L = low or logic 0.

USING A HARDWARE ENCODER

The 74148 8-to-3 priority encoder produces a 3-bit output in negative logic that identifies the highest priority input that is active (low). Table 4-2 is a function table for the device and Figure 4-6 contains its pin assignments. Note the following features of the 74148 encoder:

1) The outputs (A_2, A_1, A_0) are the logical complement of the highest-priority input that is active. For example, the outputs are 0, 1, 0 if input 5 (101 binary) is the highest-priority input that is active (low).

2) The ENABLE IN (EI) input and the ENABLE OUT (EO) output are used to combine encoders to handle more than eight inputs. If EI is high (indicating activity at a level higher than the entire encoder), all the outputs are high. If EI is low (indicating no higher activity) but there is no active input to this encoder, EO is low, thus enabling encoders of lower priority.

3) The GROUP SELECT (GS) output is low if the encoder is enabled and has an active input. GS thus differentiates between the case in which only input 0 is active (the bottom line of Table 4-2) and the cases in which the entire encoder is disabled (the top line of Table 4-2) or all the inputs are inactive (the second line of Table 4-2). Note that A_2, A_1, and A_0 are all 1's in all three cases.

Connect the encoder as described in Table 4-3. The following program will identify the highest numbered switch that is closed before the program is executed (the switch number ends up in the accumulator and in memory location 0041).

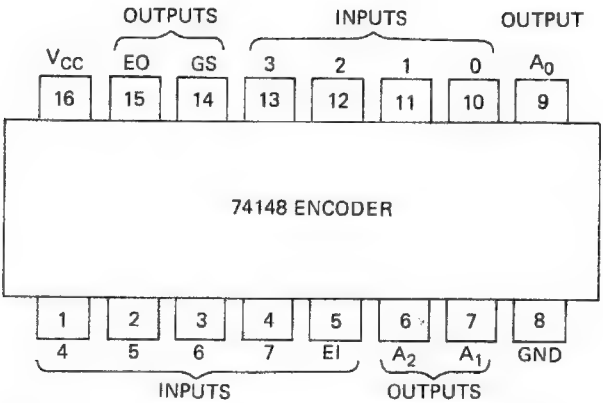


FIGURE 4-6. Pin assignments for the 74148 encoder.

Table 4-3

CONNECTIONS FOR 74148 ENCODER

PIN NUMBER	DESIGNATION	CONNECTION
1	Input 4	Switch 4
2	Input 5	Switch 5
3	Input 6	Switch 6
4	Input 7	Switch 7
5 (EI)	Enable in	Ground
6 (A_2)	Output 2	User 6530 pin PA2
7 (A_1)	Output 1	User 6530 pin PA1
8	Ground	Ground
9 (A_0)	Output 0	User 6530 pin PA0
10	Input 0	Switch 0
11	Input 1	Switch 1
12	Input 2	Switch 2
13	Input 3	Switch 3
14 (GS)	Group select	User 6530 pin PA7
15 (EO)	Enable out	No connection
16	V_{cc}	+5 V

```
LDA     $1700           ;GET SWITCH DATA
EOR     #$FF            ;INVERT LOGIC
AND     #%00000111      ;MASK SWITCH BITS
STA     $41             ;SAVE SWITCH NUMBER
BRK
```

The instruction EOR #$FF inverts the logic levels of the inputs by complementing each bit. Remember, the encoder (like many TTL devices) operates in negative logic. The hexadecimal version of this program is given as Program 4-6. Enter it and run it for several different switch closures. What happens if you run Program 4-6 with more than one switch closed?

PROGRAM 4-6

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	AD	LDA	$1700
0201	00		
0202	17		
0203	49	EOR	#$FF
0204	FF		
0205	29	AND	#%00000111
0206	07		
0207	85	STA	$41
0208	41		
0209	00	BRK	

PROBLEM 4-10

To determine if any switches are closed, the program must examine the GS line attached to bit position 7. Revise Program 4-6 so that it examines the GS line and stores either the switch number or FF (if no switches are closed) in memory location 0041. Note that we have grounded EI (see Table 4-3), so the encoder is always enabled.

PROBLEM 4-11

What would the input be if you inverted the switch connections (i.e., connected switch 7 to encoder input 0, and so on)? Write a program that places the switch number in memory location 0041 in this case. How does the inversion affect the priority of the switches?

Obviously, a hardware encoder makes the software simpler and faster and saves input bits (since it uses 4 rather than 8). On the other hand, the encoder increases the parts count, dissipates power, requires extra connections (which reduce reliability), and uses board space. In low-volume applications, you can justify the cost of extra hardware if it greatly simplifies the software. In high-volume applications, you must keep repeated hardware costs as low as possible.

KEY POINT SUMMARY

1) A mechanical switch requires a relatively long time to settle into a new position. You can either introduce a delay during which the processor does not examine the switch or you can add hardware that smooths the transition. Mechanical components typically have much longer time constants than do electrical components. The interface between the components must account for this difference.

2) Inputs must usually be converted into a convenient form before they can be processed. Either hardware or software can perform this conversion.

3) Timing and code conversion are two common functions that can be performed either in hardware or in software. Hardware implementations reduce the amount and complexity of the required software; this usually simplifies system development, particularly if the designer is more familiar with hardware than with software. Software implementations reduce the number of parts, save board space, and increase reliability.

4) Many factors affect tradeoffs between software and hardware. Among these are the cost and availability of parts, designer experience, product volume, amount of memory available, amount of board space, and performance requirements. Remember the following considerations:

a) Software costs are incurred only once, whereas hardware costs are repeated for each system produced. Thus, high-volume products should have more software and less hardware than low-volume products.

b) A single processor can perform many software tasks, particularly if they involve slow mechanical components. External hardware, on the other hand, is more difficult to share, even among similar tasks.

c) Certain tasks, such as switch and keyboard encoding, display decoding, and serial/parallel interfacing, are so common that special hardware is available to handle them at very low cost. Hardware for less common tasks, even if their complexity is comparable, may be far more expensive.

Laboratory 5

Processing Data Outputs

PURPOSE

To learn how to process data outputs.

PARTS REQUIRED

None.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502*, Howard W. Sams, Indianapolis, IN, 1980, pp. 139-153.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 205-208, 377-378.

L. A. Leventhal, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 7-4 to 7-6, 11-65 to 11-75.

L. J. Scanlon, *6502 Software Design*, Howard W. Sams, Indianapolis, IN, 1980, pp. 36-37, 58-61, 107-113, 237-241.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.*, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 58-59

(decoders), 168-170 (index register), 230-237 (practical interfacing considerations), 338-340 (compare instructions), 360-367 (indexed addressing).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapter 10.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapter 3.

TTL Data Book for Design Engineers, 2nd ed., Texas Instruments, Inc., Dallas, TX, 1976, pp. 7-22 through 7-34 (7447 through 7449 seven-segment decoder/drivers), 7-148 through 7-150 (74145 BCD-to-decimal decoder/driver).

WHAT YOU SHOULD LEARN

1) How seven-segment displays are organized and connected.
2) How to activate the on-board displays and light the segments.
3) How to turn the displays on or off for specified periods of time.
4) How to convert data into the seven-segment code required to form characters on the displays.
5) How and when to use lookup tables.
6) How indexed addressing works and how it is used.
7) How to count on displays.
8) The advantages and disadvantages of lookup tables.
9) The tradeoffs between hardware and software approaches to decoding.

TERMS

Absolute indexed addressing—a form of indexed addressing in which the instruction contains a full 16-bit base address.

Array—a collection of related data items, usually stored in consecutive memory addresses (also called a *block*).

Base address—the address in memory at which an array or table starts. Also called *starting address* or *base*.

Blanking input—an input that turns off the elements in a display.

Common-anode display—a multiple display in which signals are applied to the cathodes of the individual displays and the anodes are tied together to the power supply; uses negative logic (i.e., a logic 0 lights a display).

Common-cathode display—a multiple display in which signals are applied to the anodes of the individual displays and the cathodes

are tied together to ground; uses positive logic (i.e., a logic 1 lights a display).

Decoder—a device that produces unencoded outputs from coded inputs.

Effective address—the actual address used by an instruction to perform its overall function.

Endless loop (or **jump-to-self) instruction**—an instruction that transfers control to itself, thus executing indefinitely (or until a hardware signal interrupts it).

Index—a data item used to identify a particular element of an array or table.

Index register—a register that can be used to modify memory addresses.

Indexed addressing—an addressing method in which the address included in the instruction is modified by the contents of an index register to determine the effective address (the actual address used in the overall operation).

Lookup table—an array of data organized so that the answer to a problem may be determined merely by selecting the correct entry (without any calculations).

Ripple blanking—blanking all leading or trailing displays by having each one indicate to its successor whether it is blank.

Seven-segment code—the code required to form decimal digits or other characters on a seven-segment display.

Seven-segment display—a display made up of seven separately controlled elements that can form representations of decimal digits or other characters.

Zero-page indexed addressing—a form of indexed addressing in which the instruction contains only an address on page zero. That is, zero is implied as the more significant byte of the base address and need not be included explicitly in the instruction.

6502 INSTRUCTIONS

CPX—compare memory and index register X; subtract the contents of the specified memory location from index register X, but leave index register X unchanged. This instruction affects only the flags. The addressing modes allowed are immediate, absolute (direct), and zero page (direct).

CPY—compare memory and index register Y; subtract the con-

tents of the specified memory location from index register Y, but leave index register Y unchanged. This instruction affects only the flags. The addressing modes allowed are immediate, absolute (direct), and zero page (direct).

HANDLING MORE COMPLEX OUTPUTS

In real applications, we want the microprocessor to do more than merely turn a binary output on or off. Rather, we want the processor to produce a sequence of outputs and convert the output data into the forms that peripherals require. The processor should also be able to time the outputs properly.

As with the handling of inputs in Laboratory 4, either software or hardware can process the outputs. The designer must make tradeoffs to satisfy the requirements of a particular application. Furthermore, the designer may be able to make tradeoffs between execution time and memory usage. One simple way to perform a calculation is to use a table that contains all possible results. Now all the program must do is select the correct entry from the table, just as a person could use a book of tables to determine the value of a complex mathematical function. This method (called *table lookup*) is fast and easy to implement but usually requires more memory than an explicit calculation.

USING THE ON-BOARD SEVEN-SEGMENT DISPLAYS

We will use the on-board seven-segment displays as an example of an output device that requires parallel data, timing, and code conversion. Figure 5-1 shows the arrangement of the individual LEDs. The displays are common-anode (see Figure 5-2b) with inverting transistor drivers so that a logic 1 at the output port lights a segment. Figure 5-3 shows how the segments are connected, and Figure 5-4 shows how we have numbered the

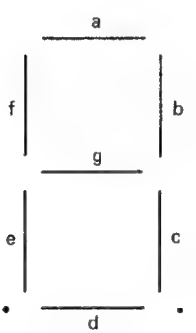


FIGURE 5-1. Seven-segment display.

six displays. This numbering corresponds with that used in the schematic of the KIM keyboard and display (see Figure 5-5 and Figure 3.5 of the *KIM-1 User Manual*).

The KIM's seven-segment displays are connected to the CPU through a 6530 Peripheral Interface/Memory device (identified as 6530-002 in the

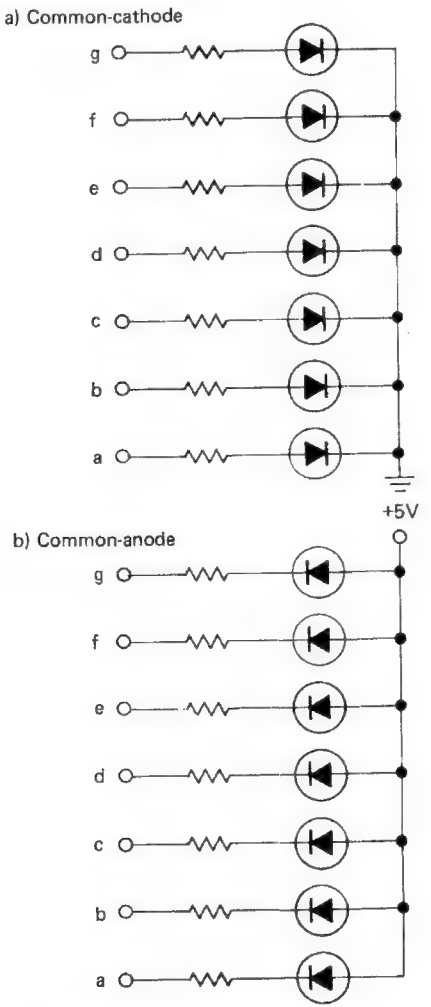


FIGURE 5-2. Alternative connections for seven-segment displays.

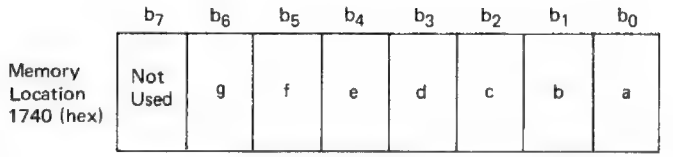


FIGURE 5-3. Segment connections for the on-board displays.

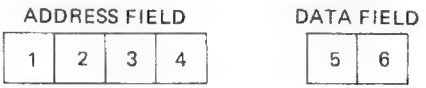


FIGURE 5-4. Numbering of the on-board displays.

schematics) that occupies memory addresses 1740 through 1743. The segments are connected to port A (address 1740) as indicated in Figure 5-3 and shown in Figure 5-5; bit 7 is not used. The commons on the displays are connected to port B (address 1742) through a decoder as shown in Figure 5-5; only bits 1 through 4 of the port are used. So, assigning numbers to the displays as in Figure 5-4 and Table 5-1, we can activate a particular display by storing a value from Table 5-2 in memory location 1742. The following sequence of instructions activates a particular display (defined by ACTIVE) and lights a particular set of segments (defined by DATA).

```
LDA     #ACTIVE     ;ACTIVATE A DISPLAY
STA     $1742
LDA     #DATA       ;SEND IT DATA
STA     $1740
```

Table 5-1

NUMBERING OF THE ON-BOARD DISPLAYS

DISPLAY	IDENTIFICATION NUMBER	PART NUMBER IN SCHEMATIC
Address Digit 1 (MSD)	1	U18
Address Digit 2	2	U19
Address Digit 3	3	U20
Address Digit 4 (LSD)	4	U21
Data Digit 1 (MSD)	5	U22
Data Digit 2 (LSD)	6	U23

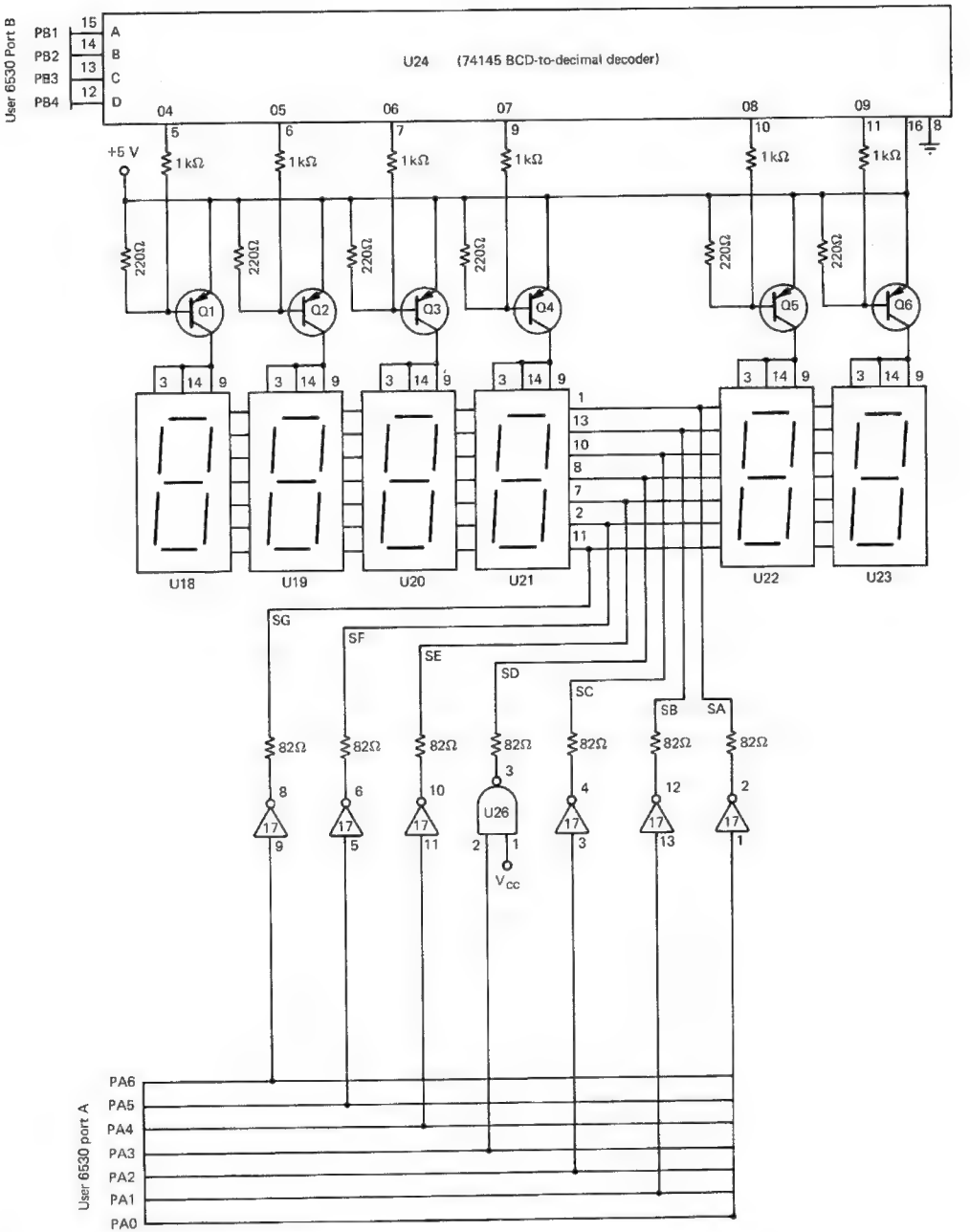


FIGURE 5-5. Schematic of the on-board seven-segment displays. (Notes: Q1 through Q6 are 2N5375 PNP transistors. U17 is a 7406 hex inverter with open-collector outputs. U26 is a 7438 quad NAND gate with open-collector outputs. U18 through U23 are HP5082/7731 common-anode, right-decimal .3″ (7.62 mm) seven-segment displays or equivalent.)

Table 5-2

OUTPUTS FOR ACTIVATING DISPLAYS
(MEMORY LOCATION 1742)

DISPLAY NUMBER	OUTPUT (BINARY)	OUTPUT (HEX)
1	00001000	08
2	00001010	0A
3	00001100	0C
4	00001110	0E
5	00010000	10
6	00010010	12

Note the following:

1) You must activate the proper display besides sending it the appropriate data. The activation involves storing a value from Table 5-2 in memory location 1742. You can activate only one display at a time, since only one of the decoder outputs can be active at a time. See Laboratory F or pp. 7-148 through 7-150 of the *TTL Data Book for Design Engineers, 2nd ed.* (Texas Instruments, Inc., Dallas, TX, 1976) for a description of the 74145 BCD-to-decimal decoder/driver.
2) You can send the activated display data by storing it in memory location 1740.

Table 5-3 contains the hexadecimal outputs required to light the various segments individually. Remember that a logic 1 lights a segment. For example, the following program will light segment f of the leftmost address display (display #1):

Table 5-3

OUTPUTS FOR LIGHTING SEGMENTS
(MEMORY LOCATION 1740)

SEGMENT LIT	OUTPUT(BINARY)	OUTPUT (HEX)
g	01000000	40
f	00100000	20
e	00010000	10
d	00001000	08
c	00000100	04
b	00000010	02
a	00000001	01

```
LDA     #$FF        ;MAKE PORT A OUTPUT
STA     $1741
LDA     #$08        ;ACTIVATE LEFTMOST DISPLAY
STA     $1742
LDA     #$20        ;LIGHT SEGMENT F
STA     $1740
BRK
```

The monitor program makes port B of the keyboard/display 6530 into an output port, so we only have to initialize port A. Storing FF in memory location 1741 loads the data direction register for port A with all 1's and hence makes it an output port. Be careful; both RESET and the BRK instruction turn port A into an input port. Program 5-1 is the hexadecimal version. We have arbitrarily cleared all the unused bit positions in addresses 1740 and 1742.

To light several segments at the same time, we must set all the corresponding bit positions. For example:

SEGMENTS LIT	OUTPUT (BINARY)	OUTPUT (HEX)
f and g	01100000	C0
d and c	00011000	18
a and b	00000011	03
a, b, and c	00000111	07
a, d, and f	00101001	29

Enter and run Program 5-1. What happens? The problem is that the monitor takes over the displays for its own purposes as soon as the BRK instruction is executed. Thus what we see is the final value of the program counter (0211) and the contents of that address. Program 5-1 can retain control of the displays if you conclude it with an endless loop (i.e., an instruction that jumps to itself).

```
020F    4C      HERE    JMP     HERE
0210    0F
0211    02
```

The revised program will run forever, so you will have to reset the computer to return control to the monitor.

PROGRAM 5-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A9	LDA	#$FF
0201	FF		
0202	8D	STA	$1741
0203	41		
0204	17		
0205	A9	LDA	#$08
0206	08		
0207	8D	STA	$1742
0208	42		
0209	17		
020A	A9	LDA	#$20
020B	20		
020C	8D	STA	$1740
020D	40		
020E	17		
020F	00	BRK	

PROBLEM 5-1

Write a program that lights segment g of the leftmost address display (display 1).

PROBLEM 5-2

Write a program that lights segment g of the rightmost address display (display 4).

PROBLEM 5-3

Write a program that lights segments e and g of the rightmost data display (display 6). What letter is formed?

ADDING A DELAY

We can easily leave the display on for a specified amount of time by having the computer execute a delay routine before returning control to the monitor:

```
        LDY     #CT1        ;SET MULTIPLYING FACTOR
DLY1    LDX     #CT2        ;SET DELAY FACTOR
DLY2    DEX
        BNE     DLY2
        DEY
        BNE     DLY1
```

Program 5-2 is the hexadecimal version of the additions required to place a delay at the end of Program 5-1. Enter the changes and run the program with CT1 (memory location 0210) = CT2 (memory location 0212) = 00.

PROGRAM 5-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
020F	A0		LDY	#CT1
0210	CT1			
0211	A2	DLY1	LDX	#CT2
0212	CT2			
0213	CA	DLY2	DEX	
0214	D0		BNE	DLY2
0215	FD			
0216	88		DEY	
0217	D0		BNE	DLY1
0218	F8			
0219	00		BRK	

PROBLEM 5-4

Leaving CT2 = 00, run Program 5-2 repeatedly using the following values for CT1 (memory location 0210): 80, 40, 20, 10, 08, 04, 02, 01. What is the smallest value of CT1 for which you can see the LED light?

PROBLEM 5-5

Revise Program 5-2 so that it shows the letter H on the rightmost display (display 6).

SEVEN-SEGMENT CODE CONVERSION

We can form any decimal digit on a seven-segment display. Table 5-4 contains the required codes. The problem is how to convert a decimal digit to

a seven-segment code. Certainly, the values in Table 5-4 are not related to one another in any obvious way.

Table 5-4

DECIMAL-TO-SEVEN-SEGMENT CONVERSION TABLE FOR THE ON-BOARD DISPLAYS

	SEVEN-SEGMENT CODE	
DECIMAL DIGIT	HEX	BINARY
0	3F	00111111
1	06	00000110
2	5B	01011011
3	4F	01001111
4	66	01100110
5	6D	01101101
6	7D	01111101
7	07	00000111
8	7F	01111111
9	6F	01101111

One approach would be to use Boolean algebra to simplify Table 5-4. We could then transform the conversion into a series of logical ANDs and ORs. This is how we would implement the conversion with logic gates.

A simpler approach, however, is to place Table 5-4 in memory and use it as a lookup table. The program can then perform the conversion as follows:

1) Calculate the address of the desired code by adding the starting (or *base*) address of the table to the element number (or *index*).
2) Obtain the code by loading it from the calculated address.

Our first thought might be to calculate the address with an addition instruction and then try to use that address in a load instruction. However, the 6502 microprocessor provides us with a much simpler way to implement the conversion. We can use the indexed addressing mode in which the processor first adds the contents of an index register to the address included in the instruction. It then uses the sum as the address it needs to execute the instruction. We refer to the address included in the instruction as the *base address*, the contents of the index register as the *index*, and the sum (the actual address used to perform the overall operation) as the *effective address*. The effective address calculated in the indexed addressing mode is exactly what we need to select a code from our lookup table.

The following program uses indexed addressing to convert a decimal digit in memory location 0040 into a seven-segment code in memory location 0041.

```
LDX    $40        ;GET DATA
LDA    $0380,X    ;GET SEVEN-SEGMENT CODE FROM TABLE
STA    $41        ;SAVE RESULT
BRK
```

This program assumes that we have placed the seven-segment code table in memory starting at address 0380. The table thus does not interfere with any of our programs or with the area used by the monitor for temporary storage (see Figure A5-1 for a map of the microcomputer's memory). Note that you must enter both Program 5-3 (starting at address 0200) and Table 5-4 (starting at address 0380) into memory.

PROGRAM 5-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A6	LDX	$40
0201	40		
0202	BD	LDA	$0380,X
0203	80		
0204	03		
0205	85	STA	$41
0206	41		
0207	00	BRK	
0380	3F	0	
0381	06	1	
0382	5B	2	
0383	4F	3	
0384	66	4	
0385	6D	5	
0386	7D	6	
0387	07	7	
0388	7F	8	
0389	6F	9	

We use the absolute indexed addressing mode for LDA, since the table is not located on page zero. In the course of executing LDA with

indexed addressing, the processor both calculates the required address and loads the data from memory.

Program 5-3 works as follows (assuming that memory location 0040 contains 06):

1) LDX $40 loads the data (06) into index register X.
2) LDA $0380,X first calculates the effective address by adding the contents of index register X (06) to the base address included in the instruction (0380). The sum of base and index is 0380 + 06 = 0386. The processor then loads the accumulator from address 0386, which contains 7D, the code that forms a 6 on one of the on-board seven-segment displays.

To make the result easier to observe, add the following instructions that show it on the leftmost display:

```
        LDX   #$FF      ;MAKE PORT A OUTPUT
        STX   $1741
        STA   $1740     ;SEND RESULT TO DISPLAYS
        LDA   #$08      ;ACTIVATE LEFTMOST DISPLAY
        STA   $1742
HERE    JMP   HERE
```

The required hexadecimal additions are:

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0207	A2		LDX	#$FF
0208	FF			
0209	8E		STX	$1741
020A	41			
020B	17			
020C	8D		STA	$1740
020D	40			
020E	17			
020F	A9		LDA	#$08
0210	08			
0211	8D		STA	$1742
0212	42			
0213	17			
0214	4C	HERE	JMP	HERE
0215	14			
0216	02			

It does not matter if you activate the display after storing the data in the output port, as long as the lag is too short to be visible. Note that you can form some letters as well as decimal digits on seven-segment displays (see Table 5-5). If you run the program with the additions, you will probably notice that the displays are much brighter than usual. Explain the increased brightness. **Hint**: Remember the results of Problem 3-15.

PROBLEM 5-6

Extend Program 5-3 so that it converts hexadecimal digits into seven-segment codes using Table 5-6.

PROBLEM 5-7

Revise Program 5-3 to use index register Y instead of index register X. What difference would it make if the table were on page zero? Note that there is a zero page indexed mode for LDA with index register X. Is there one with index register Y? Do you think that you would normally store this kind of table on page zero? Explain your answer.

Table 5-5

SEVEN-SEGMENT CODES FOR LETTERS AND OTHER CHARACTERS

SYMBOL	SEVEN-SEGMENT CODE (HEX)
Capital letters	
A	77
C	39
E	79
F	71
H	76
I	06
J	1E
L	38
O	3F
P	73
U	3E
Y	6E
Lowercase letters	
b	7C
c	58
d	5E
h	74
i	04

Table 5-5 (continued)

SEVEN-SEGMENT CODES
FOR LETTERS AND OTHER CHARACTERS

SYMBOL	SEVEN-SEGMENT CODE (HEX)
n	54
o	5C
r	50
t	78
u	1C
Other characters	
?	53
- (hyphen)	40
_ (underscore)	08

Table 5-6

HEXADECIMAL-TO-SEVEN-SEGMENT
CONVERSION TABLE FOR THE ON-BOARD DISPLAYS

HEXADECIMAL DIGIT	SEVEN-SEGMENT CODE (HEX)
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F
A	77
b (lowercase)	7C
C	39
d (lowercase)	5E
E	79
F	71

PROBLEM 5-8

The monitor also uses a table to form hexadecimal digits on the displays. Revise Program 5-3 to use the monitor conversion table starting at memory address 1FE7. This table is in ROM, so you do not have to enter it. How does it differ

from Table 5-6? Does its placement in memory support your answer to the last question in Problem 5-7?

COUNTING ON THE DISPLAYS

We can use Table 5-6 to count in hexadecimal on the displays. The following program will count up on the leftmost address display (display 1):

```
        LDA   #$FF      ;MAKE PORT A OUTPUT
        STA   $1741
        LDX   #0        ;START THE COUNT AT ZERO
        LDA   #$08      ;ACTIVATE LEFTMOST DISPLAY
        STA   $1742
DSPLY   LDA   $0380,X   ;GET SEVEN-SEGMENT CODE FOR COUNT
        STA   $1740     ;DISPLAY CURRENT COUNT
        STX   $40       ;SAVE INDEX
        LDY   #CT1      ;WASTE SOME TIME
DLY1    LDX   #CT2
DLY2    DEX
        BNE   DLY2
        DEY
        BNE   DLY1
        LDX   $40       ;RESTORE INDEX
        INX             ;ADD 1 TO COUNT
        CPX   #$10      ;HAS COUNT BEEN COMPLETED?
        BNE   DSPLY     ;NO, CONTINUE
        BRK
```

Program 5-4 is the hexadecimal version. Remember that you must also place Table 5-6 in memory addresses 0380 through 038F. CPX #$10 subtracts 10 hex from index register X; X is not changed but the flags are affected. Note that CPX's operand must be 10 (not 0F), since the program increments index register X before the comparison. We must save the count temporarily in memory location 0040 because the delay routine uses index register X.

The 6502 microprocessor has special instructions for manipulating the index registers. These instructions are:

COMPARE MEMORY AND INDEX REGISTER (CPX or CPY)
DECREMENT INDEX REGISTER BY 1 (DEX or DEY)
INCREMENT INDEX REGISTER BY 1 (INX or INY)
LOAD INDEX REGISTER (LDX or LDY)
STORE INDEX REGISTER (STX or STY)
TRANSFER ACCUMULATOR TO INDEX REGISTER (TAX or TAY)
TRANSFER INDEX REGISTER TO ACCUMULATOR (TXA or TYA)

PROGRAM 5-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#$FF
0201	FF			
0202	8D		STA	$1741
0203	41			
0204	17			
0205	A2		LDX	#0
0206	00			
0207	A9		LDA	#$08
0208	08			
0209	8D		STA	$1742
020A	42			
020B	17			
020C	BD	DSPLY	LDA	$0380,X
020D	80			
020E	03			
020F	8D		STA	$1740
0210	40			
0211	17			
0212	86		STX	$40
0213	40			
0214	A0		LDY	#CT1
0215	CT1			
0216	A2	DLY1	LDX	#CT2
0217	CT2			
0218	CA	DLY2	DEX	
0219	D0		BNE	DLY2
021A	FD			
021B	88		DEY	
021C	D0		BNE	DLY1
021D	F8			
021E	A6		LDX	$40
021F	40			
0220	E8		INX	
0221	E0		CPX	#$10
0222	10			
0223	D0		BNE	DSPLY
0224	E7			
0225	00		BRK	

There are some differences between the index registers which we will discuss as we encounter them. You should note the variations in which indexed addressing modes different instructions allow (see Table A1-1 or your programming card).

PROBLEM 5-9

Make Program 5-4 use the rightmost display (display 6). How could you revise the program so that it fetches the display number (assumed to be between 1 and 6 inclusive) from memory location 0041?

Example:

(0041) = 02 causes the program to count on display #2, the next to most significant digit of the address display.

PROBLEM 5-10

Make Program 5-4 count continuously, starting over at zero after it reaches F.

PROBLEM 5-11

Make Program 5-4 start at F and count down to zero. Why is it easier to program the 6502 processor to count down rather than up?

Hint: Subtract 1 from the base address in the indexed LDA instruction and start the count at 10 (hex). Then you can use the ZERO flag as an exit condition.

PROBLEM 5-12

Revise the continuous counting program (Problem 5-10) so that it continues only as long as the switch attached to bit 0 of port A of the user 6530 device is open. Write one version that checks the switch after each digit is displayed and one version that checks the switch only when it reaches the end of the table (i.e., after displaying F).

PROBLEM 5-13

Implement the counting program with the following nonstandard hexadecimal digit set used by Hewlett-Packard in their 5001 Signature Analyzer, a piece of test equipment that can detect faults in microprocessor-based systems. Hewlett-Packard uses this set rather than the normal one because its digits are easy to tell apart and can be read upside down; both of these characteristics are necessary in service instruments. Besides, this set can be formed on seven-segment displays without using any lowercase letters.

NORMAL HEXADECIMAL DIGIT	HP5001 HEXADECIMAL DIGIT
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	A
B or b	C
C	F
D or d	H
E	P
F	U

SWITCH AND LIGHT PROGRAM

We can also use the table to display the number of the last switch closed.

PROBLEM 5-14

Combine the switch identification program of Laboratory 4 (Program 4-5) with the seven-segment conversion program (Program 5-3) to wait for a switch closure at port A of the user 6530 device and report its number on the leftmost seven-segment display (display 1).

ADVANTAGES AND DISADVANTAGES OF LOOKUP TABLES

By now, you have seen many of the advantages and disadvantages of look-up tables. Among the advantages are:

- No computation is necessary so tables are faster than calculations unless the calculations are very simple.
- No program is required beyond the basic lookup routine. Lookup tables are thus easy to implement.
- The same lookup routine can be used for many different tables. Changes and extensions are simple and additional tables involve almost no programming at all.
- Table entries are available for other purposes (such as counting) in a convenient order.

- The table-lookup procedure is the same for all values. There are no boundary problems or variations in execution time.

Among the disadvantages of tables are:

- They require extra memory, particularly if the range of input values is large or great accuracy is necessary.
- They may be difficult to organize unless the input data is simple.
- The table-lookup procedure cannot distinguish common or simple cases that might be handled easily.
- Programs that use tables may be very difficult to understand, since no calculations are performed explicitly.

HARDWARE/SOFTWARE TRADEOFFS

As with inputs, hardware can do part of the output processing. For example, a 7447 decoder (see Table 5-7 and Figure 5-6) will automatically convert decimal inputs into common-anode seven-segment code. Neither a table nor a conversion routine is necessary.

In fact, the 7447 device is more than just a decoder. It also has

1) A LAMP TEST input that lights all the segments to show if they are working.
2) A blanking input (BI) that turns all the segments off.
3) A ripple blanking input and output (RBI and RBO) that can be used to blank leading or trailing 0's (e.g., display 37 instead of 0037 or 37.00). If the ripple blanking input is low, a zero data input will not be displayed and the ripple blanking output will be low. If the display is not blanked, the ripple blanking output will be high. This output is then attached to the ripple blanking input of the next digit.

A decoder such as the 7447 can replace a large amount of software. Like an encoder, a decoder also increases the number of parts and connections, decreases reliability, uses board space, and adds to the per-unit cost.

PROBLEM 5-15

Change the program for Problem 5-14 so that it displays a blank instead of a zero. Do not change the conversion table. How would you blank a zero if you were using a 7447 decoder?

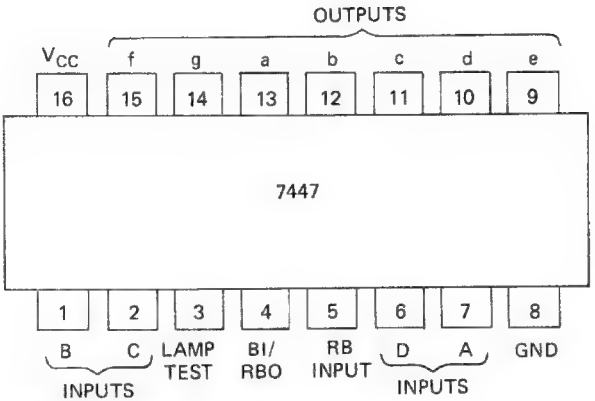


FIGURE 5-6. Pin assignments for the 7447 seven-segment decoder/driver.

Table 5-7

FUNCTION TABLE FOR THE 7447 SEVEN-SEGMENT DECODER/DRIVER

DECIMAL OR FUNCTION	INPUTS							OUTPUTS						
	LT	RBI	D	C	B	A	BI/RBO	a	b	c	d	e	f	g
0	1	1	0	0	0	0	1	0	0	0	0	0	0	1
1	1	X	0	0	0	1	1	1	0	0	1	1	1	1
2	1	X	0	0	1	0	1	0	0	1	0	0	1	0
3	1	X	0	0	1	1	1	0	0	0	0	1	1	0
4	1	X	0	1	0	0	1	1	0	0	1	1	0	0
5	1	X	0	1	0	1	1	0	1	0	0	1	0	0
6	1	X	0	1	1	0	1	1	1	0	0	0	0	0
7	1	X	0	1	1	1	1	0	0	0	1	1	1	1
8	1	X	1	0	0	0	1	0	0	0	0	0	0	0
9	1	X	1	0	0	1	1	0	0	0	1	1	0	0
10	1	X	1	0	1	0	1	1	1	1	0	0	1	0
11	1	X	1	0	1	1	1	1	1	0	0	1	1	0
12	1	X	1	1	0	0	1	1	0	1	1	1	0	0
13	1	X	1	1	0	1	1	0	1	1	0	1	0	0
14	1	X	1	1	1	0	1	1	1	1	0	0	0	0
15	1	X	1	1	1	1	1	1	1	1	1	1	1	1
BI	X	X	X	X	X	X	0	1	1	1	1	1	1	1
RBI	1	0	0	0	0	0	0	1	1	1	1	1	1	1
LT	0	X	X	X	X	X	1	0	0	0	0	0	0	0

KEY POINT SUMMARY

1) Most output devices (and observers) require that data be available for a relatively long time by processor standards. The I/O ports must latch the data and the processor must not change it too frequently.

2) Outputs must usually be converted into the forms required by peripherals. Either hardware (decoders) or software can perform the conversions.

3) Output transfers generally involve control signals as well as data. These control signals may be used for multiplexing or for controlling peripheral operations.

4) Lookup tables are a convenient way to perform code conversions if the functions are complex. Such tables simply contain all the codes organized in a convenient manner. They are easy and quick to use but may occupy a large amount of memory.

5) In the indexed addressing mode, the processor calculates the actual (effective) address to be used in executing the instruction. The calculation involves adding the contents of an index register to the address included in the instruction. Indexed addressing lets the programmer implement lookup tables and access successive elements in an array or table.

6) The 6502 microprocessor has special instructions for manipulating its index registers. These instructions are CPX and CPY (COMPARE MEMORY AND INDEX REGISTER), DEX and DEY (DECREMENT INDEX REGISTER BY 1), INX and INY (INCREMENT INDEX REGISTER BY 1), LDX and LDY (LOAD INDEX REGISTER), STX and STY (STORE INDEX REGISTER), TAX and TAY (TRANSFER ACCUMULATOR TO INDEX REGISTER), and TXA and TYA (TRANSFER INDEX REGISTER TO ACCUMULATOR). Instructions that change the index registers also change the effective addresses of instructions that use indexed addressing. Indexed instructions can therefore refer to different effective addresses at different times.

7) A microprocessor can usually update operator displays while performing other tasks, since the displays change slowly.

Laboratory 6

Processing Data Arrays

PURPOSE

To learn how to process arrays or blocks of data.

PARTS REQUIRED

None.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, pp. 81-96, 105-106, 127-129, 156-161.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 5.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 37-39, 42-48, 58-61, 80-83, 147-149.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 168-170 (index register), 338-340 (compare instructions), 367-368 (indirect addressing).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapters 5 and 10.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 6 and 7, Appendix G.

WHAT YOU SHOULD LEARN

1) What identifies elements of an array or block of data.
2) Why flexible addressing methods are important.
3) The most efficient approach to processing arrays with the 6502 microprocessor.
4) How to perform a summation.
5) How to use a terminator.
6) How to determine whether numbers are within specified limits.
7) How to display a block of data.
8) How to display a message.
9) How to make programs more general and more flexible.
10) How to use the indirect indexed (postindexed) addressing mode to process arrays with variable base addresses.
11) What a pointer is and how it is used.

TERMS

Array—a collection of related data items, usually stored in consecutive memory locations (also called a *block*).

Block—*see* Array.

Borrow (or **true borrow**)—a bit which is set to 1 if a subtraction produces a negative result and to 0 if a subtraction produces a positive or zero result. The borrow is used commonly to subtract numbers that are too long to be handled in a single operation.

Checksum—a logical sum of data used to guard against errors.

Indirect addressing—an addressing mode in which the effective address is the contents of the address included in the instruction, rather than the address itself.

Indirect indexed addressing—an addressing mode in which the effective address is determined by first obtaining the base address indirectly and then indexing from that base address. Also known as *postindexing,* since the indexing is performed after the indirection.

Inverted borrow—a bit which is set to 0 if a subtraction produces a negative result and to 1 if a subtraction produces a positive or

zero result. An inverted borrow can be used like a (true) borrow, except that the complement of its value (i.e., 1 minus its value) must be used in the extension to longer numbers.

Limit checking—determining if a data item is within specified limits; that is, if it is below an upper threshold and above a lower threshold. This procedure eliminates the problem of dealing with clearly invalid data that may be the result of operator error or communications errors. Typical examples of clearly invalid data are a transaction dated February 30th and a room temperature setting of 70°C (presumably, the operator meant 70°F).

Logical sum—a binary sum with no carries between bit positions.

Object code (or **object program**)—the program that is the output of a translator program, such as an assembler. Usually, a machine language program ready for execution.

Offset—distance from a starting point or base address.

Pointer—a register or memory location that contains an address rather than data.

Postindexing—*see* Indirect indexed addressing.

Source code (or **source program**)—computer program written in an assembly language or high-level language.

Terminator—a data item that has no function other than to mark the end of an array.

6502 INSTRUCTIONS

CLC—clear carry; set the CARRY flag to zero.

NOP—no operation; do nothing except increment the program counter.

TAX—transfer accumulator to index register X; transfer the contents of the accumulator to index register X. The accumulator does not change.

TXA—transfer index register X to accumulator; transfer the contents of index register X to the accumulator. Index register X does not change.

DATA ARRAYS

Most computing tasks involve applying the same instructions to many different data items. A collection of related data is usually called an *array* or *block*. Typical operations on arrays are calculating averages, finding the largest element for scaling, organizing data for storage on tape or disk,

editing strings of characters, sorting, arranging sequences of operations, performing statistical analysis, and searching for particular commands or other inputs.

The elements of arrays are most often stored in successive memory addresses. Two items are then necessary to reach a particular element of the array:

1) The starting address of the entire array or *base address.*
2) The element number or *index.*

We often refer mathematically to an element of an array as A_i, where A identifies the array as a whole (i.e., base address), and i identifies the particular element (i.e., index). Note that once you have determined the starting address of an array, you may refer to all elements relative to it (i.e., you may refer to "the seventh element" or "the fifteenth element"). Programs that handle arrays in this way need only be told where the arrays start; the data need not be moved to particular memory locations.

Flexible addressing methods are the keys to processing arrays of data. One sequence of instructions should be able to process any element. Otherwise, minor changes in the locations, lengths, or other characteristics of the arrays will require major revisions in the program. A flexible addressing method such as indexed or indirect addressing allows a single instruction to handle data at many different effective addresses.

PROBLEM 6-1

Which of these instructions could you use to handle any element of an array? Why?

a) LDA $40
b) LDX #$A3
c) LDA $0340,X
d) LDX $40

Which instructions can be used to transfer data from several different memory locations even if the program is stored in read-only memory?

PROBLEM 6-2

If an array starts at base address B and each element occupies one memory location, what is the address of the second element? Assume that memory location B contains the "zeroth" element. What is the address of the jth element, where j is an arbitrary integer? How are these addresses affected if we refer to the

element in B as the "first" element? This variation is similar to the alternative numbering of floors in a building as Ground (0), 1, 2, 3, etc., or as 1, 2, 3, 4, etc.

PROBLEM 6-3

How are the answers to Problem 6-2 affected if each element occupies two memory locations? What if each element occupies k memory locations, where k is an arbitrary integer?

PROBLEM 6-4

If the arrays are two-dimensional, we can store them by row (or by column) in the linear memory of the computer. For example, we can refer to an element as A_{jk}, where j is the row number and k is the column number. We can store the elements in memory in the following order, starting with the zeroth row: A_{00}, A_{01}, A_{02}, . . . , A_{0n}, A_{10}, A_{11}, A_{12}, . . . , A_{m0}, A_{m1}, A_{m2}, . . . , A_{mn}, where m is the number of the last row and n is the number of the last column (the array has a total of $m + 1$ rows and $n + 1$ columns, since we have started each dimension at zero). If we store element A_{00} in the base address B, what is the address of A_{12}? What is the address of element A_{jk}, where j and k are arbitrary integers? How are these addresses affected if the elements each occupy more than one memory location?

PROBLEM 6-5

Assume that an array describes the angles at which a vehicle should move (0 to 359 degrees) and the number of minutes (0 to 59) for which it should travel at each angle. Thus each entry consists of 3 bytes; the first two contain the angle and the third the travel time at that angle. If the array starts at memory address BASE with the first angle, determine which address (or addresses) contain:

a) The third angle in the series.

b) The travel time at the fifth angle.

c) The sixth angle in the series.

d) The travel time at the eighth angle.

PROCESSING ARRAYS WITH THE 6502 MICROPROCESSOR

The most efficient approach to processing arrays with the 6502 microprocessor is as follows (see Figure 6-1 for a flowchart):

1) Place the length of the array in an index register using LDX or LDY. Working backward through the array is more efficient than working forward because you can decrement the index register to zero and use

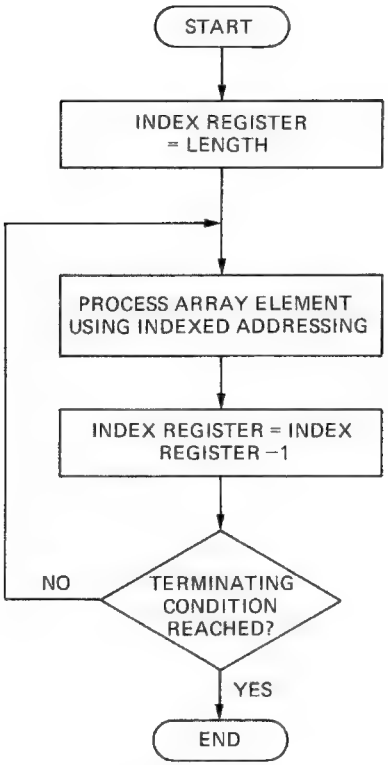


FIGURE 6-1. Array processing with the 6502 microprocessor.

the setting of the ZERO flag as the terminating condition. No comparison instruction is necessary.

2) Refer to an element of the array by indexing from a base address one less than the lowest address the array occupies. A typical instruction would be ADC START-1,X where START is the lowest occupied address. The −1 is necessary because we terminate the loop as soon as the index register is decremented to zero. Thus 1 is the smallest index that is ever used.

3) Access other elements in the array either by using different bases or by changing the index register. For example, ADC START+5,X will add to the accumulator the contents of an element located 6 bytes ahead of where the processor is working.

4) Move backward to the preceding element of the array by executing the instruction DEX or DEY (or forward by executing INX or INY).

Note some of the features of this approach:

1) The instructions assume that the array starts at a fixed base address. We will show later how to eliminate this restriction by using the postindexed (indirect indexed) addressing mode.
2) You can perform operations directly on any element in the array simply by using the appropriate base; for example,

LDA START-1,X—load the element at address START−1 + (X) into the accumulator.

EOR START+9,X—logically EXCLUSIVE OR the element at address START + 9 + (X) with the contents of the accumulator.

PROBLEM 6-6

Write a program that logically ANDs the contents of memory location START + 8 + (X) with the contents of location START − 1 + (X) and stores the result in START + 8 + (X). START is a fixed base address.

Example:

START = 0340

(X) = 06

Result:

(034E) = (034E) AND (0345)

Remember that the parentheses indicate "contents of."

PROBLEM 6-7

Write a program that moves the contents of memory location START − 2 + (X) to location START + (X).

Example:

START = 0340

(X) = 06

Result:

(0346) = (0344)

PROBLEM 6-8

Write a program that adds 3 to index register X. Write one version that uses INX and one version that uses ADC. Which version is shorter? Which is faster? Which approach is better if you have to add 9 to register X? How could you add the contents of memory location 0040 to register X?

Example:

(X) = 0C

(0040) = 27

Results:

a) After adding 3 to index register X

(X) = 0F

b) After adding 9 to index register X

(X) = 15

c) After adding (0040) to index register X

(X) = 0C + 27 = 33

SUM OF DATA

A simple example of array processing is finding the sum of the elements. This task is part of calculating an average, a summation, or a numerical integral. The following program assumes that the array consists of four elements in memory locations 0340 through 0343 (see Figure 6-2 for a flowchart):

```
        LDX   #4          ;INDEX = LENGTH
        LDA   #0          ;CLEAR THE SUM INITIALLY
ADDELM  CLC               ;SET CARRY TO ZERO ALWAYS
        ADC   $033F,X     ;ADD AN ELEMENT TO THE SUM
        DEX
        BNE   ADDELM      ;CONTINUE UNTIL ALL ELEMENTS ADDED
        STA   $40         ;SAVE SUM
        BRK
```

Program 6-1 is the hexadecimal version.

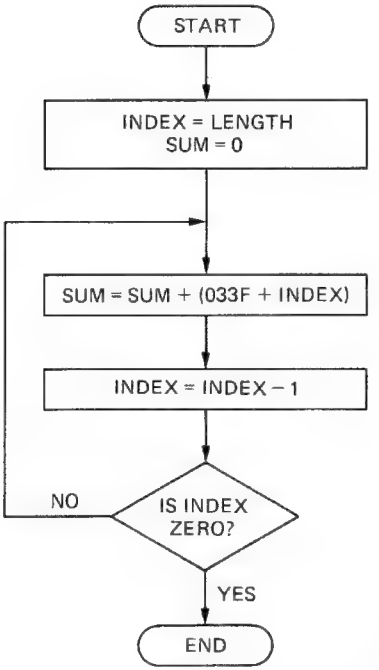


FIGURE 6-2 Flowchart of summation program.

PROGRAM 6-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#4
0201	04			
0202	A9		LDA	#0
0203	00			
0204	18	ADDELM	CLC	
0205	7D		ADC	$033F,X
0206	3F			
0207	03			
0208	CA		DEX	
0209	D0		BNE	ADDELM
020A	F9			
020B	85		STA	$40
020C	40			
020D	00		BRK	

Run Program 6-1 with the following data:

(0340) = 07

(0341) = 23

(0342) = 31

(0343) = 20

Result:

(0040) = 7B

Remember that all the numbers are hexadecimal. Replace (0342) with F1. What is the result, and why?

Note that we must clear the CARRY flag during each iteration, so it does not affect the addition. Most microprocessors have an addition instruction that does not include the CARRY, but the 6502 has only ADC. What happens if you run Program 6-1 without the CLC instruction? Try the two sets of data that we used previously. To eliminate CLC from the program, replace it with a NOP (NO OPERATION, EA hex).

Implement the following variations of Program 6-1.

PROBLEM 6-9

Add six numbers starting with memory location 0340.

Sample Problem:

(0340) = 07

(0341) = 23

(0342) = 31

(0343) = 20

(0344) = 16

(0345) = 38

Result:

(0040) = C9

PROBLEM 6-10

Get the number of elements in the array from memory location 0041.

Sample Problem:

(0041) = 05 (i.e., the array has five elements)

(0340) = 07

(0341) = 23

(0342) = 31

(0343) = 20

(0344) = 16

Result:

(0040) = 91

PROBLEM 6-11

Change Program 6-1 so that it EXCLUSIVE ORs the numbers together instead of adding them. The result is called a *logical sum* or *checksum* and is often used to detect errors in tape or disk records.

Sample Problem (four data items starting with memory location 0340, result in 0040):

(0340) = 07

(0341) = 23

(0342) = 31

(0343) = 20

Result:

(0040) = 35

PROBLEM 6-12

Extend Program 6-1 so that it saves the carries and stores the 16-bit sum in memory locations 0040 and 0041 (more significant byte in 0041).

Sample Problem:

(0340) = F7

(0341) = 23

(0342) = 31

(0343) = 20

(0344) = 16

Result:

(0040) = 81 (less significant byte of sum)

(0041) = 01 (more significant byte of sum)

USING A TERMINATOR

If you are not sure how long the array is (or do not want to count the elements each time), you can end the array with a special marker or terminator. Note that the terminator must have a value that cannot be confused with a real element. In the case of a sum, zero is a good choice because it does not affect the sum anyway. The program using the terminator is (see Figure 6-3 for a flowchart)

```
        LDX   #0          ;INDEX = ZERO
        TXA               ;SUM = ZERO
ADDELM  LDY   $0340,X     ;IS ELEMENT ZERO?
        BEQ   DONE        ;YES, DONE
        CLC               ;NO, SUM = SUM + ELEMENT
        ADC   $0340,X
        INX
        JMP   ADDELM
DONE    STA   $40         ;SAVE SUM
        BRK
```

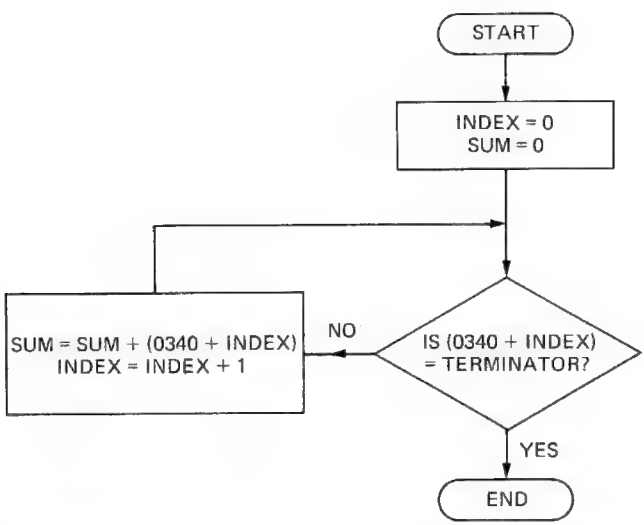


FIGURE 6-3. Flowchart of summation program with terminator.

Program 6-2 is the hexadecimal version. Here we must work forward through the array, since we have assumed that the terminator is at the end. Of course, we could put the terminator at the beginning and work backward.

PROGRAM 6-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#0
0201	00			
0202	8A		TXA	
0203	BC	ADDELM	LDY	$0340,X
0204	40			
0205	03			
0206	F0		BEQ	DONE
0207	08			
0208	18		CLC	
0209	7D		ADC	$0340,X
020A	40			
020B	03			
020C	E8		INX	
020D	4C		JMP	ADDELM
020E	03			
020F	02			
0210	85	DONE	STA	$40
0211	40			
0212	00		BRK	

Run Program 6-2 with the following data:

(0340) = 07

(0341) = 23

(0342) = 31

(0343) = 20

(0344) = 16

(0345) = 38

(0346) = 00

Result:

(0040) = C9

What happens if you set (0343) = 00? What are the advantages and disadvantages of using a terminator as compared to counting the number of elements? Which approach results in faster executing programs? Which approach makes data entry simpler?

How could you revise Program 6-2 to make the JMP instruction unnecessary? **Hint**: Adding zero to the sum does not change its value, so you can check for the terminator after adding in the current element.

PROBLEM 6-13

Revise Program 6-2 so that it does not use index register Y. **Hint**: Use the sequence INC, DEC to set the ZERO flag if the memory location contains zero.

PROBLEM 6-14

If some elements in the array could be zero, you must use a nonzero terminator. Revise Program 6-2 to use FF as a terminator. Would this approach be better than the one in Program 6-2 if the data values were the numbers of characters received from a teletypewriter [10 characters per second (cps)] in 1 s? Which approach would be better if the values were the time delays between characters? Assume that the processor must wait one time unit before it checks for the next character and will end the search if the next character does not appear before 256 time units have elapsed.

LIMIT CHECKING

Often we must determine if elements in an array are valid: that is, if they are within certain limits, below a threshold, or have legal values. Determining if elements are between lower and upper limits is referred to as *limit checking*. The key instruction here is a comparison (CMP, CPX, or CPY) which subtracts the contents of a memory location from the contents of an index register or the accumulator. Compare instructions affect the flags, but do not change the index register or accumulator.

If the operands are unsigned, we can use the CARRY flag to determine which one is larger. Compare instructions set the CARRY flag as follows:

$$\text{CARRY} = 1 \text{ if } (\text{REG}) \geq (\text{M})$$

$$\text{CARRY} = 0 \text{ if } (\text{REG}) < (\text{M})$$

where REG refers to the accumulator or index register and M to the memory location. We refer to the CARRY flag as an *inverted borrow*, since it is set to 0 if the subtraction requires a borrow and to 1 if it does not. This effect on the CARRY flag is a peculiarity of the 6502 microprocessor; most processors produce a true borrow by performing an extra inversion on-chip.

The following program sums six elements but rejects those that are 80 hex or above (i.e., negative).

```
        LDX   #6          ;INDEX = LENGTH
        LDA   #0          ;CLEAR THE SUM INITIALLY
ADDELM  LDY   $033F,X     ;CHECK ELEMENT AGAINST THRESHOLD
        CPY   #$80
        BCS   COUNT       ;REJECT IF VALUE ABOVE THRESHOLD
        ADC   $033F,X     ;ADD AN ELEMENT (CARRY IS ZERO)
COUNT   DEX
        BNE   ADDELM
        STA   $40
        BRK
```

We have taken advantage of the fact that we know CARRY = 0 if we reach ADC $033F,X, since otherwise BCS would have caused a branch. If we want to make the threshold itself valid (i.e., reject elements that are above 80 hex instead of 80 hex or above), we could simply replace CPY #$80 with CPY #$81. Program 6-3 is the hexadecimal version; enter and run it with the following data:

(0340) = 07

(0341) = 20

(0342) = F1

(0343) = 3C

(0344) = 80

(0345) = 73

Result:

(0040) = D6

PROBLEM 6-15

Revise Program 6-3 so that it rejects elements that are 80 hex or above or 20 hex or below.

Sample Problem:

(0340) = 07

(0341) = 20

(0342) = F1

(0343) = 3C

(0344) = 80

(0345) = 73

Result:

(0040) = AF

PROGRAM 6-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#6
0201	06			
0202	A9		LDA	#0
0203	00			
0204	BC	ADDELM	LDY	$033F,X
0205	3F			
0206	03			
0207	C0		CPY	#$80
0208	80			
0209	B0		BCS	COUNT
020A	03			
020B	7D		ADC	$033F,X
020C	3F			
020D	03			
020E	CA	COUNT	DEX	
020F	D0		BNE	ADDELM
0210	F3			
0211	85		STA	$40
0212	40			
0213	00		BRK	

Limit checking is often a good way to reduce the effects of one or two measurements that are distinctly different from the others (and therefore suspect). Many data analysis routines throw out the highest and lowest values before averaging or performing other functions to eliminate the impact of readings that may be the result of noise, improper settings, external handling of the system, or other unreproducible conditions. Limit checking also eliminates silly errors, such as an operator specifying a nonexistent time (e.g., 0880 hours instead of the intended 0800 hours)

or a parameter value that is unphysical or unrealistic (e.g., an automobile speed of 1000 km/hr instead of the intended 100 km/hr).

DISPLAYING AN ARRAY

We can use our array handling techniques to place different data on each of the on-board seven-segment displays. The following program will do the job (see Figure 6-4 for a flowchart).

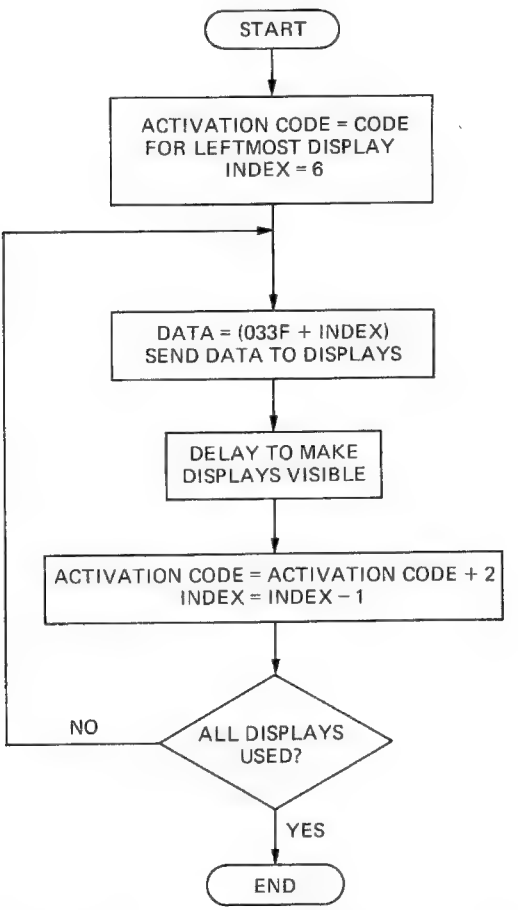


FIGURE 6-4. Flowchart for displaying an array.

```
        LDA   #$FF          ;MAKE PORT A OUTPUT
        STA   $1741
START   LDA   #%00001000    ;START WITH LEFTMOST DISPLAY
        STA   $1742
        LDX   #6            ;POINT TO START OF ARRAY
DSPLY   LDA   $033F,X       ;SEND DATA TO DISPLAY
        STA   $1740
        TXA
        LDY   #CT1          ;DELAY A WHILE
DLY1    LDX   #CT2
DLY2    DEX
        BNE   DLY2
        DEY
        BNE   DLY1
        INC   $1742         ;ACTIVATE NEXT DISPLAY
        INC   $1742
        TAX
        DEX                 ;COUNT DISPLAYS
        BNE   DSPLY
        BEQ   START         ;START OVER IF ALL USED
```

The hexadecimal version is Program 6-4.

PROGRAM 6-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A9		LDA	#$FF
0201	FF			
0202	8D		STA	$1741
0203	41			
0204	17			
0205	A9	START	LDA	#%00001000
0206	08			
0207	8D		STA	$1742
0208	42			
0209	17			
020A	A2		LDX	#6
020B	06			
020C	BD	DSPLY	LDA	$033F,X
020D	3F			
020E	03			
020F	8D		STA	$1740
0210	40			
0211	17			
0212	8A		TXA	

PROGRAM 6-4 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0213	A0		LDY	#CT1
0214	CT1			
0215	A2	DLY1	LDX	#CT2
0216	CT2			
0217	CA	DLY2	DEX	
0218	D0		BNE	DLY2
0219	FD			
021A	88		DEY	
021B	D0		BNE	DLY1
021C	F8			
021D	EE		INC	$1742
021E	42			
021F	17			
0220	EE		INC	$1742
0221	42			
0222	17			
0223	AA		TAX	
0224	CA		DEX	
0225	D0		BNE	DSPLY
0226	E5			
0227	F0		BEQ	START
0228	DC			

Set CT1 (memory location 0214) = CT2 (memory location 0216) = 00 and run Program 6-4 with the following data:

(0340) = 00

(0341) = 3F

(0342) = 38

(0343) = 38

(0344) = 79

(0345) = 76

You can use Tables 5-4 and 5-5 to produce your own messages on the displays. Remember, however, to load the data into memory backward; that is, start with the character intended for the leftmost display in memory location 0345. Note that we activate the next display (to the right) by incrementing memory location 1742 twice. If you are not sure

why this works, return to Laboratory 5 and examine Figures 5-4 and 5-5 and Tables 5-1 and 5-2.

PROBLEM 6-16

Leaving CT2 = 00, run Program 6-4 repeatedly with the following series of hexadecimal values for CT1 (memory location 0214): 80, 40, 20, 10, 08, 04, 02, 01. Explain what happens. How could you change Program 6-4 to produce a "newspanel" or "Times Square" display in which the message appears to move from right to left? This illusion of motion is necessary in video games and graphics systems.

PROBLEM 6-17

Change Program 6-4 so that it uses the following data:

(0350) = 54

(0351) = 5C

(0352) = 5B

(0353) = 3F

(0354) = 6D

(0355) = 7D

Could you make this change if the program were in ROM?

VARYING THE BASE ADDRESS

All the programs we have shown so far in this Laboratory have assumed a fixed base address. Such programs, of course, lack generality, since they always work on data at a fixed place in memory. We would like to make the base address into a variable, so that we can tell the program where the data is located. This would eliminate the need to move large amounts of data from one place to another and to change the entire program to use it on a computer with a different arrangement of memory addresses.

The 6502 microprocessor has an addressing mode that lets us specify the base address of an array as the contents of two successive memory locations on page zero. This is the indirect indexed mode, sometimes referred to as *postindexing* since indexing is performed after the base address is obtained indirectly. This mode requires the use of page zero for the indirect address and register Y for the index. A typical example is the instruction

```
LDA     ($40),Y
```

which loads the accumulator from the effective address obtained by adding index register Y to the base address in memory locations 0040 and 0041. If, for example, (0040) = 80, (0041) = 03, and (Y) = 3C, the base address is 0380 and the effective address is 0380 + 3C = 03BC. Thus memory locations 0040 and 0041 serve as a *pointer*, since they contain an address rather than data.

We can easily change Program 6-1 to use the indirect indexed mode. All that we must do is use register Y instead of register X and the indirect indexed addressing mode with ADC instead of the absolute indexed mode. The revised program is

```
        LDY   #4          ;INDEX = LENGTH
        LDA   #0          ;CLEAR THE SUM INITIALLY
ADDELM  CLC               ;SET CARRY TO ZERO ALWAYS
        ADC   ($40),Y     ;ADD AN ELEMENT TO THE SUM
        DEY
        BNE   ADDELM
        STA   $42
        BRK
```

Program 6-5 is the hexadecimal version using indirect indexed addressing. To have this program work on an array located anywhere in memory, all that we must do is place the base address in memory locations 0040 and 0041. Enter and run Program 6-5 for the same sample cases we used with Program 6-1. Remember to load the base address

PROGRAM 6-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A0		LDY	#4
0201	04			
0202	A9		LDA	#0
0203	00			
0204	18	ADDELM	CLC	
0205	71		ADC	($40),Y
0206	40			
0207	88		DEY	
0208	D0		BNE	ADDELM
0209	FA			
020A	85		STA	$42
020B	42			
020C	00		BRK	

(033F) upside down into memory locations 0040 and 0041 before executing the program.

PROBLEM 6-18

Change Program 6-2 so that it uses indirect indexed addressing and stores the sum in memory location 0042. How large a time penalty do we pay for using the indirect indexed mode? Compare the execution time of this mode with the zero page indexed mode and with the absolute indexed mode.

PROBLEM 6-19

Write a program starting in memory location 0280 that loads 0340 upside-down into memory locations 0040 and 0041. How long does this program take to execute? Explain why this task must be performed before the computer executes an instruction that uses indirect indexed addressing.

PROBLEM 6-20

Write a program that logically ANDs the contents of memory location START + 8 + (Y) with the contents of location START − 1 + (Y) and stores the result in START + 8 + (Y). Assume that the address START is stored in memory locations 0040 and 0041. Be careful; your program must work correctly even if the two operands are stored on different pages. For the sample cases, assume that (0040) = 00 and (0041) = 02 (i.e., the base address is 0200).

Sample Problems:

1) (Y) = 80
(027F) = 23
(0288) = 34
Result: (0288) = (0288) AND (027F) = 20

2) (Y) = FE
(02FD) = C7
(0306) = 6D
Result: (0306) = (0306) AND (02FD) = 45

Remember, you can initialize index register Y by loading its starting value into memory location 00F4 before executing the program.

KEY POINT SUMMARY

1) Arrays are collections of data items that have similar meanings or purposes. An element of an array is characterized by its position or index; the entire array is characterized by its starting address. Thus, to

reach a particular element of an array, you must know the starting address of the array and the index of the element.

2) The keys to processing arrays are:

- An index that determines which element is being processed.
- A flexible addressing method that allows a single set of instructions to handle any or all of the elements.
- A counter or terminator that can be used to determine the length of the array.

3) To process arrays with the 6502 microprocessor, you can use either index register to hold the index, indexed addressing to reach the data in memory, and the other index register or a memory location to hold the counter or terminator.

4) A convenient way to process an array is to use the starting address minus 1 as the base address in the indexed instructions and work backward through the elements. You can then use the setting of the ZERO flag as an exit condition, since the program is counting the index register down to zero.

5) The comparison instructions can be used to determine if an element is within a specified range. If the operands are unsigned, the CARRY flag indicates which is larger. In the 6502 microprocessor, the CARRY flag acts as an inverted borrow; it is set to 1 if no borrow is necessary and to 0 if a borrow is required.

6) Loops within loops (i.e., nested loops) and variable base addresses and counters can be used to handle multidimensional arrays and to provide greater flexibility. The indirect indexed addressing mode allows you to specify the base address as the contents of two memory locations on page zero. This mode assumes the use of page zero and index register Y. The locations on page zero thus serve as a pointer to the actual array.

Laboratory 7

Forming Data Arrays

PURPOSE

To learn how to form arrays of data.

PARTS REQUIRED

Eight switches or pushbuttons attached as shown in Figure 2-1.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, pp. 157-164.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 179-198.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 3-9 to 3-10, Chapter 5 (particularly pp. 5-20 to 5-22), p. 11-123.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 37-40, 92-107, 111-112.

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chpater 5 (particularly pp. 51-53) and Chapter 10 (particularly pp. 157-161).

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapter 3.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapters 6 and 7, Appendix G.

WHAT YOU SHOULD LEARN

1) How to use indexed addressing to form arrays.
2) How to conclude the formation of an array.
3) How to clear an area of memory.
4) How to initialize or fill an area of memory.
5) How to enter input data into an array.
6) How to access a specific element of an array.
7) How to keep counts or running totals in an array.
8) How to use indexed indirect addressing to handle arrays of addresses.
9) The differences between logical and physical devices and the advantages of distinguishing between them.
10) How to construct and use an I/O device table that establishes the correspondence between logical and physical devices.
11) How to handle arrays that occupy more than 256 bytes of memory.

TERMS

Arithmetic shift—a shift operation that preserves the value of the sign bit (most significant bit). In a right shift, this results in the sign bit being copied into the succeeding bit positions (called *sign extension*).

Clear—set to zero.

Indexed indirect addressing—an addressing mode in which the effective address is determined by first indexing from the base address and then using the indexed address indirectly. Also known as *preindexing*, since the indexing is performed before the indirection. Of course, the array starting at the given base address must consist of addresses that can be used indirectly.

I/O device table—a table that establishes the correspondence between the logical devices to which programs refer and the physical devices that are actually used in data transfers. An I/O device table must be placed in memory in order to run a program that uses

logical devices on a computer with a particular set of actual (physical) devices.

Logical device—the input or output device to which a program refers. The actual or physical device is determined by looking up the logical device in an I/O device table—that is, in a table consisting of actual I/O addresses corresponding to the logical device numbers.

Logical shift—a shift operation that fills the vacated bits with 0's as it moves the original data to the left or right.

Physical device—an actual input or output device, as opposed to a logical device.

Preindexing—*see* Indexed indirect addressing.

Rotate—a shift operation that treats the data as if it were arranged in a circle, that is, as if the most significant and least significant bits were connected directly or through a carry bit.

6502 INSTRUCTIONS

ROL—rotate left; shift each bit of the accumulator or a memory location left one position as if bits 0 and 7 were connected through the CARRY flag (see Figure E-2).

ROR—rotate right; shift each bit of the accumulator or a memory location right one position as if bits 0 and 7 were connected through the CARRY flag (see Figure E-2).

STANDARD PROCEDURE FOR FORMING ARRAYS

The arrays that we used in Laboratory 6 do not, of course, appear magically in the computer's memory. In real applications, the program must form the array before processing it. Typically, forming an array requires a base address that tells us where the array starts and an index that tells us which element to fill next. Remember that on processors like the 6502, addresses are 16 bits long and data is 8 bits long. The index may occupy an 8-bit register or one memory location if the array occupies fewer than 256 bytes. We will discuss the handling of longer arrays as the last topic in this Laboratory.

The standard procedure for forming an array is as follows (see Figure 7-1):

1) Initialization.

BASE = STARTING ADDRESS OF ARRAY

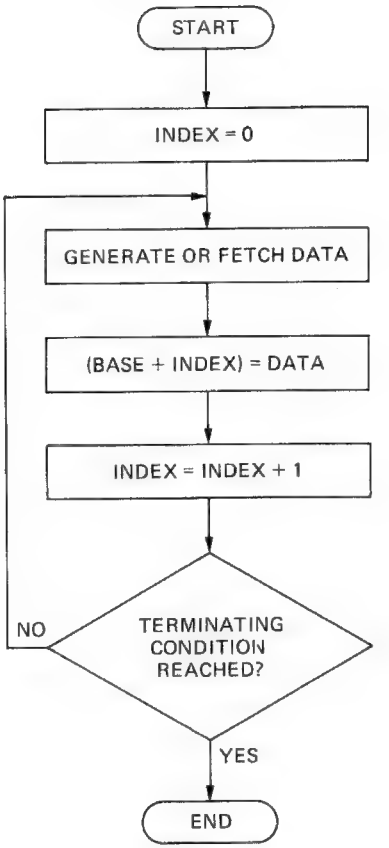


FIGURE 7-1. Flowchart for array formation.

INDEX = 0

LENGTH = LENGTH OF ARRAY (if known)

2) Entering an element.

(BASE + INDEX) = DATA

INDEX = INDEX + 1

Remember that the parentheses around BASE+INDEX mean "contents of." The data may be a constant, the result of a calculation, or an external input.

3) Conclusion.

   a) Maximum length.
   If INDEX = LENGTH then DONE; otherwise, return to step 2.

   b) Terminator.
   If DATA = TERMINATOR then DONE; otherwise, return to step 2.

Remember that on the 6502, we often find it more convenient to work backward through the array rather than forward, since we can then count the index down to zero and use the setting of the ZERO flag as an exit condition. A variety of methods can be used to conclude array formation.

CLEARING AN ARRAY

A simple way to form an array is to set all the elements to zero. This is a natural starting point for accumulating totals or test results. Note that you cannot assume that an unused RAM location contains zero; it could start in any state whatsoever when power is applied. The following program clears memory locations 0340 through 0347:

```
        LDA     #0          ;DATA = ZERO
        LDX     #8          ;NUMBER OF BYTES = 8
CLR1    STA     $033F,X     ;CLEAR A BYTE
        DEX
        BNE     CLR1        ;COUNT BYTES
        BRK
```

Program 7-1 is the hexadecimal version. Enter and run the program; try the variations in Problems 7-1 through 7-5.

PROBLEM 7-1

Clear memory locations 0042 through 0051.

PROBLEM 7-2

Clear memory locations 0350 through 035F.

PROGRAM 7-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#0
0201	00			
0202	A2		LDX	#8
0203	08			
0204	9D	CLR1	STA	$033F,X
0205	3F			
0206	03			
0207	CA		DEX	
0208	D0		BNE	CLR1
0209	FA			
020A	00		BRK	

PROBLEM 7-3

Place 80 hex in memory locations 0340 through 0347.

PROBLEM 7-4

Place the value from memory location 0040 in memory locations starting with 0340 and continuing through a number of locations given by the contents of memory location 0041. Does your program work properly if (0041) = 00?

Example:

(0040) = 3F (value to be used)

(0041) = 03 (number of locations to be filled)

Result:

(0340) = 3F

(0341) = 3F

(0342) = 3F

The program should not change any memory locations if (0041) = 00.

PROBLEM 7-5

Change your answer to Problem 7-4 so that it obtains the base address from memory locations 0042 and 0043. Assume that those locations contain an address one less than the lowest address occupied by the array.

Example:

(0040) = 3F (value to be used)

(0041) = 03 (number of locations to be filled)

(0042) = 7F (LSBs of base address)

(0043) = 03 (MSBs of base address)

Result:

(0380) = 3F

(0381) = 3F

(0382) = 3F

The program should not change any memory locations if (0041) = 00. **Hint**: Use the indirect indexed addressing mode, but remember that it works only with index register Y.

PLACING VALUES IN AN ARRAY

The next step is to place different values in the different elements of the array. The following program places the element number or index (1 through 8) in the corresponding position (see Figure 7-2 for a flowchart). Program 7-2 is the hexadecimal version. This program is complicated by the limited number of addressing modes available for the STX instruction. Note that STA, STX, and STY all have different sets of addressing modes; can you suggest reasons for this?

```
        LDX   #8          ;NUMBER OF BYTES = 8
LDIND   TXA               ;ELEMENT = INDEX
        STA   $033F,X
        DEX
        BNE   LDIND
        BRK
```

Enter and run Program 7-2. Note that this program is more than just an academic exercise, since it produces an array of identification numbers. For example, assume that you have a set of pressure readings taken at different points in a chemical process. You could sort that set into decreasing order while using the identification numbers to keep track of the points at which the readings were taken. The operator could then immedi-

PROGRAM 7-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#8
0201	08			
0202	8A	LDIND	TXA	
0203	9D		STA	$033F,X
0204	3F			
0205	03			
0206	CA		DEX	
0207	D0		BNE	LDIND
0208	F9			
0209	00		BRK	

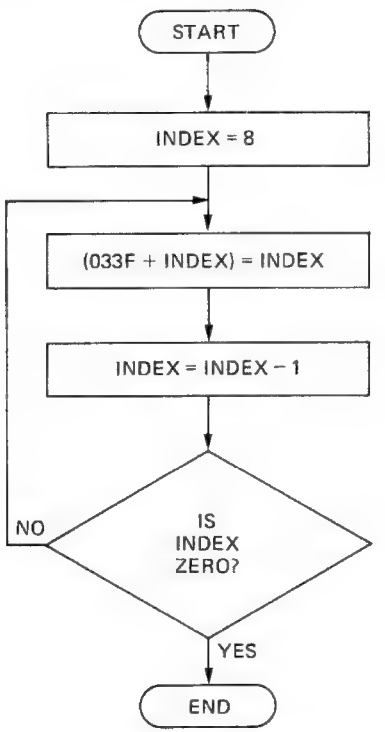


FIGURE 7-2. Flowchart for placing the element numbers in an array.

ately determine the highest value and where it occurred by examining the top line of the printed results. For instance, you could start with

POSITION	PRESSURE
1	40
2	27
3	66
4	59

and end with the pressures arranged in decreasing order

POSITION	PRESSURE
3	66
4	59
1	40
2	27

If we did not have the array of identification (position) numbers, we would know the highest pressure but we would not know where it occurred. Try the following variations of Program 7-2.

PROBLEM 7-6

Reverse the order of the elements; that is, start with (0347) = 01 and end with 0340) = 08.

PROBLEM 7-7

Start with 1 and let each subsequent element have twice the value of the previous element; that is,

(0340) = 01

(0341) = 02

(0342) = 04

(0343) = 08

(0344) = 10

(0345) = 20

(0346) = 40

(0347) = 80

The operation is a *logical shift,* since the vacated bit (bit 0) is always cleared.

PROBLEM 7-8

Create the following sequence:

(0340) = 80 (10000000 binary)

(0341) = C0 (11000000 binary)

(0342) = E0 (11100000 binary)

(0343) = F0 (11110000 binary)

(0344) = F8 (11111000 binary)

(0345) = FC (11111100 binary)

(0346) = FE (11111110 binary)

(0347) = FF (11111111 binary)

What are the values of these numbers if they are in the two's-complement form? The operation shown is a right *arithmetic shift*, since it does not change the sign (most significant) bit. Producing a right arithmetic shift with the 6502 microprocessor is awkward at best, since it requires an extra copy of bit 7. For example, the following sequence of instructions will shift the accumulator right 1 bit arithmetically:

```
TAY          ;SAVE ACCUMULATOR
ASL    A     ;MOVE BIT 7 TO CARRY
TYA          ;RESTORE ACCUMULATOR
ROR    A     ;SHIFT RIGHT WITH COPY OF BIT 7
```

ENTERING INPUT DATA INTO AN ARRAY

Our next task is to form an array from input data entered on the switches attached to port A of the user 6530 device. The steps are as follows (see Figure 7-3):

1) Initialize the index and the base address (if necessary).

   BASE = STARTING ADDRESS – 1

   INDEX = LENGTH OF ARRAY (4)

2) Wait for a switch to be closed.
3) Debounce the switch closure.
4) Identify the switch.
5) Enter the switch number into the array.

   (BASE + INDEX) = SWITCH NUMBER

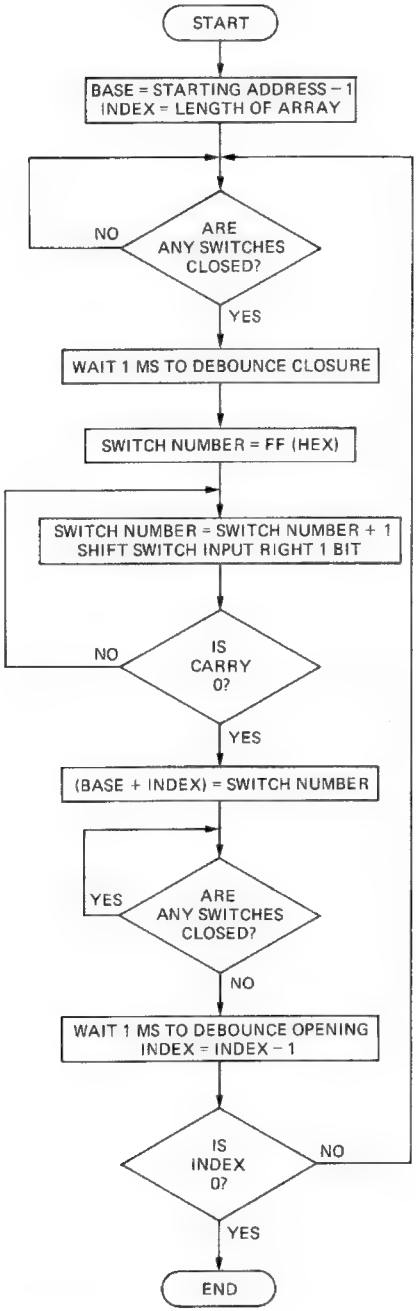


FIGURE 7-3. Flowchart for forming an array from the switches.

6) Update the index to prepare for the next entry.

$$INDEX = INDEX - 1$$

7) Wait for all switches to be open.
8) Debounce the switch opening.
9) If INDEX $\neq 0$, return to step 2.

The following program forms an array starting in memory location 0340 from 4 switch closures (all switches must be opened between closures).

```
;
;     SET UP USER 6530 PORT A FOR INPUT
;
                LDA     #0          ;MAKE PORT A INPUT
                STA     $1701
;
;     INITIALIZE INDEX TO LENGTH OF ARRAY
;
                LDX     #4          ;INDEX = ARRAY LENGTH
;
;     WAIT FOR SWITCH TO BE CLOSED
;
WAITC           LDA     $1700       ;READ DATA FROM SWITCHES
                CMP     #$FF        ;ARE ANY SWITCHES CLOSED?
                BEQ     WAITC       ;NO, WAIT
;
;     DEBOUNCE SWITCH CLOSURE WITH 1 MS DELAY
;
                LDY     #$C8        ;DELAY 1 MS AFTER CLOSURE
DLYC            DEY
                BNE     DLYC
;
;     IDENTIFY SWITCH BY SHIFTING INPUT
;
                LDY     #$FF        ;SWITCH NUMBER = -1
SRCHS           INY                 ;SWITCH NUMBER =
                                    ; SWITCH NUMBER + 1
                LSR     A           ;IS NEXT SWITCH CLOSED?
                BCS     SRCHS       ;NO, KEEP LOOKING
;
;     ENTER SWITCH NUMBER INTO ARRAY
;
                TYA
                STA     $033F,X     ;PUT SWITCH NUMBER IN ARRAY
;
;     WAIT FOR ALL SWITCHES TO OPEN
```

```
;
WAITO           LDA     $1700       ;READ DATA FROM SWITCHES
                CMP     #$FF        ;ARE ANY SWITCHES CLOSED?
                BNE     WAITO       ;YES, WAIT
;
;    DEBOUNCE SWITCH OPENING WITH 1 MS DELAY
;
                LDY     #$C8        ;DELAY 1 MS AFTER OPENING
DLYO            DEY
                BNE     DLYO
;
;    COUNT SWITCH CLOSURES
;
                DEX
                BNE     WAITC
                BRK
```

Program 7-3 is the hexadecimal version; enter and run the program. Use the following sequence of switch closures: 5, 7, 0, 3. Remember to open all switches after each closure. The result should be:

(0340) = 03

(0341) = 00

(0342) = 07

(0343) = 05

PROGRAM 7-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#0
0201	00			
0202	8D		STA	$1701
0203	01			
0204	17			
0205	A2		LDX	#4
0206	04			
0207	AD	WAITC	LDA	$1700
0208	00			
0209	17			
020A	C9		CMP	#$FF
020B	FF			
020C	F0		BEQ	WAITC
020D	F9			

PROGRAM 7-3 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
020E	A0		LDY	#$C8
020F	C8			
0210	88	DLYC	DEY	
0211	D0		BNE	DLYC
0212	FD			
0213	A0		LDY	#$FF
0214	FF			
0215	C8	SRCHS	INY	
0216	4A		LSR	A
0217	B0		BCS	SRCHS
0218	FC			
0219	98		TYA	
021A	9D		STA	$033F,X
021B	3F			
021C	03			
021D	AD	WAITO	LDA	$1700
021E	00			
021F	17			
0220	C9		CMP	#$FF
0221	FF			
0222	D0		BNE	WAITO
0223	F9			
0224	A0		LDY	#$C8
0225	C8			
0226	88	DLYO	DEY	
0227	D0		BNE	DLYO
0228	FD			
0229	CA		DEX	
022A	D0		BNE	WAITC
022B	DB			
022C	00		BRK	

Revise Program 7-3 to perform the following tasks:

PROBLEM 7-9

Enter eight switch closures into an array starting at memory location 0061.

PROBLEM 7-10

Use switch 0 as a terminator (i.e., the program should exit when you close switch 0). Can you ever get a data entry of zero?

Hint: If you have to insert a few instructions into the program, a simple procedure is to replace a sequence of instructions with a JMP to an unused area of memory. In the unused area, place the insert plus the instructions that you replaced. Then complete the patch with a JMP back to the instruction following the ones that you replaced. You can use NOPs to fill 1 or 2 extra bytes if necessary.

PROBLEM 7-11

Take the four entries in memory locations 0340 through 0343 and combine them to form two two-digit numbers in memory locations 0061 and 0062. Load memory location 0061 with the contents of memory locations 0340 (4 LSBs) and 0341 (4 MSBs); load memory location 0062 with the contents of memory locations 0342 (4 LSBs) and 0343 (4 MSBs).

Example:

Switches closed are 7, 3, 4, 2.

(0340) = 02

(0341) = 04

(0342) = 03

(0343) = 07

Result:

(0061) = 42

(0062) = 73

Note the obvious similarity between this process and the entry of a four-digit hexadecimal address from the KIM keyboard. Remember that the keys are simply binary switches.

ACCESSING SPECIFIC ELEMENTS

Still another problem is how to find a specific element of the array. This is essential when the program must count events (number of transactions of a particular type or number of activations of a particular sensor) or must accumulate data properly (e.g., total for a particular account, test point, or station). For example, the following program clears a particular element of an array starting at address 0340. Memory location 0041 contains the element number.

Examples:

1) (0041) = 02
   Result: [0340 + (0041)] = (0342) = 00
2) (0041) = 07
   Result: [0340 + (0041)] = (0347) = 00

```
LDX     $41         ;GET INDEX
LDA     #0          ;GET DATA
STA     $0340,X     ;CLEAR INDEXED ELEMENT
BRK
```

Program 7-4 is the hexadecimal version; enter it and run the two examples. Note the obvious similarity between Program 7-4 and the seven-segment code conversion routine (Program 5-3).

Note that the instruction STA $3040,X stores the accumulator in the indexed address (i.e., in a memory location); it has no effect on index register X. Be particularly careful of instructions like DEC $40,X; these instructions affect a memory location, not the index register. Note the difference between DEC 0,X and DEX.

PROGRAM 7-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A6	LDX	$41
0201	41		
0202	A9	LDA	#0
0203	00		
0204	9D	STA	$0340,X
0205	40		
0206	03		
0207	00	BRK	

Try the following variations of Program 7-4.

PROBLEM 7-12

Place 80 hex in the accessed memory location.

Example:

$$(0041) = 02$$

Result:

$$[0340 + (0041)] = (0342) = 80$$

PROBLEM 7-13

Add 1 to the contents of the accessed memory location.

Example:

$$(0041) = 04 \quad \text{(index)}$$
$$(0344) = CF \quad \text{(original value)}$$

Result:

$$[0340 + (0041)] = (0344) = (0344) + 1 = D0$$

How would you change your program to add 10 (hex) instead of 1? That is, the result should now be (starting from the original example)

$$[0340 + (0041)] = (0344) = (0344) + 10 = DF$$

PROBLEM 7-14

Place the value from memory location 0042 in the accessed memory location.

Example:

$$(0041) = 06 \quad \text{(index)}$$
$$(0042) = 3F \quad \text{(value)}$$

Result:

$$[0340 + (0041)] = (0346) = (0042) = 3F$$

How would you make your program replace the old value only if the new one is larger? Assume that the numbers are unsigned. This procedure would be necessary if the results represented the worst cases for a set of tests or scaling values for a set of plots.

PROBLEM 7-15

Change the answer to Problem 7-14 so that it obtains the starting address of the array from memory locations 0043 and 0044.

Example:

(0041) = 06 (index)

(0042) = 3F (value)

(0043) = 80 (LSBs of starting address)

(0044) = 03 (MSBs of starting address)

Result:

$$[(0044)\,(0043) + (0041)] = [0380 + (0041)] = (0386) = 3F$$

PROBLEM 7-16

Assume that each element of the array is 2 bytes long and clear the appropriate element.

Example:

(0041) = 03

Result:

$$[0340 + 2 \times (0041)] = (0346) = 00$$

$$[0340 + 2 \times (0041) + 1] = (0347) = 00$$

Hint: Use ASL to double the element number. Remember to clear both the indexed address and the next higher address (see Problem 6-3). Be careful of the fact that you cannot apply ASL to an index register. Assume that the index is less than 128, so doubling it does not produce a carry. How would you change your program so that it obtains the starting address of the array from memory locations 0042 and 0043?

PROBLEM 7-17

Assume that each element of the array is 2 bytes long and place the contents of memory locations 0042 and 0043 in the appropriate element. Place the contents of memory location 0042 in the byte at the lower address and the contents of 0043 in the next byte.

Example:

(0041) = 03 (index)

(0042) = D1 (LSBs of value)

(0043) = 3F (MSBs of value)

Result:

$$[0340 + 2 \times (0041)] = (0346) = (0042) = D1$$

$$[0340 + 2 \times (0041) + 1] = (0347) = (0043) = 3F$$

As in Problem 7-16, you may assume that the index is less than 128.

COUNTING SWITCH CLOSURES

PROBLEM 7-18

Write a program that counts how many times each switch attached to port A of the user 6530 device is closed. Only consider single switch closures and assume that all switches must be opened between closures. The steps required are (see Figure 7-4):

1) Initialize the array of counts by clearing all the elements.
2) Wait until a switch is closed.
3) Debounce the switch closure.
4) Identify the switch.
5) Add 1 to the count for that switch.
6) Wait until all switches are open.
7) Debounce the switch opening.
8) Return to step 2.

Use address 0340 through 0347 for the number of times switches 0 through 7 are closed.

ARRAYS OF ADDRESSES

We can extend the use of arrays even further by considering arrays that consist of addresses rather than data. By choosing a particular element from such an array, we are actually choosing one of a set of addresses to use in data transfers. Thus we want a combination of indexing and indirection, just as in the postindexed addressing mode we discussed in Laboratory 6, but here we want the indirection to occur after the indexing.

The 6502 microprocessor provides the required combination; this addressing mode is called *indexed indirect addressing* or *preindexing*, since the indexing occurs before the indirection. This mode assumes the use of page zero (for the indirect address) and index register X, much as indirect indexed addressing assumes the use of page zero and index register Y. A typical instruction is LDA ($40,X), which loads the accumu-

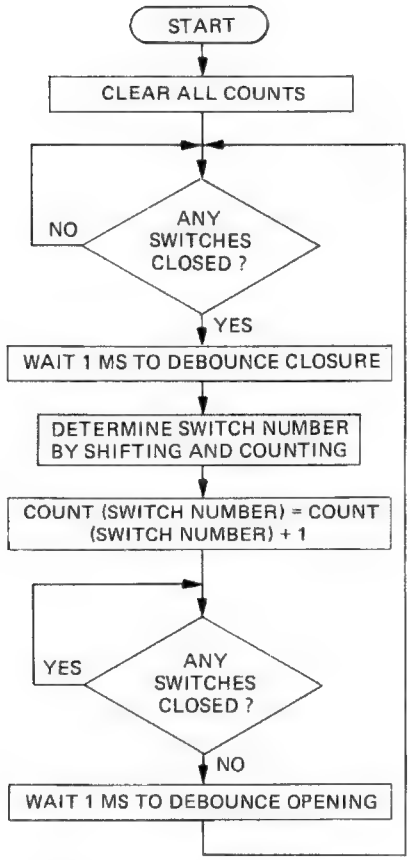


FIGURE 7-4. Flowchart for cumulative counts program.

lator from the address obtained indirectly by adding index register X to the base address 0040 (hex). Thus the result is

$$(A) = [(0040) + (X) + 1)\ (0040 + (X))]$$

where the indirect address, as usual, occupies two memory locations. If, for example, (X) = 04, then the indirect address is in memory locations 0044 and 0045. If (0044) = 86, (0045) = 03, and (0386) = E4, the result of LDA ($40,X) is

$$\begin{aligned}(A) &= [(0040 + 04 + 1)\ (0040 + 04)] \\ &= [(0045)\ (0044)] = (0386) = E4\end{aligned}$$

Be careful when you use preindexing; the array starting at the base address on page zero must consist of addresses and you can refer only to even-numbered elements. (Why?) Note that the entire array of addresses must be on page zero; thus your program must initialize the entire array and you can use preindexing only sparingly. After all, there is only a limited amount of page zero to go around.

Observing these restrictions is the programmer's responsibility. The 6502 microprocessor does not indicate an error if your program refers to an odd-numbered element in an array of addresses or indexes off the end of page zero (the processor does, however, provide wraparound with no carry). The processor obviously does not care if the indirect address is reasonable or is even implemented in a particular computer.

Indexed indirect addressing is often used to assign device numbers to the actual I/O addresses or the starting addresses of I/O routines for a particular system. The system operator or programmer can then refer to I/O devices by number (e.g., he or she can tell the system to "print on device #2" or "read data from device #3"). A table (called an *I/O device table*) contains the actual I/O addresses corresponding to the numbers. We call the device number to which the operator or programmer refers a *logical device.* The actual I/O device used in data transfers is called a *physical device.*

The advantages of maintaining this distinction are:

1) Programmers and operators need not be concerned about actual I/O addresses. They can refer to devices by number without worrying about changes in the underlying system or variations resulting from different models or types of systems.

2) Programs can be written for a set of device numbers and can be made to work on a particular system by constructing an appropriate I/O device table. Thus a single program can be modified easily to work on computers with different numbers and arrangements of peripherals.

3) A programmer or operator can change the actual I/O addresses by modifying the device table. For example, the operator may want the results of a test run to be shown on a CRT display rather than printed. Similarly, the operator may assign a control console to simulate I/O devices that are unavailable or malfunctioning. Implementing these changes works much like switching the output of a stereo amplifier from the front speaker to the back speaker or from a set of headphones to a loudspeaker system. Nothing is changed except the I/O assignments.

The following program sends the data from memory location 0040 to either device 0 (the LEDs attached to user 6530 port B) or device 1 (KIM seven-segment display 1), depending on the contents of memory

location 0041. We have placed the device table in memory locations 0050 through 0053.

```
        LDA   #$FF      ;MAKE DISPLAY PORTS OUTPUT
        STA   $1703
        STA   $1741
        STA   $1702     ;TURN OFF ATTACHED LEDS
        LDA   #0        ;TURN OFF ALL SEGMENTS
        STA   $1740
        LDA   #$08      ;ACTIVATE LEFTMOST KIM DISPLAY
        STA   $1742
        LDA   $41       ;GET DEVICE NUMBER
        ASL   A         ;DOUBLE DEVICE NUMBER FOR INDEXING
        TAX
        LDA   $40       ;GET DATA
        STA   ($50,X)   ;SEND DATA TO PHYSICAL DEVICE
HERE    JMP   HERE      ;ENDLESS LOOP
```

Program 7-5 is the hexadecimal version of the program and the device table. We must double the device number to access the correct entry, since each entry in the table is 2 bytes long. Turning off all the attached LEDs (by placing FF in memory address 1702) and all the segments (by placing 00 in memory address 1740) makes the results more obvious. Remember that the LEDs attached to the user 6530 device operate in negative logic (see Laboratory 3), whereas the KIM displays operate in positive logic (see Laboratory 5). Run Program 7-5 with (0040) = 55 and (0041) = 00. What happens if you change 0041 to 01 and run the program again?

PROBLEM 7-19

Change Program 7-5 so that it sends the contents of memory location 0040 to the output device assigned to the device number in memory location 0042 and the contents of memory location 0041 to the other output device.

Example:

(0040) = F0 (data for primary device)

(0041) = 0F (data for secondary device)

(0042) = 00 (number of primary device)

Result:

(1702) = F0 (device #0 gets primary data)

(1740) = 0F (device #1 gets secondary data)

What happens if you change (0042) to 01? What happens if you invert the order of the addresses in the device table?

PROGRAM 7-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#$FF
0201	FF			
0202	8D		STA	$1703
0203	03			
0204	17			
0205	8D		STA	$1741
0206	41			
0207	17			
0208	8D		STA	$1702
0209	02			
020A	17			
020B	A9		LDA	#0
020C	00			
020D	8D		STA	$1740
020E	40			
020F	17			
0210	A9		LDA	#$08
0211	08			
0212	8D		STA	$1742
0213	42			
0214	17			
0215	A5		LDA	$41
0216	41			
0217	0A		ASL	A
0218	AA		TAX	
0219	A5		LDA	$40
021A	40			
021B	81		STA	($50,X)
021C	50			
021D	4C	HERE	JMP	HERE
021E	1D			
021F	02			
0050	02	DEVICE 0 (ADDRESS 1702)		
0051	17			
0052	40	DEVICE 1 (ADDRESS 1740)		
0053	17			

LONG ARRAYS

We have discussed indirect indexed addressing both here and in Laboratory 6. This addressing mode lets us specify the base address of an array as the contents of two memory locations on page zero. Thus we can change that base address freely, since it is in RAM rather than in ROM.

Indirect indexed addressing has another important use in 6502 programming. It lets us handle arrays that are more than 256 bytes long. Such arrays cannot be processed using ordinary indexed addressing since the 6502's index registers are only 8 bits long and changing the base address is generally undesirable or impossible if the program is in ROM. We can, however, handle long arrays using a counter and a pointer stored in RAM, although at the cost of reduced execution speed.

The following program (see Program 7-6 for a hexadecimal version) clears a section of memory starting at the address in memory locations 0040 and 0041 and continuing through a number of locations given by the contents of memory locations 0042 and 0043 (in complemented form).

PROGRAM 7-6

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A9		LDA	#0
0201	00			
0202	A8		TAY	
0203	91	CLEAR	STA	($40), Y
0204	40			
0205	C8		INY	
0206	D0		BNE	COUNT
0207	02			
0208	E6		INC	$41
0209	41			
020A	E6	COUNT	INC	$42
020B	42			
020C	D0		BNE	CLEAR
020D	F5			
020E	E6		INC	$43
020F	43			
0210	D0		BNE	CLEAR
0211	F1			
0212	00		BRK	

```
        LDA   #0          ;DATA = ZERO
        TAY               ;STARTING INDEX = ZERO
CLEAR   STA   ($40),Y     ;CLEAR A BYTE
        INY               ;MOVE TO NEXT BYTE
        BNE   COUNT
        INC   $41         ;AND TO NEXT PAGE IF NEEDED
COUNT   INC   $42         ;COUNT BYTES
        BNE   CLEAR
        INC   $43         ;WITH CARRY TO MSB
        BNE   CLEAR
        BRK
```

Since neither INY nor INC affects the CARRY flag, we can recognize a carry only by examining the ZERO flag. When Y is incremented to zero, we move on to the next page by adding 1 to the more significant byte of the base address (memory location 0041). Both the base address and the 16-bit counter are stored in the usual 6502 arrangement with the less significant byte first. Try running this program with the following sample data:

(0040) = 80 (LSBs of initial base address)

(0041) = 02 (MSBs of initial base address)

(0042) = C0 (LSBs of complemented count)

(0043) = FE (MSBs of complemented count)

Which memory addresses are cleared? We count up here rather than down, since incrementing a 16-bit counter is much simpler than decrementing one. Why? Counting up does require us to calculate the negative (two's complement) of the number of locations we want to process and store the complemented count in memory locations 0042 and 0043. Laboratory 4 contains a brief description of the handling of 16-bit counters in memory.

PROBLEM 7-20

What values must you place in memory locations 0040 through 0043 to make Program 7-6 clear memory addresses 024C through 03EF inclusive? You can use Table 2-3 to calculate a two's complement.

PROBLEM 7-21

Extend Program 7-6 so that it clears one element of a long array. Assume that the starting address of the array is in memory locations 0040 and 0041 and the 16-bit index is in memory locations 0042 and 0043.

Example:

(0040) = 00 (LSBs of starting address)

(0041) = 02 (MSBs of starting address)

(0042) = 80 (LSBs of index)

(0043) = 01 (MSBs of index)

Result:

(0380) = 00, since the address that is cleared is the sum of the starting address and the index (i.e., 0200 + 0180 = 0380).

Hint: One way to solve this problem is to add the more significant bytes in the accumulator and use the sum as part of an indirect address.

PROBLEM 7-22

Extend the answer to Problem 7-16 so that it works properly even if the array occupies more than 256 bytes. That is, you want to clear a 2-byte element of an array; the starting address of the array is in memory locations 0042 and 0043 and the index (an 8-bit number) is in memory location 0041.

Example:

(0041) = C4 (index)

(0042) = 00 (LSBs of starting address)

(0043) = 02 (MSBs of starting address)

Result:

$[0200 + 2 \times (0041)] = (0388) = 00$

$[0200 + 2 \times (0041) + 1] = (0389) = 00$

KEY POINT SUMMARY

1) Arrays can be formed by using an index to determine which element is being filled. Either a maximum length or a terminator can be used to conclude the formation.

2) On the 6502 microprocessor, an index register can be used to hold the element number or index. Then you can use indexed addressing

to access the required memory location. The simplest procedure is to start the index register at the number of elements and use the lowest address of the array minus 1 as the base.

3) To reach a particular element in an array, you must know the starting address of the entire array and the element number or index. The access procedure on the 6502 microprocessor is simple: load the index register with the element number and use indexed addressing with the starting address as the base.

4) You can handle an array with multibyte elements by multiplying the element number times the size of an element and then adding the product to the starting address. Multiplication by a small integer can be implemented as a series of additions. An arithmetic left shift is equivalent to a multiplication by 2.

5) The indexed indirect addressing mode allows the programmer to select one of a set of indirect addresses to be used in transferring data. This mode is often used to convert the logical device numbers to which a program refers into the physical I/O addresses for a particular computer.

6) Maintaining a distinction between physical and logical I/O devices allows you to write programs that will run on a variety of computers. All that one must do to make the programs run on a particular system is construct the appropriate I/O device table that converts logical device numbers into physical I/O addresses. Changing the table lets the programmer or operator vary the actual I/O addresses used without changing the program. Thus the programmer or operator can easily direct test results to a console, choose whether outputs should be displayed or printed for permanent records, or switch between local and remote control.

7) Arrays longer than 256 bytes are awkward to handle on the 6502 microprocessor because of its 8-bit index registers. However, you can handle long arrays by using the indirect indexed addressing mode and incrementing the more significant byte of the indirect address after processing each 256-byte section. You must also provide a 16-bit counter in two memory locations. The simplest way to count is to count up from the two's complement of the length of the array. Incrementing the less significant byte of a 16-bit counter sets the ZERO flag to 1 if a carry occurs. That flag can then be used to determine when to increment the more significant byte of the counter.

Laboratory 8

Designing and Debugging Programs

PURPOSE

To learn the fundamental approaches to program design and debugging.

PARTS REQUIRED

None.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502*, Howard W. Sams, Indianapolis, IN, 1980, pp. 43-45, 349-356.

J. K. Hughes and J. I. Michtom, *A Structured Approach to Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, Chapter 6.

L. A. Leventhal, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapters 13 through 15.

L. J. Scanlon, *6502 Software Design*, Howard W. Sams, Indianapolis, IN, 1980, pp. 92-107, 172-176.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.*, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 377-378 (address vectors, BRK instruction), 378-387 (program writing).

W. J. Weller, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapter 19, Appendixes D and F.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 39-42 (register storage and operating programs), 67-69 (single-step mode).

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 144-146 (BRK instruction).

WHAT YOU SHOULD LEARN

1) The stages of software development.
2) The standard flowcharting symbols.
3) How to use flowcharting as a design tool.
4) How to draw flowcharts.
5) The common debugging tools.
6) How to insert breakpoints.
7) How to use the single-step mode.
8) How to debug simple programs systematically.
9) Some common errors in 6502 assembly and machine language programs.

TERMS

Breakpoint—a condition specified by the user under which program execution is to end temporarily, used as an aid in debugging. The specification of the conditions under which execution will end is referred to as *setting breakpoints* and the deactivation of those conditions is referred to as *clearing breakpoints.*

Bug—error or flaw.

Coding—writing instructions in a computer language.

Data flowchart—a flowchart that traces the path of a particular type of data through a program.

Debugger—a program that helps in locating and correcting errors in a user program. Some versions are referred to as dynamic debugging tools or DDT after the famous insecticide.

Debugging—locating and correcting errors in a system.

Dump—a facility that displays the contents of an entire section of memory or group of registers on an output device.

Editor—a program that manipulates text material and allows the user to make corrections, additions, deletions, and other changes.

File—a collection of related information that is treated as a unit for purposes of storage or retrieval.

Flowchart—a graphic representation of a procedure or computer program.

Modular programming—a programming method whereby the overall program is divided into logically separate sections or *modules.*

Murphy's Law—the famous maxim that "Whatever can go wrong, will." No one has ever doubted its applicability to computer programming.

No-op (or **no operation**)—an instruction that does nothing other than increment the program counter.

Problem definition—the determination of exactly what requirements a system must meet.

Program design—the design of a computer program to meet the requirements specified in the problem definition.

Program flowchart—a flowchart that traces the operation of the program.

Single step—a facility that allows a program to be executed one step at a time.

Structured programming—a programming method whereby all programs consist of structures from a limited but complete set; each structure should have a single entry and a single exit.

Testing—checking a system to ensure that it meets the requirements specified in the problem definition.

Text file—a file consisting of symbolic characters rather than numbers (a *data file*) or computer instructions (a *program file*).

Top-down design—a design method whereby the overall structure is designed first and parts of the structure are subsequently defined in greater detail.

Trace—a facility that displays all or part of the status of a computer at specified points while a program is being executed.

Unsigned number—a number in which all the bits are used to represent magnitude.

6502 INSTRUCTIONS

NOP—no operation; do nothing except increment the program counter. The hexadecimal code is EA.

STAGES OF SOFTWARE DEVELOPMENT

So far, we have dealt with short programs and we have started with initial versions. In real applications, of course, programming is far more difficult and uncertain. We cannot deal with all its aspects here, but we will discuss design and debugging in enough detail so that you should be able to write and run programs of moderate size.

In fact, software development consists of a series of stages:

1) *Problem definition,* in which you determine exactly what requirements the program must meet.
2) *Program design,* in which you provide a "blueprint" for the program that will meet those requirements.
3) *Coding,* in which you translate the program design into computer instructions. Note that writing instructions is only one of many stages.
4) *Debugging,* in which you locate and correct errors in the program.
5) *Testing,* in which you ensure that the program meets the requirements of the problem definition.
6) *Documentation,* in which you describe the program so that it can be used, maintained, and extended.
7) *Maintenance,* in which you correct and upgrade the program to handle problems found in field use.
8) *Extension* and *redesign,* in which you upgrade the program to handle new requirements or new tasks.

The life cycle of a computer program is thus similar to the life cycles of other engineering projects. As usual, definition, design, debugging, testing, documentation, and maintenance typically require far more time and effort than does the writing of a program (or the construction of a hardware prototype). As with most projects, you should spend an adequate amount of time in the definition and design stages and proceed cautiously and systematically through the debugging and testing stages.

We will concentrate here on simple problems in which

1) The requirements have already been determined.
2) The program can be designed with a flowchart.
3) Debugging and testing are virtually the same.
4) The later stages (e.g., documentation, maintenance) can be ignored. This is certainly not the case in actual practice; maintenance is often the most time consuming and costly stage of all.

FLOWCHARTING

Flowcharting is the traditional method for designing programs. Its advantages are that it shows the structure of the program in a pictorial form, it has a set of standard symbols (see Figure 8-1), and it is comprehensible even to those who are unfamiliar with computer programming.

We strongly recommend the following approach to flowcharting:

1) Start by drawing a rough flowchart. Don't worry about how artistic or how complete it is.

2) Check the flowchart for obvious errors and possible improvements. Be sure that all branches lead somewhere, all variables are initialized or derived, and all decisions make sense (try a simple case if you are not sure).

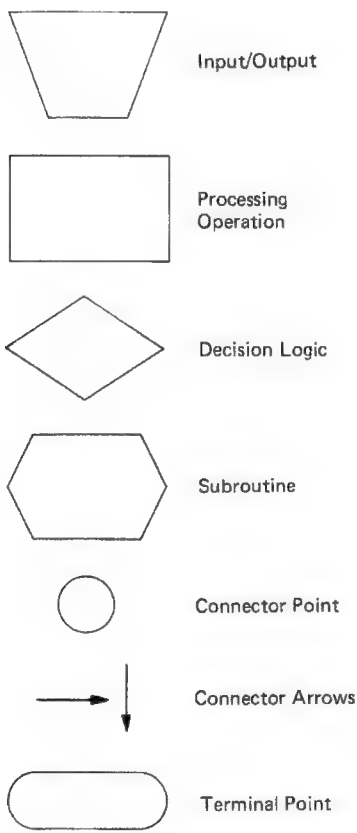


FIGURE 8-1. Standard flowchart symbols.

3) Next, revise your flowchart. Again, do not worry about the details or the appearance. It is now time to write an initial program.

4) When you finish coding, debugging, and testing the program, draw a clear, current flowchart as part of the final documentation.

Don't let the flowchart become a burden. There is no systematic way to debug a flowchart or to code from it. You might as well work on the actual program as draw additional versions of the flowchart. If the program logic is complex, flowcharting is not a satisfactory design method. You must then consider such methods as modular programming, structured programming, and top-down design, which are described in the references.

FLOWCHARTING EXAMPLE 1—COUNTING ZEROS

Purpose:

Count the number of 0's in memory locations 0340 through 0347 and place the result in memory location 0040.

Sample Case:

(0340) = 37

(0341) = 40

(0342) = 00

(0343) = 5E

(0344) = 00

(0345) = D1

(0346) = 39

(0347) = 00

Result:

(0040) = 03, since there are 0's in memory locations 0342, 0344, and 0347.

Our initial flowchart is Figure 8-2. A hand check shows that we forgot to initialize NZERO and that we inverted the branches after deciding whether the memory location contains zero. Figure 8-3 shows the revised flowchart which we will use as a guide in writing the program. We have not checked the flowchart in detail; we will describe how to debug the actual program later.

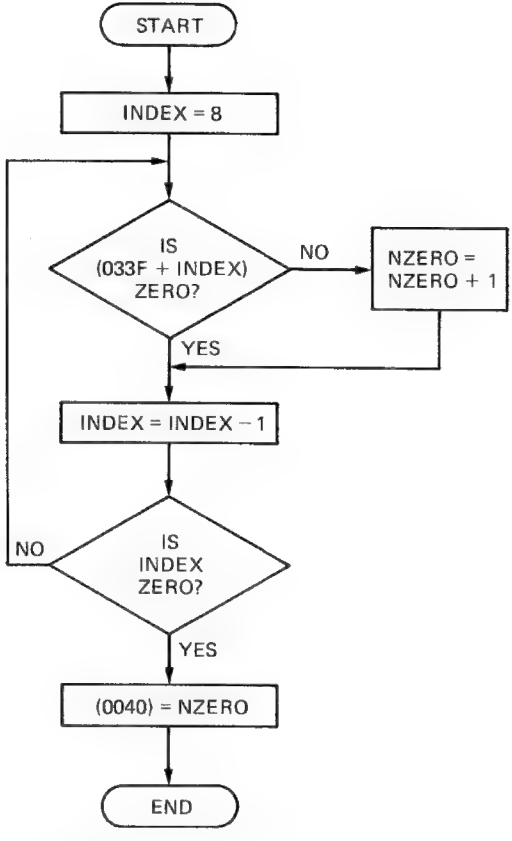


FIGURE 8-2. Initial flowchart for the zero counting program.

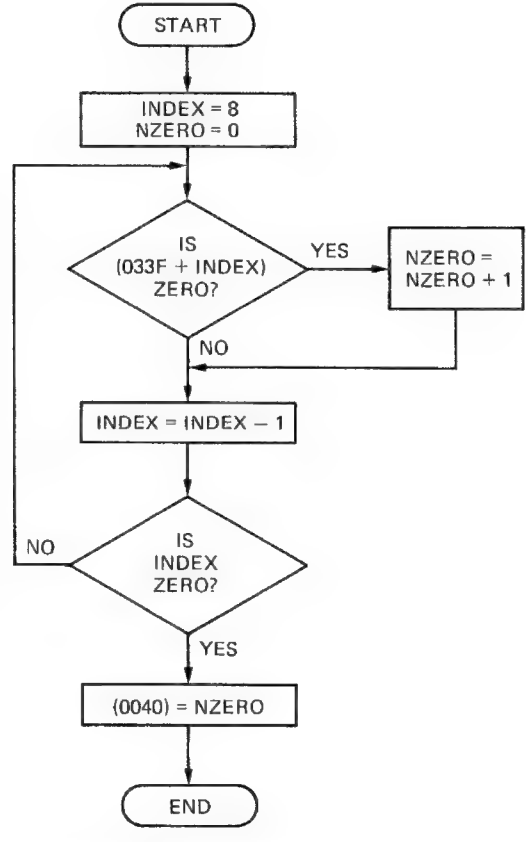


FIGURE 8-3. Revised flowchart for the zero counting program.

PROBLEM 8-1

Draw a flowchart for a program that counts the number of values in memory locations 0340 through 0347 that exceed the value in memory location 0041. Place the result in memory location 0040. Assume that all numbers are unsigned.

Example:

(0041) = 67 (threshold)
(0340) = 35 (first value)
(0341) = 4A
(0342) = A9
(0343) = 67
(0344) = B3
(0345) = 69
(0346) = 14
(0347) = 33 (last value)

Result:

(0040) = 03, since memory locations 0342, 0344, and 0345 contain values larger than the one in memory location 0041.

PROBLEM 8-2

Draw a flowchart for a program that searches an array in memory locations 0340 through 0347 for a nonzero value. If one is found, the search terminates, the value is placed in memory location 0041, and the memory location from which it was taken is cleared. If all the values are zero, memory location 0041 is cleared.

Example 1:

(0340) = 07
(0341) = 04
(0342) = 12
(0343) = 00
(0344) = 13
(0345) = 06
(0346) = 00
(0347) = 00

Result:

(0041) = 06, since that is the first nonzero value encountered. (0345) = 00, since the element removed from the array is then cleared. Note that we are working backward through the array in accordance with our usual practice on the 6502.

Example 2:

(0340) through (0347) = 00

Result:

(0041) = 00, since all the elements are zero.

FLOWCHARTING EXAMPLE 2—MAXIMUM VALUE

Purpose:

Find the largest unsigned binary number in memory locations 0340 through 0347 and store it in memory location 0040.

Sample Case:

(0340) = 37
(0341) = 40
(0342) = 88
(0343) = 5E
(0344) = 2B
(0345) = D1
(0346) = 39
(0347) = AE

Result:

(0040) = D1, since that is the largest unsigned binary number.

Our initial flowchart is Figure 8-4. A simple hand check shows that we forgot to initialize MAX and that we forgot to save the new maximum. In fact, as you will probably see if you implement the program, even the flowchart of Figure 8-5 is far from optimal.

PROBLEM 8-3

Draw a flowchart for a program that finds the smallest unsigned binary number in memory locations 0340 through 0347 and stores it in memory location 0040.

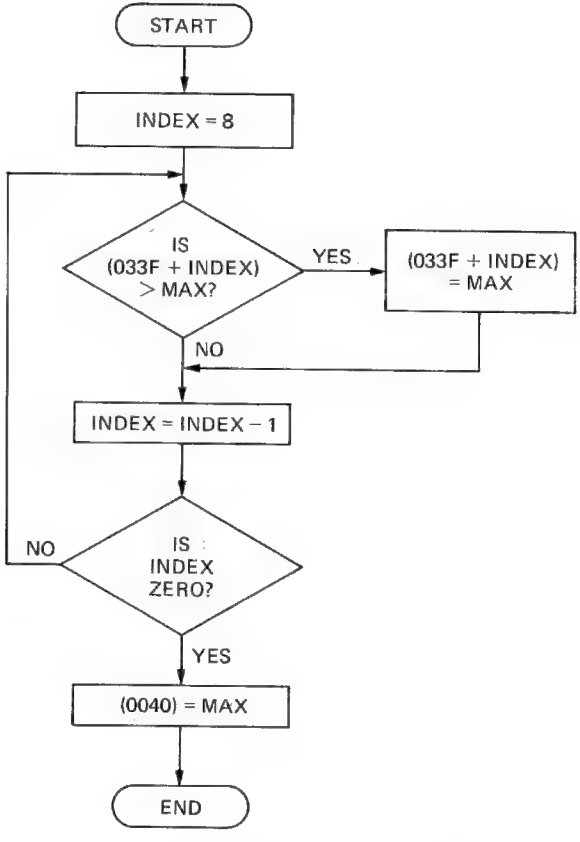


FIGURE 8-4. Initial flowchart for the maximum program.

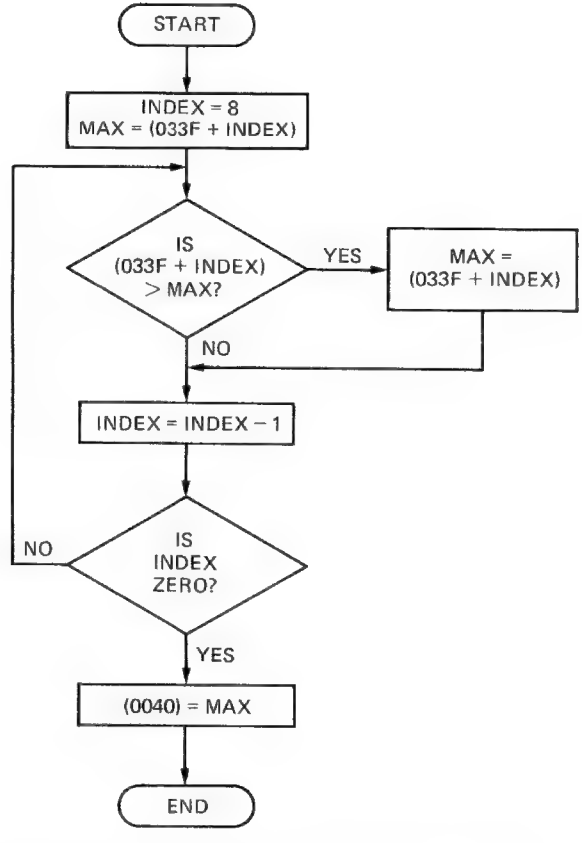


FIGURE 8-5. Revised flowchart for the maximum program.

Example:

(0340) = 37

(0341) = 40

(0342) = 88

(0343) = 5E

(0344) = 2B

(0345) = D1

(0346) = 39

(0347) = AE

Result:

(0040) = 2B, since that is the smallest unsigned binary number.

PROBLEM 8-4

Draw a flowchart for a program that finds the largest unsigned 16-bit binary number in memory locations 0340 through 0347 and stores it in memory locations 0040 and 0041. All numbers are stored in the standard 6502 style with the least significant bits first (i.e., at the lower address).

Example:

(0340) = 40 (LSBs of first number)

(0341) = 88 (MSBs of first number)

(0342) = 5E (LSBs of second number)

(0343) = 2B (MSBs of second number)

(0344) = D1 (LSBs of third number)

(0345) = 39 (MSBs of third number)

(0346) = AE (LSBs of fourth number)

(0347) = A6 (MSBs of fourth number)

Result:

(0040) = AE (LSBs of maximum)

(0041) = A6 (MSBs of maximum)

since A6AE is the largest unsigned 16-bit binary number (A6AE is larger than 39D1, 2B5E, or 8840).

FLOWCHARTING EXAMPLE 3—VARIABLE DELAY

Purpose:

A switch attached to bit 7 of port A of the user 6530 device acts as a DELAY switch. When the switch is closed, the processor waits for the number of seconds (0 through 63) specified by the switches attached to bits 0 through 5 of port A.

Sample Case:

The switches attached to bits 0 through 5 of user 6530 port A produce a reading of 011110 (1 = open, 0 = closed). When switch 7 is closed, the processor will wait for 30 s (011110 binary = 1E hex = 30 decimal). Figure 8-6 contains the initial flowchart. A check shows that the flowchart is incorrect if the length of the delay is zero. (Why?) Figure 8-7 contains the revised flowchart.

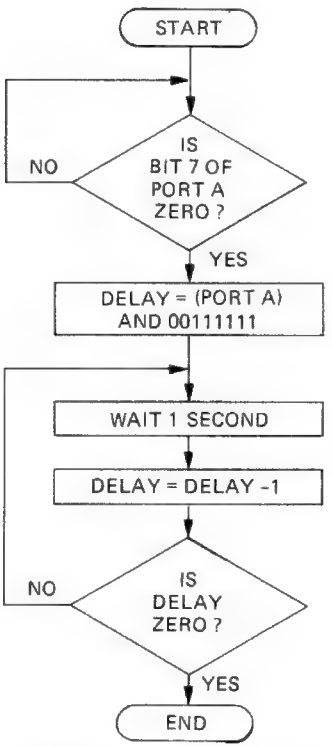


FIGURE 8-6. Initial flowchart for variable delay program.

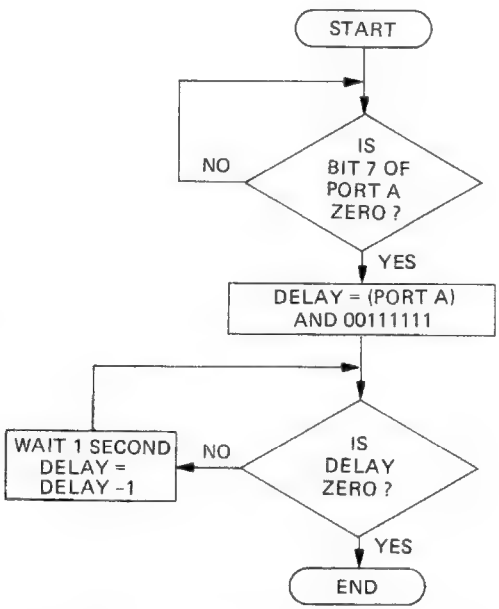


FIGURE 8-7. Revised flowchart for variable delay program.

PROBLEM 8-5

Draw a flowchart for an extended program that uses switch 6 to determine if the delay is to be in seconds (switch open) or in milliseconds (switch closed).

PROBLEM 8-6

Draw a flowchart for an extended program that uses switch 6 to escape from the delay. The program should check that switch every tenth of a second to see if the delay should be continued. If that switch is closed, the delay is immediately ended.

DEBUGGING TOOLS

There are several important debugging tools, most of which are available in the KIM monitor. The tools are:

1) A *breakpoint,* which allows the user to stop the program and examine its current status. Breakpoints help you localize an error within a section of a program and pass through sections that you know are correct.

2) A *single-step* facility, which allows the user to execute the program one step at a time. The KIM monitor provides this capability. A single-step mode helps you pinpoint an error.

3) A *dump,* which displays the contents of an entire section of memory on an output device. The KIM teletypewriter monitor has this capability. A dump lets you examine many values simultaneously. The KIM keyboard/display monitor lets you examine only one register or memory location at a time.

4) A *trace,* which displays the current contents of registers and memory locations while the program is executing. Traces provide a detailed accounting of the operation of the program. The KIM monitor provides tracing in conjunction with the single-step mode, but you can see the values only by examining the memory locations in which the monitor stores the user registers (see Table 1-1).

BREAKPOINTS

You can insert (set) a breakpoint in a KIM program by replacing an operation code with a BRK instruction (00 hex). As we noted earlier, BRK returns control to the monitor, saving the current values of the registers in the memory locations listed in Table 1-1. Once you are back in the monitor, you can examine or change registers and memory locations.

Resuming your program, however, is tricky. If you want the program to continue as if the breakpoint had not occurred, you must make sure

that the computer executes the instruction whose operation code you replaced and proceeds to the next instruction in its normal sequence. Thus we need to back up the program counter and reinsert the original operation code or simulate its execution. A complicating factor, as we mentioned in Laboratory 1, is that BRK increments the program counter by 2 even though it is only a one-byte instruction.

The possible solutions are the following:

1) If you wish to remove the BRK instruction, then you must restore the operation code you replaced and resume execution at an address two less than the one displayed at the breakpoint. Your program will resume where it left off, but the breakpoint will not be available for later use.

2) If you want to leave the BRK instruction in your program, then you must simulate the execution of the operation code you replaced. That is, you must change the user registers as if the original instruction had been executed.

When you press PC and GO (after changing the registers), the KIM will resume your program at the address two larger than the one containing the BRK instruction. You must ensure that the computer does not skip an instruction or execute a byte that does not contain an operation code. The way to do this depends on how many bytes there are between the operation code you replaced and the next operation code—that is, on how long the current instruction is. There are three possibilities:

a) The current instruction is 1 byte long (e.g., TAX or ASL A). Then you must decrement the program counter by 1 before resuming or else the computer will skip the next operation code.

b) The current instruction is 2 bytes long (e.g., LDA $40 or DEC $40). Then no changes are necessary, since the program counter contains the address of the next operation code.

c) The current instruction is 3 bytes long (e.g., LDA $1700 or INC $1742). Then you must increment the program counter by 1 before resuming or else the computer will attempt to execute the third byte of the instruction. Alternatively, you can replace the third byte with a NOP (EA hex). NOP (no operation) is a space filler; executing it does nothing except increment the program counter by 1.

Take the following precautions when you set breakpoints in a program:

1) Interpret the results carefully. Remember that the computer has not executed the instruction you replaced and the values you observe reflect only the previous instructions. Furthermore, if you do not back up the program counter, you must simulate the current instruction before resuming the program.

2) Set breakpoints only at addresses that contain operation codes. Replacing data or parts of addresses with BRK instructions will obviously result in chaos.

3) Do not resume your program in the middle of an instruction. You may have to adjust the program counter or insert some NOPs. You may, of course, leave extra locations initially in which you can place either BRK or NOP instructions.

Microcomputer development systems usually have far more extensive breakpointing facilities than the KIM has. Such systems typically let the user specify where he or she wishes to set a breakpoint. The system software then handles the actual insertion of the breakpoint and the proper resumption of the user's program. Other useful features include the ability to clear individual breakpoints and to set breakpoints on conditions such as the following:

1) Whenever a particular operation code is executed. The usual ones selected are those that perform input or output operations.
2) Whenever a particular memory address is accessed.
3) Whenever a particular sequence of instructions is executed.
4) Whenever the instruction at a particular ROM address is executed. Obviously, neither you nor the monitor program can replace an instruction in ROM.
5) Whenever a particular signal or combination of signals occurs. This is a purely hardware breakpoint.

Still more advanced features include the ability to combine the simpler features and to count the number of occurrences. Note the parallels between the ways in which one can set breakpoints and the ways in which one can specify triggering events on an oscilloscope.

SINGLE-STEP MODE

You can put the KIM in a single-step mode by moving the SINGLE-STEP slide switch (in the top row of the keyboard) to the ON position. Now each time you press the GO key, the KIM will execute a single instruction (which may occupy one to three memory locations) and will display the

next operation code and its address. Remember that the KIM displays the next operation code, not the one that it has just executed.

When you are using the single-step mode, be sure to press GO rather than + to execute the next instruction. Pressing + will display the next higher memory address, but will not cause the computer to execute an instruction in your program. Be careful; this is an easy error to make.

Besides moving the SINGLE-STEP switch to the ON position, you must also place its return address in memory locations 17FA and 17FB. The usual procedure is to place 1C00 there just as in activating the ST (stop) key. That is,

(17FA) = 00

(17FB) = 1C

Important Note: Remember that the KIM stores the user registers in the following memory locations (see Table 1-1):

00EF	Less significant byte of program counter
00F0	More significant byte of program counter
00F1	Status register
00F2	Stack pointer
00F3	Accumulator
00F4	Index register Y
00F5	Index register X

As we noted earlier, you should tape a copy of Table 1-1 to your KIM.

DEBUGGING EXAMPLE—COUNTING ZEROS

From the flowchart in Figure 8-3, we write the following program for counting zeros:

```
        LDX     8
        LDY     0
        LDA     $0340,X
CNTZ    BEQ     CHCNT
        INY
CHCNT   DEC     $0340,X
        BNE     CNTZ
        LDY     $40
        BRK
```

Program 8-1 is the hexadecimal version.

PROGRAM 8-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A6		LDX	8
0201	08			
0202	A4		LDY	0
0203	00			
0204	BD		LDA	$0340,X
0205	03			
0206	40			
0207	F0	CNTZ	BEQ	CHCNT
0208	03			
0209	C8		INY	
020A	DE	CHCNT	DEC	$0340,X
020B	40			
020C	03			
020D	D0		BNE	CNTZ
020E	F8			
020F	A4		LDY	$40
0210	00		BRK	

Enter this program but *don't run it. Important rule:* Never just let a program run the first time. The program may easily write over itself or cause other problems. Expect errors and plan for them.

Let us first place a breakpoint at the end of the initialization; that is, replace the operation code of LDA $0340,X with a BRK instruction (00 in memory location 0204 instead of BD). The results at this point should be

(Y) = 00 (number of zeros found)

(X) = 08 (starting value of index)

Run the program with the breakpoint in it. What are your results? Ours are

(Y) = 36

(X) = 16

Obviously, the program is far from correct. Your results may be somewhat different from ours since the program is way off base.

Let us go back and try the single-step mode. We move the single-step switch to the ON position and start executing the program at memory address 0200. The CPU executes the first instruction in the program (LDX 8) and displays the address of the next instruction (0202) and the contents of that address (A4). We should observe

(X) = 08 (starting value of index)

Instead, the value we find in memory location 00F5 is

(X) = 16

Obviously, we have the wrong instruction. We cannot observe the actual execution of the instruction—all we see are the results. To see more would require a logic analyzer or a control panel that would display the contents of the address and data buses and the states of the various control signals.

Thus LDX 8 is wrong, but why? The simplest alternative would be LDX #8. That is, in fact, the instruction we want, since our aim is to load index register X with the number 8, not with the contents of memory location 0008. Always be careful of the difference between immediate and direct addressing, that is, between an address and the contents of that address.

So we replace the LDX zero page (A6) in memory location 0200 with LDX immediate (A2). The result of a single step is now

(X) = 08 (hurray!)

We can proceed to the next instruction by pressing PC (to restore the address 0202) and then GO. Sure enough, the result is still wrong (Y contains 36 instead of 00). We immediately suspect the same error as in the first instruction; we want LDY immediate, not LDY zero page. So we replace the A4 in memory location 0202 with A0. The results after two steps are now

(Y) = 00 (number of zeros found)

(X) = 08 (starting value of index)

Note the key points of this debugging exercise:

1) A breakpoint can tell you if an entire section of a program is correct or not.
2) A single-step mode (particularly if it allows you to trace the registers) can show you precisely what is wrong.
3) Most programmers are consistent and always make the same errors. You should soon be able to draw up a list of your own favorite mistakes; of course, it is much easier to construct the list than to change one's habits.

A SECOND BREAKPOINT

Remove the first breakpoint and place a second breakpoint at the end of the loop: that is,

(0204) = BD (remember to remove breakpoints that you no longer need)

(020D) = 00 (note that we have replaced the first byte of the 2-byte BEQ instruction)

At this point, we need some data. Clearly, the choices are to make memory location 0347 zero or nonzero. Let us first try

(0347) = 00

The results at the breakpoint should be

(A) = 00 (the element loaded from memory location 0347)

(Y) = 01 (one zero element has been found)

(X) = 07 (the original starting index reduced by 1)

Return the SINGLE-STEP switch to the OFF position and run the program. Our results are

(A) = 16

(Y) = 01

(X) = 08

Since the results are uniformly wrong (at least we are still consistent), we return to the single-step mode. After the LDA instruction we have

(A) = 16

Our first guess is that we have the wrong operation code. But a hand check shows that BD is correct (at least something is!). How about the order of the bytes in the address? Remember, the 6502 expects its addresses upside down. Sure enough, we have entered the address right side up (at least from our perspective). Note that computers simply work the way they were designed; they have no cultural or physiological preferences such as forward over backward, top to bottom over bottom to top, left to right over right to left, or positive numbers over negative numbers. Of course, computers really do not have a right, left, top, or bottom to worry about; people, on the other hand, are used to seeing things arranged in particular ways and become extremely confused when the order or direction is reversed.

So we reverse the address by setting

(0205) = 40

(0206) = 03

Now let us try again. After the LDA instruction the new result is

(A) = EC

Well, that is different but still wrong. An obvious pitfall in using the 6502's indexed addressing is having the base address off by 1. Let us examine memory location 0348 to see if that might be the problem here. Sure enough, we find

(0348) = EC

So the error is that we are working beyond the end of the array. We want LDA $033F,X instead of LDA $0340,X. We must change memory location 0205 from 40 to 3F.

With this last change, we get the correct value in the accumulator. But the next instruction is completely wrong. In the first place, the program branches when it shouldn't (the accumulator does contain zero). In the second place, the branch sends the processor to memory location 020C, which does not even contain an instruction. The first problem can be corrected by replacing BEQ (F0) with BNE (D0); the second problem can be corrected by counting from the end of the branch instead of from the beginning. The offset should be 01, not 03. Here we have managed to make a pair of common errors—inverting the decision logic and calculating a relative offset incorrectly.

Let us now try to get through one iteration. The results at memory location 020D are

(A) = 00

(Y) = 01

(X) = 08

Everything is fine except that index register X has not been decremented. A quick check shows that the instruction DEC $0340,X is not what we want at all; this instruction decrements a memory location, not index register X. What we need is simply DEX (CA). But this correction leaves us with 2 extra bytes of memory to fill (they did contain the address for DEC). How do we handle the extra bytes? The answer is that we fill them with NOPs (EA hex). Later we can compact the program by eliminating the NOPs. Now the program works properly through the second breakpoint.

Let us try another iteration with

(0346) = 00

(0347) = 01

Remember to replace the second breakpoint with the original instruction and place a third breakpoint in memory location 020B; that is,

(020B) = 00

(020D) = D0

Here the BRK instruction with its odd increment by 2 of the program counter works perfectly. The processor executes the BRK instruction in memory location 020B and returns to the monitor with the program counter equal to 020B + 2 = 020D. 020D is exactly the address we need to resume the program without executing the extra NOP we placed in memory location 020C. Unfortunately, things do not generally work out this well. Most of the time, we have to adjust the program counter and other registers to resume our program correctly. Placing extra NOPs in the original program simplifies breakpointing, but makes the program longer.

Execute the program starting at memory address 0200. It will reach the breakpoint with the correct values; that is,

(A) = 01 (the element loaded from memory location 0347)

(Y) = 00 (no zero element has been found)

(X) = 07 (the original starting index reduced by 1)

To proceed through the next iteration, simply press PC and GO. The computer finishes the second iteration instantaneously (at least by our standards—it actually takes about 10 μs) and reaches the breakpoint again. In fact, you may think you did not really press GO at all, since nothing appears to have changed. If the program were correct, the results would be

(A) = 00 (the element loaded from memory location 0346)

(Y) = 01 (one zero element has been found)

(X) = 06 (the original starting index reduced by 2)

The actual results are

(A) = 01

(Y) = 00

(X) = 06

The program is not loading the second value from memory. A hand check shows that we misplaced the label CNTZ. It should be one instruction earlier (i.e., attached to LDA $0340,X) so that the program will load a new element before it checks the ZERO flag. The corrected offset is

$$(020E) = F5$$

Now try the program on some test data, such as

1) (0340) – (0347) = 00.
2) Same as (1) except that (0340) = 01.
3) Same as (1) except that (0347) = 01.

The final version of the program is

```
        LDX   #8        ;NUMBER OF ELEMENTS = 8
        LDY   #0        ;NUMBER OF ZEROS FOUND = ZERO
CNTZ    LDA   $033F,X   ;IS NEXT ELEMENT ZERO?
        BNE   CHCNT
        INY             ;YES, INCREMENT NUMBER OF ZEROS FOUND
CHCNT   DEX
        BNE   CNTZ
        LDY   $40       ;SAVE NUMBER OF ZEROS FOUND
        BRK
```

Program 8-2 is a compacted version of our corrected program; we have removed the NOPs and adjusted the relative offsets appropriately.

PROBLEM 8-7

What errors still remain in Program 8-2? Correct them and run the final version for the three test cases that we just described.

PROGRAM 8-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A2		LDX	#8
0201	08			
0202	A0		LDY	#0
0203	00			
0204	BD	CNTZ	LDA	$033F,X
0205	3F			

PROGRAM 8-2 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0206	03			
0207	D0		BNE	CHCNT
0208	01			
0209	C8		INY	
020A	CA	CHCNT	DEX	
020B	D0		BNE	CNTZ
020C	F7			
020D	A4		LDY	$40
020E	00		BRK	

PROBLEM 8-8

Revise Program 8-2 so that it counts the number of positive elements in memory locations 0340 through 0347 and stores that number in memory location 0040. An element is positive if its most significant bit (bit 7) is zero, but its value is not zero.

Example:

(0340) = 01

(0341) = 80

(0342) = 7F

(0343) = FF

(0344) = 00

(0345) = 00

(0346) = 00

(0347) = 00

Result:

(0040) = 02, since there are positive elements in memory locations 0340 and 0342.

PROBLEM 8-9

Code, debug, and test Flowcharting Example 2, the maximum-value program.

PROBLEM 8-10

Code, debug, and test Flowcharting Example 3, the program that produces a variable delay. The following routine uses memory location 0040 and the index registers to provide a 1-s wait:

```
        LDA   #5        ;WAIT 1 SECOND
        STA   $40
DLY1    LDY   #$C8
DLY2    LDX   #$C8
DLY3    DEX
        BNE   DLY3
        DEY
        BNE   DLY2
        DEC   $40
        BNE   DLY1
```

You may want to check the delay constants for yourself to see that they are approximately correct.

PROBLEM 8-11

Code, debug, and test Problem 8-4, the 16-bit maximum.

As we have shown, deleting instructions or extra bytes from machine language programs is simple. All that you have to do is fill the unused memory locations with NOP instructions; NOPs have no effect on the program other than to increase its execution time slightly. On the other hand, inserting extra bytes is very difficult because it forces us to move all subsequent instructions. If, for example, we accidentally omit one byte at the start of a program, we will have to move each instruction to the next higher address. This is equivalent to reloading the program. Of course, Murphy's Law guarantees that omissions will always occur at the beginning of the program, rather than at the end.

Obviously, we cannot handle a long program in this way, since reloading thousands of locations is impractical. A common alternative is to prepare the assembly language program using an *editor* that allows us to make insertions, deletions, replacements, and other changes. Finally, the editor lets us save the completed program as a *text file* (in memory, on cassette, or on disk), which can then be assembled. If we find errors in the assembly or in the execution of the program, we can correct them by returning to the editor, making the appropriate changes in the text file, and reassembling the program. This approach, however, requires a computer that is more elaborate, more expensive, and has more substantial peripherals than our KIM-1.

COMMON PROGRAMMING ERRORS

You should watch for the following common errors in 6502 machine language programs:

1) Confusing data and addresses. Remember the difference between immediate and direct addressing; immediate addressing means that the data follows the operation code "immediately" while direct addressing means that the address of the data follows the operation code. Remember that the value stored in a memory location that is addressed directly or through indexing is not related to the effective address, base address, or index.

2) Inverting the order of the bytes in 2-byte addresses. Remember that the 6502 expects the less significant byte first.

3) Copying operation codes incorrectly. You should check programs before executing them.

4) Using the CARRY incorrectly. Remember that comparisons such as CMP, CPX, or CPY (as well as subtraction) set the CARRY if no borrow is required. The CARRY flag is an inverted borrow, not a true borrow as on most other microprocessors. Note also that addition and subtraction instructions (ADC and SBC) always include the CARRY. You must explicitly clear the CARRY before addition or set it before subtraction if you do not want its value to affect the result.

5) Inverting the logic of conditional branch instructions (e.g., using BCC instead of BCS or BNE instead of BEQ). Be particularly careful after a comparison instruction (CMP, CPX, or CPY).

6) Jumping to the wrong address. This often results in repeating or omitting initialization instructions or instructions that update indexes or indirect addresses.

7) Calculating relative offsets incorrectly. You should always check results you have obtained by hand. If you perform many such calculations, you should use a hexadecimal calculator.

8) Omitting addresses, offsets, or data. Watch for instructions such as JMP, which requires a full 16-bit address in the next 2 bytes of memory. Remember that absolute addressing modes always require 2 bytes of memory for their addresses and zero page modes (including preindexing and postindexing) always require 1 byte.

Some of these errors (e.g., 2, 3, and 7) will not occur if you use an assembler.

Other common errors are:

9) Failing to initialize counters, indexes, and indirect addresses.

10) Branching incorrectly when operands are equal (i.e., neither is larger). Note that comparing equal values *sets* the CARRY flag.

11) Overlooking trivial cases such as zero or one element in an array or table, no inputs, and so on.

12) Using the flags incorrectly. Typical examples are trying to use a flag that an instruction does not affect and ignoring changes produced by intermediate instructions. The only way to be sure of the effects of instructions on flags is to look them up in Table A1-1 or in a 6502 manual. Some of the 6502 instructions that often cause problems are: load instructions (they affect the ZERO and NEGATIVE flags), store instructions (they affect no flags at all), increment and decrement instructions (they do not affect the CARRY flag), and the BIT instruction (it sets the OVERFLOW and NEGATIVE flags from bits 6 and 7 of the addressed memory location, regardless of the contents of the accumulator).

13) Using the same register for more than one purpose without saving and restoring the different values. You should use page zero of memory (addresses 0000 through 00FF hexadecimal) as extra scratchpad registers, since locations on that page can be accessed quickly and efficiently using the zero page addressing modes.

14) Using the wrong base address in indexing. As we have noted in Laboratories 6 and 7, you often have to subtract 1 from the actual starting address to process an array efficiently. It is easy to be off by one location and end up being either misaligned or beyond the boundaries of an array or table.

You will undoubtedly make and discover errors that we have not mentioned, but this list should at least suggest some possibilities. Unfortunately, debugging computer programs is more of an art than a science.

KEY POINT SUMMARY

1) The writing of software, like the building of hardware, consists of many stages. Writing the actual computer instructions (or *coding*) is one of the easiest stages.

2) Flowcharting is a simple graphic technique for designing and documenting programs. A set of standard flowchart symbols is in widespread use.

3) A flowchart is a good starting point for writing a program, but it should not become a burden all by itself.

4) Breakpoints are stopping places in programs that you can use to determine whether sections are correct or to pass through sections that you know are correct. You can set a breakpoint in a 6502 machine language program by replacing an operation code with a BRK instruction.

COMMON PROGRAMMING ERRORS

You should watch for the following common errors in 6502 machine language programs:

1) Confusing data and addresses. Remember the difference between immediate and direct addressing; immediate addressing means that the data follows the operation code "immediately" while direct addressing means that the address of the data follows the operation code. Remember that the value stored in a memory location that is addressed directly or through indexing is not related to the effective address, base address, or index.

2) Inverting the order of the bytes in 2-byte addresses. Remember that the 6502 expects the less significant byte first.

3) Copying operation codes incorrectly. You should check programs before executing them.

4) Using the CARRY incorrectly. Remember that comparisons such as CMP, CPX, or CPY (as well as subtraction) set the CARRY if no borrow is required. The CARRY flag is an inverted borrow, not a true borrow as on most other microprocessors. Note also that addition and subtraction instructions (ADC and SBC) always include the CARRY. You must explicitly clear the CARRY before addition or set it before subtraction if you do not want its value to affect the result.

5) Inverting the logic of conditional branch instructions (e.g., using BCC instead of BCS or BNE instead of BEQ). Be particularly careful after a comparison instruction (CMP, CPX, or CPY).

6) Jumping to the wrong address. This often results in repeating or omitting initialization instructions or instructions that update indexes or indirect addresses.

7) Calculating relative offsets incorrectly. You should always check results you have obtained by hand. If you perform many such calculations, you should use a hexadecimal calculator.

8) Omitting addresses, offsets, or data. Watch for instructions such as JMP, which requires a full 16-bit address in the next 2 bytes of memory. Remember that absolute addressing modes always require 2 bytes of memory for their addresses and zero page modes (including preindexing and postindexing) always require 1 byte.

Some of these errors (e.g., 2, 3, and 7) will not occur if you use an assembler.

Other common errors are:

9) Failing to initialize counters, indexes, and indirect addresses.

5) To resume a program after a breakpoint, you must remember that BRK increments the program counter by 2. If you want to resume your program as if the breakpoint had not occurred, you may have to adjust the program counter and other registers. You must make sure that the computer does not omit an instruction accidentally or execute a location that does not contain an operation code.

6) You can use the single-step (actually single-instruction) mode to pinpoint an error, usually after you have used breakpoints to localize it.

7) Common programming errors include confusing data and addresses, inverting logic or reversing the direction of operations, failing to initialize variables or save results, omitting operands, forgetting how instructions affect flags, ignoring trivial cases, and branching incorrectly.

Laboratory 9

Arithmetic

PURPOSE

To learn to perform arithmetic calculations using the 6502 microprocessor.

PARTS REQUIRED

None.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, pp. 81-99, 146-156.

J. F. Hart et al., *Computer Approximations,* Wiley, New York, 1978.

K. Hwang, *Computer Arithmetic,* Wiley, New York, 1979.

U. W. Kulisch and W. L. Mirankar, *Computer Arithmetic in Theory and Practice,* Academic Press, New York, 1980.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 198-210.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 4-13 through 4-15, Chapter 8.

Y. L. Luke, *Mathematical Functions and Their Approximations,* Academic Press, New York, 1975.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 114-139.

H. Schmid, *Decimal Computation,* Wiley, New York, 1974.

D. Stevenson, "A Proposed Standard for Binary Floating-Point Arithmetic (IEEE Task P754)," *Computer,* March 1981, pp. 51-62.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 12-14 (BCD code), 17-24 (binary arithmetic), 24-26 (BCD arithmetic), 174-180 (arithmetic/logic unit), 180-187 (comparison of microprocessors), 322-324 (status control instructions), 324-329 (arithmetic instructions), 375-377 (multibyte arithmetic operations).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapters 2, 6, 7, 8, 12, 13, and 14.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 6-20 (arithmetic unit), 92-95 (applications of indexes).

WHAT YOU SHOULD LEARN

1) The standard BCD representation.
2) How to choose between the binary and BCD representations.
3) How to add and subtract 8-bit binary numbers.
4) How to use the 6502's decimal mode.
5) How to add and subtract 16-bit numbers.
6) How to round binary and decimal numbers.
7) How to perform multiple-precision binary and decimal addition and subtraction.
8) How to use lookup tables to perform arithmetic.

TERMS

BCD (binary-coded-decimal)—a representation of decimal numbers in which each decimal digit is coded separately into a binary number.

Carry—a bit which is 1 if an addition overflows into the succeeding digit position.

Half (or auxiliary) carry—a flag used in 8-bit computers to indicate whether there was a carry from the less significant 4 bits or less significant digit.

Interpolation—estimating values of a function at points between those at which the values are known already.

Linearization—the mathematical approximation of a function by a straight line between two points at which its values are known.

Pseudo-operation (or **pseudo-op** or **pseudo-instruction**)—an assembly language operation code that directs the assembler to perform some action but does not result in the generation of a machine language instruction.

Rounding—approximating a number by the closest whole number.

Standard (or **8, 4, 2, 1**) **BCD**—a BCD representation in which the bit positions have the same weights as in ordinary binary numbers.

Truncation—dropping the less significant part of a number.

6502 INSTRUCTIONS

ADC—add with carry; add the contents of the specified memory location and the CARRY flag to the accumulator. The result is placed in the accumulator.

CLC—clear carry; set the CARRY flag to zero.

CLD—clear decimal mode flag; set the DECIMAL MODE flag to zero. The processor will perform subsequent addition (ADC) and subtraction (SBC) instructions in the binary mode.

SBC—subtract with carry; subtract the contents of the specified memory location and the complemented (inverted) contents of the CARRY flag from the accumulator. The result is (A) = (A) – (M) – (1 – CARRY), where M is the specified memory address.

SEC—set carry; set the CARRY flag to one.

SED—set decimal mode flag; set the DECIMAL MODE flag to one. The processor will perform subsequent addition (ADC) and subtraction (SBC) instructions in the decimal (standard BCD) mode.

6502 ASSEMBLER PSEUDO-OPERATIONS

.BYTE—form byte-length data; place the specified byte-length data (8 bits per item) in the next available memory locations. The items in the list of data should be separated by commas. This pseudo-operation loads memory with fixed data (such as tables, messages, and numerical constants) that is necessary for the proper execution of the program.

.DBYTE—form double-byte-length data with more significant byte first; place the specified 16-bit data in the next available memory

locations. The items in the list of data should be separated by commas. This pseudo-operation loads memory with 16-bit fixed data items that are stored in the opposite order from that normally used with the 6502.

.WORD—form double-byte-length data with less significant byte first; place the specified 16-bit data in the next available memory locations. The items in the list should be separated by commas. This pseudo-operation loads memory with 16-bit fixed data or addresses in the order normally used with the 6502.

*=—set origin; assign the object code generated from the subsequent assembly language statements to memory addresses starting with the one specified. This pseudo-operation lets the assembly language programmer assign a starting memory address for the main program and assign data or other sections of the program to different areas of memory as required.

APPLICATIONS OF ARITHMETIC

The processing of data almost always involves arithmetic. Typical operations are the averaging of data readings, scaling, linearization of inputs, calculation of numerical integrals and derivatives, determination of frequency responses, statistical analysis, and preparation of plots. Simple applications require only binary or decimal addition and subtraction. Decimal arithmetic is necessary in calculators, business equipment, terminals, instruments, appliances, and games.

In this laboratory, we will reexamine earlier programs that performed 8-bit binary arithmetic. We will also discuss decimal addition and subtraction, 16-bit arithmetic, rounding, multibyte binary and decimal addition and subtraction, and the use of lookup tables.

AN 8-BIT SUM

The following program adds two 8-bit unsigned binary numbers from memory locations 0340 and 0341 and stores the sum in memory location 0040, ignoring any carry that might be generated.

```
LDA     $0340     ;GET FIRST NUMBER
CLC               ;CLEAR CARRY
ADC     $0341     ;ADD SECOND NUMBER
STA     $40       ;SAVE SUM
BRK
```

Note that we must clear the CARRY before adding, since ADC always includes it in the sum. The hexadecimal version of this program is Program 9-1; we have inserted two NOPs in anticipation of a decimal version.

PROGRAM 9-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	EA	NOP	
0201	AD	LDA	$0340
0202	40		
0203	03		
0204	18	CLC	
0205	6D	ADC	$0341
0206	41		
0207	03		
0208	85	STA	$40
0209	40		
020A	EA	NOP	
020B	00	BRK	

Enter Program 9-1 and run it for the following cases:

1) (0340) = 32
   (0341) = 25
   Result: (0040) = 57
2) (0340) = 38
   (0341) = 25
   Result: (0040) = 5D

PROBLEM 9-1

Extend Program 9-1 so that it sets memory location 0041 to 0 if there is no carry from the addition and to 1 if there is. Try the following cases:

1) (0340) = 38
   (0341) = 25
   Result: (0040) = 5D
   (0041) = 00
2) (0340) = 98
   (0341) = 89
   Result: (0040) = 21
   (0041) = 01

PROBLEM 9-2

Revise Program 9-1 so that it performs binary subtraction instead of binary addition. How do you keep the CARRY from affecting the result? Run the program for the following cases:

1) (0340) = 32
   (0341) = 25
   Result: (0040) = 0D
2) (0340) = 32
   (0341) = 58
   Result: (0040) = DA

PROBLEM 9-3

What is the value of CARRY at the end of each sample run of the binary subtraction program? Is CARRY equal to the actual carry from the two's-complement addition? Remember, the microprocessor subtracts by adding the two's complement of the subtrahend. Most microprocessors (e.g., the 8080, 8085, Z-80, and 6800) invert the carry from a subtraction to form a true borrow. Draw a circuit that produces a true borrow from the carry output of the arithmetic-logic unit and a status line that differentiates between addition and subtraction instructions. Assume that the status line (SUBTRACT/$\overline{\text{ADD}}$) is 0 if the processor is adding and 1 if it is subtracting.

BINARY-CODED-DECIMAL (BCD) REPRESENTATION

A BCD code is the simplest way to represent decimal numbers in a computer. No multiplications or divisions by 10 are necessary since each decimal digit is coded separately into 4 bits. In the standard (or 8, 4, 2, 1) BCD code (see Table 9-1), the numbers 0 through 9 are the same as in binary. However, numbers above 9 are different because of the separate coding of the decimal digits (see Table 9-2 for some examples). Note the following features of BCD as compared to binary:

1) Each decimal digit is coded separately in BCD. This is not the case in binary, since 10 is not an integral power of 2. Computers will surely correct this situation when they take over the world—people will be required to have either 8 or 16 fingers.

2) The BCD representation is always greater than or equal to the binary representation of the same number.

3) The BCD representation requires more memory than the binary representation. For example, 8 bits can represent a binary number as large as 255 but only 99 in BCD. The number 999 requires three BCD digits (12 bits) but only 10 bits in binary (since $2^{10} = 1024$).

Table 9-1

STANDARD BCD REPRESENTATION

DECIMAL DIGIT	BCD REPRESENTATION
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 9-2

STANDARD BCD REPRESENTATIONS OF SOME DECIMAL NUMBERS

DECIMAL NUMBER	BCD REPRESENTATION	BINARY REPRESENTATION
10	0001 0000	00001010
11	0001 0001	00001011
12	0001 0010	00001100
13	0001 0011	00001101
16	0001 0110	00010000
25	0010 0101	00011001
50	0101 0000	00110010
66	0110 0110	01000010
83	1000 0011	01010011

4) Some binary numbers are not valid BCD numbers. In the standard BCD code, no digit can be larger than 9.

One problem with BCD numbers is that they are difficult to process in binary arithmetic units. This is because the BCD representation of 10 (00010000) is not one larger than the BCD representation of 9 (00001001) —it is, in fact, seven larger. (Try subtracting!) Thus, to produce a BCD sum using a binary adder, you must add an extra factor of six whenever the sum of two digits is 10 or more.

Example 1:

$$\begin{array}{rcl} 33\text{ (BCD)} & = & 00110011 \\ +\,25\text{ (BCD)} & = & \underline{00100101} \\ & & 01011000 = 58\text{ (BCD)} \end{array}$$

There is no problem here, since neither sum of digits is 10 or more.

Example 2:

$$\begin{array}{rcl} 38\text{ (BCD)} & = & 00111000 \\ +\,25\text{ (BCD)} & = & \underline{00100101} \\ & & 01011101 = 5D \end{array}$$

Here we need an extra factor of 6, since 8 + 5 produces a carry in ordinary decimal arithmetic.

$$\begin{array}{r} 5D \\ +\,\underline{06} \\ 63 \end{array}$$

Example 3:

$$\begin{array}{rcl} 98\text{ (BCD)} & = & 10011000 \\ +\,25\text{ (BCD)} & = & \underline{00100101} \\ & & 10111101 = BD \end{array}$$

Here we need extra factors of 6 for both digits.

$$\begin{array}{r} BD \\ +\,\underline{66} \\ 123 \end{array}$$

Obviously, deciding when to add 6 is no simple matter. You have to check each digit of the sum to see whether the result is 10 or more. This is particularly difficult to do if the computer is handling more than one digit at a time, since you must examine each digit separately. Because decimal arithmetic is essential in such common microprocessor applications as point-of-sale terminals and electronic games, most processors have instructions specifically aimed at implementing it. The 6502 microprocessor has a special decimal mode in which it performs all additions and subtractions in BCD. The processor enters this mode by executing an SED (SET DECIMAL MODE) instruction, thus setting the D (DECIMAL MODE) flag. The processor leaves the decimal mode by executing a CLD (CLEAR DECIMAL MODE) instruction. When the D flag is set, ADC and SBC instructions produce decimal results; increment and decrement in-

structions (DEC, DEX, DEY, INC, INX, INY), however, still produce binary results. You can determine which mode the processor is in by examining the D flag (bit 3 of the status register).

AN 8-BIT DECIMAL SUM

The following program adds two BCD numbers from memory locations 0340 and 0341 and stores the sum in memory location 0040, ignoring any carry that might be generated.

```
SED             ;ENTER DECIMAL MODE
LDA   $0340     ;GET FIRST NUMBER
CLC             ;CLEAR CARRY
ADC   $0341     ;ADD SECOND NUMBER
STA   $40       ;SAVE SUM
CLD             ;LEAVE DECIMAL MODE
BRK
```

The only changes from Program 9-1 are:

1) Memory location 0200 contains SED (F8) instead of NOP.
2) Memory location 020A contains CLD (D8) instead of NOP.

The 6502 performs both decimal addition and decimal subtraction directly; most other microprocessors require the programmer to implement decimal subtraction as the addition of a negative number.

Run the BCD version of Program 9-1 with the following sample data:

1) (0340) = 32
   (0341) = 25
   Result: (0040) = 57
2) (0340) = 38
   (0341) = 25
   Result: (0040) = 63

Note the difference in the second result from the binary version.

PROBLEM 9-4

Determine the contents of the accumulator and the CARRY flag and the value of the half-carry (i.e., the carry from bit 3) after the processor executes ADC $0341 for the following examples in the binary mode:

a) (0340) = 38
(0341) = 25

b) (0340) = 98
(0341) = 25

c) (0340) = 98
(0341) = 89

d) (0340) = 90
(0341) = 91

Why does the processor need the half-carry when it is operating in the decimal mode? **Hint**: Examine the results of examples c and d.
You can observe the contents of the accumulator in memory location 00F3 and the CARRY flag in bit 0 of memory location 00F1, but you will have to calculate the half-carry since the processor does not make it available externally.

PROBLEM 9-5

Add a continuation to the decimal addition program that shows the least significant digit of the sum on the rightmost seven-segment display (display #6—see Tables 5-1 and 5-2). Using the examples from Problem 9-4, the rightmost display should show 3 for example a (least significant digit of 63), 3 for example b (least significant digit of 123), 7 for example c (least significant digit of 187), and 1 for example d (least significant digit of 181). Use the KIM seven-segment code table that starts in memory location 1FE7.

PROBLEM 9-6

Make the program perform decimal subtraction instead of decimal addition. Try the examples in Problem 9-4. What is the value of CARRY at the end of each example? What does CARRY mean at the end of this program? Remember, the CARRY ends up in bit 0 of memory location 00F1.

Using the decimal mode to perform BCD addition and subtraction is straightforward. The decimal mode does, however, cause some confusion in interpreting the flags and in determining the effects of instructions other than ADC and SBC. You should remember the following facts:

1) Increment and decrement instructions produce binary results even when the processor is in the decimal mode.

2) The NEGATIVE (SIGN) flag is meaningless after ADC or SBC instructions executed in the decimal mode. The NEGATIVE flag reflects only the binary result, not the BCD result. Thus, for example, subtracting 60 (hex) from 30 (hex) in the decimal mode sets the NEGA-

TIVE flag (since the binary result is 11010000 or D0 hex), even though the decimal result is 01110000 binary or 70 hex, which has a most significant bit of 0.

3) Comparison instructions executed in the decimal mode set the ZERO and CARRY flags properly (the mode does not matter—why?). Comparisons set the NEGATIVE flag according to the binary result as we just discussed for ADC and SBC.

A further problem with the DECIMAL MODE flag is that it may have unexpected effects on programs that do not need decimal arithmetic. If a program includes ADC or SBC instructions, its results will depend on the value of the D flag. If the program does not clear the D flag explicitly, it will produce incorrect results if it is accidentally executed with the D flag set. If, on the other hand, the program clears the D flag, it may cause errors in other routines that expect the D flag to be set.

A complicating factor is that resetting the 6502 has no effect on the D flag. Thus that flag could be either 0 or 1 initially, and the programmer must remember to establish its value (usually with CLD) before the program executes any ADC or SBC instructions. An early version of the KIM monitor failed sporadically because it did not initialize the D flag. We will discuss the handling of the D flag in more detail in Laboratory A.

As you might expect, the processor does not indicate an error if your program uses the decimal mode improperly. If, for example, your program adds or subtracts numbers that are not decimal (such as E4 hex or DA hex) in the decimal mode, the processor will simply execute the instructions and produce meaningless results.

Be careful that you do not inadvertently set the D flag or forget to clear it. We have found this to be a problem in changing or debugging programs and in cases where we wanted to initialize the status register by loading memory location 00F1. If a program that involves ADC or SBC instructions is producing odd results for no apparent reason, you should check the value of the D flag.

DECIMAL SUMMATION

We have already shown in Laboratory 6 how to perform a binary summation (see Program 6-1). The following program adds an array of unsigned binary numbers starting in memory location 0340; it places the sum in memory location 0040, ignoring any carries. The length of the array is assumed to be in memory location 0041. Program 9-2 is the hexadecimal version.

```
NOP
LDX   $41        ;INDEX = ARRAY LENGTH
LDA   #0         ;SUM = ZERO INITIALLY
```

```
ADDELM    CLC                ;CLEAR CARRY
          ADC   $033F,X      ;ADD AN ELEMENT TO THE SUM
          DEX
          BNE   ADDELM
          STA   $40          ;SAVE SUM
          NOP
          BRK
```

PROGRAM 9-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	EA		NOP	
0201	A6		LDX	$41
0202	41			
0203	A9		LDA	#0
0204	00			
0205	18	ADDELM	CLC	
0206	7D		ADC	$033F,X
0207	3F			
0208	03			
0209	CA		DEX	
020A	D0		BNE	ADDELM
020B	F9			
020C	85		STA	$40
020D	40			
020E	EA		NOP	
020F	00		BRK	

We have included NOPs again to make a decimal version easy to implement. Enter Program 9-2 and run it with the following sample data:

(0041) = 03 (number of elements)

(0340) = 35 (elements)

(0341) = 47

(0342) = 28

Result:

(0040) = A4

Change the program so the summation is decimal rather than binary. Run the revised program with the same sample data. The answer now should be

(0040) = 10

16-BIT ARITHMETIC

We can extend Program 9-2 to handle 16-bit numbers. 16-bit addition, however, involves more than just two 8-bit additions. Now we have the problem of carries from the less significant byte to the more significant byte.

Here the ADC (ADD WITH CARRY) instruction becomes really useful, since it produces the result

$$(A) = (A) + (M) + (CARRY)$$

where A is the accumulator and M is the addressed memory location. So all that we must do to perform 16-bit addition is:

1) Clear the CARRY to start.
2) Add the less significant bytes.
3) Add the more significant bytes.

When adding the more significant bytes, ADC automatically includes the carry from the less significant bytes. In fact, we would have to clear the carry to exclude it. Note that there is never a carry into the less significant bytes.

The following program adds an array of 16-bit numbers starting in memory location 0340 and places the sum in memory locations 0040 and 0041. Each number is stored in 2 bytes, with the less significant byte at the lower address. The length of the array (how many 16-bit numbers there are) is in memory location 0042.

```
        NOP             ;NOP FOR DECIMAL VERSION
        LDY   $42       ;COUNT = ARRAY LENGTH
        LDA   #0        ;SUM = ZERO INITIALLY
        STA   $40
        STA   $41
        TAX             ;INDEX = ZERO INITIALLY
ADDELM  LDA   $40       ;ADD IN LSB OF ELEMENT
        CLC
        ADC   $0340,X
        STA   $40
        INX
        LDA   $41       ;ADD IN MSB OF ELEMENT
        ADC   $0340,X
        STA   $41
        INX
        DEY             ;COUNT ELEMENTS
        BNE   ADDELM
        NOP             ;NOP FOR DECIMAL VERSION
        BRK
```

PROGRAM 9-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	EA		NOP	
0201	A4		LDY	$42
0202	42			
0203	A9		LDA	#0
0204	00			
0205	85		STA	$40
0206	40			
0207	85		STA	$41
0208	41			
0209	AA		TAX	
020A	A5	ADDELM	LDA	$40
020B	40			
020C	18		CLC	
020D	7D		ADC	$0340,X
020E	40			
020F	03			
0210	85		STA	$40
0211	40			
0212	E8		INX	
0213	A5		LDA	$41
0214	41			
0215	7D		ADC	$0340,X
0216	40			
0217	03			
0218	85		STA	$41
0219	41			
021A	E8		INX	
021B	88		DEY	
021C	D0		BNE	ADDELM
021D	EC			
021E	EA		NOP	
021F	00		BRK	

Program 9-3 is the hexadecimal version; enter and run it with the following sample data:

(0042) = 02 (number of 16-bit elements)

(0340) = 3E (LSBs of first element)

(0341) = 47 (MSBs of first element)

(0342) = F5 (LSBs of second element)

(0343) = 2A (MSBs of second element)

Result:

(0040) = 33 (LSBs of sum)

(0041) = 72 (MSBs of sum)

that is,

$$\begin{array}{r} 473E \\ +\ 2AF5 \\ \hline 7233 \end{array}$$

PROBLEM 9-7

Revise Program 9-3 so that it works backward through the array rather than forward. Which version is shorter and faster?

PROBLEM 9-8

Revise Program 9-3 so that it obtains the base address from memory locations 0043 and 0044. That is, for the example given, we would have to load

(0043) = 40 (LSBs of base address)

(0044) = 03 (MSBs of base address)

PROBLEM 9-9

Revise Program 9-3 so that the 16-bit elements and the sum are stored with their more significant bytes first (at the lower address). Some computers (particularly those based on Motorola 6800 or 6809 microprocessors) use this format. Remember to rearrange the data before executing the revised program. This format makes working backward through the array simpler. Why?

PROBLEM 9-10

Make Program 9-3 perform decimal (BCD) addition rather than binary addition. Try the decimal program on the following sample data:

(0042) = 02 (number of four-digit elements)

(0340) = 36 (LSDs of first element)

(0341) = 21 (MSDs of first element)

(0342) = 97 (LSDs of second element)

(0343) = 18 (MSDs of second element)

Result:

(0040) = 33 (LSDs of sum)

(0041) = 40 (MSDs of sum)

that is,

```
  2136
+ 1897
------
  4033
```

PROBLEM 9-11

Extend the answer to Problem 9-10 so that it concludes with the carries saved in memory location 0042. Use memory location 0043 for the number of elements. Try the following sample data:

(0043) = 02 (number of four-digit elements)

(0340) = 36 (LSDs of first element)

(0341) = 21 (MSDs of first element)

(0342) = 97 (LSDs of second element)

(0343) = 98 (MSDs of second element)

Result:

(0040) = 33 (LSDs of sum)

(0041) = 20 (middle digits of sum)

(0042) = 01 (MSDs of sum = carries)

that is,

```
  2136
+ 9897
------
 12033
```

The carries should also be a decimal number (i.e., you cannot just increment the register or memory location in which you are keeping them). Remember that DEC, DEX, DEY, INC, INX, and INY produce binary results even when the DECIMAL MODE flag is set.

ROUNDING

Rounding binary numbers is simple because the digits can only have the values 0 or 1. So all that you must do is examine the most significant bit of the part of the number that you plan to drop. The procedure is:

1) If MSB = 1, round up by adding 1 to the remaining bits.
2) If MSB = 0, leave the remaining bits unchanged.

The following program will round a 16-bit number in memory locations 0041 and 0042 (MSBs in 0041) to an 8-bit number in memory location 0040.

```
        LDA   $42      ;TRANSFER MSB'S
        STA   $40
        LDA   $41      ;MUST WE ROUND MSB'S UP?
        BPL   DONE
        INC   $40      ;YES, ADD 1 TO MSB'S
DONE    BRK
```

Program 9-4 is the hexadecimal version of the binary rounding program. Try it for the following cases:

1) (0041) = 61
   (0042) = 69
   Result: (0040) = 69
2) (0041) = D1
   (0042) = 69
   Result: (0040) = 6A

PROGRAM 9-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A5		LDA	$42
0201	42			
0202	85		STA	$40
0203	40			
0204	A5		LDA	$41
0205	41			
0206	10		BPL	DONE
0207	02			
0208	E6		INC	$40
0209	40			
020A	00	DONE	BRK	

Decimal rounding is more difficult because:

1) You must determine if the most significant digit to be truncated is 5 or more. If so, the remaining digits must be rounded up.
2) All additions must be decimal. You cannot use INC to add 1. (Why not?) Instead, you must use a sequence such as

```
SED             ;ENTER DECIMAL MODE
CLC             ;ADD 1 IN DECIMAL
ADC     #1
CLD             ;LEAVE DECIMAL MODE
```

PROBLEM 9-12

Write a program that rounds a four-digit BCD number in memory locations 0041 and 0042 (most significant digits in 0042) to a two-digit BCD number in memory location 0040. Try the program for the following cases:

1) (0041) = 61
   (0042) = 69
   Result: (0040) = 70
2) (0041) = 28
   (0042) = 69
   Result: (0040) = 69

PROBLEM 9-13

Write a program that rounds a 24-bit binary number in memory locations 0040, 0041, and 0042 (most significant bits in 0042) to a 16-bit binary number in memory locations 0041 and 0042 (MSBs in 0042). Try the program for the following test cases:

1) (0040) = 75
   (0041) = 6F
   (0042) = 93
   Result: (0041) = 6F
   (0042) = 93
2) (0040) = D5
   (0041) = 6F
   (0042) = 93
   Result: (0041) = 70
   (0042) = 93

3) (0040) = D5
(0041) = FF
(0042) = 93
Result: (0041) = 00
(0042) = 94

Remember that INC affects the ZERO flag but not the CARRY flag.

MULTIPLE-PRECISION ARITHMETIC

We can extend the previous programs to handle numbers of any bit length. The procedure for adding binary numbers of arbitrary length is (see Figure 9-1) as follows:

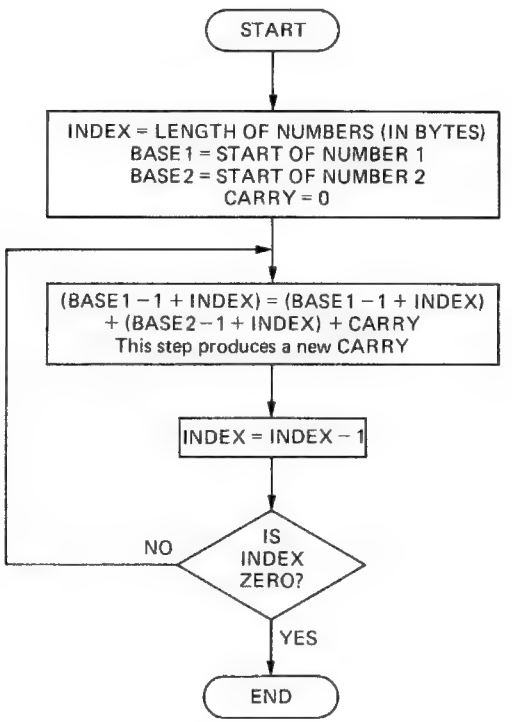


FIGURE 9-1. Flowchart for multiple-precision arithmetic program.

1) Initialization.

INDEX = LENGTH OF NUMBERS (IN BYTES)

CARRY = 0, since there is never a carry into the least significant bytes

2) Add 8 bits.

(BASE1 − 1 + INDEX) = (BASE1 − 1 + INDEX) +
(BASE2 − 1 + INDEX) + CARRY

This step produces a new CARRY.

3) Update index.

INDEX = INDEX − 1

If INDEX ≠ 0, return to 2.

Here we have stored the numbers in the inverse of our normal order: that is, with their least significant bytes at the highest address. We could store the sum in a third array simply by indexing from a third base address.

If the length of the numbers is in memory location 0040, the numbers start (most significant bytes first) in memory location 0340 and 0360, and the answer replaces the number starting in memory location 0340, the required program is

```
        LDX   $40          ;INDEX = LENGTH OF NUMBERS
        CLC                ;CLEAR CARRY TO START
ADBYTE  LDA   $033F,X      ;GET BYTE OF FIRST NUMBER
        ADC   $035F,X      ;ADD BYTE OF SECOND NUMBER
        STA   $033F,X      ;STORE RESULT AS FIRST NUMBER
        DEX
        BNE   ADBYTE
        BRK
```

Program 9-5 is the hexadecimal version. A key factor here is that DEX does not affect the CARRY flag; the carry from one 8-bit addition is therefore available to be included in the next 8-bit addition. If DEX changed the CARRY flag, we would have to save its value after each addition and restore it before the next addition.

PROGRAM 9-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A6		LDX	$40
0201	40			
0202	18		CLC	
0203	BD	ADBYTE	LDA	$033F,X
0204	3F			
0205	03			
0206	7D		ADC	$035F,X
0207	5F			
0208	03			
0209	9D		STA	$033F,X
020A	3F			
020B	03			
020C	CA		DEX	
020D	D0		BNE	ADBYTE
020E	F4			
020F	00		BRK	

Try Program 9-5 on the following 48-bit problem:

(0040) = 06 (length of numbers in bytes)

(0340) = 29 (MSBs of first number)
(0341) = 3E
(0342) = AB
(0343) = F0
(0344) = 59
(0345) = C7 (LSBs of first number)

(0360) = 19 (MSBs of second number)
(0361) = D0
(0362) = 28
(0363) = A1
(0364) = 93
(0365) = EA (LSBs of second number)

Result:

(0340) = 43 (MSBs of sum)

(0341) = 0E

(0342) = D4

(0343) = 91

(0344) = ED

(0345) = B1 (LSBs of sum)

that is,

$$\begin{array}{r} 293EABF059C7 \\ +\ 19D028A193EA \\ \hline 430ED491EDB1 \end{array}$$

PROBLEM 9-14

Change Program 9-5 so that it obtains the base addresses for the two arrays from memory locations 0030 and 0031 (first number and sum) and 0032 and 0033 (second number). That is, the new program should produce the same result as Program 9-5 if the initial conditions are

(0030) = 3F (LSBs of base address minus 1 for first number)

(0031) = 03 (MSBs of base address minus 1 for first number)

(0032) = 5F (LSBs of base address minus 1 for second number)

(0033) = 03 (MSBs of base address minus 1 for second number)

Note that we have subtracted 1 from both base addresses.

PROBLEM 9-15

Write a program that adds decimal numbers of arbitrary length. Assume the same conditions as in Program 9-5. Try the program on the following 12-digit sample case:

(0040) = 06 (length of numbers in bytes)

(0340) = 29 (most significant digits of first number)

(0341) = 34

(0342) = 71

(0343) = 60

(0344) = 59

(0345) = 87 (least significant digits of first number)

(0360) = 19 (most significant digits of second number)

(0361) = 60

(0362) = 28

(0363) = 81

(0364) = 93

(0365) = 15 (least significant digits of second number)

Result:

(0340) = 48 (most significant digits of sum)

(0341) = 95

(0342) = 00

(0343) = 42

(0344) = 53

(0345) = 02 (least significant digits of sum)

that is,

$$\begin{array}{r} 293471605987 \\ +\ 196028819315 \\ \hline 489500425302 \end{array}$$

PROBLEM 9-16

Write a program that subtracts decimal numbers of arbitrary length. Assume the same conditions as in Program 9-5. The number starting in memory location 0360 is to be subtracted from the number starting in memory location 0340. Try the program on the following 12-digit sample case. Be careful of the fact that the CARRY, as usual, acts as an inverted borrow. That is, CARRY = 1 if no borrow is generated from the subtraction, and CARRY = 0 if a borrow is generated.

(0040) = 06 (length of numbers in bytes)

(0340) = 29 (most significant digits of minuend)

(0341) = 34

(0342) = 71
(0343) = 60
(0344) = 59
(0345) = 87 (least significant digits of minuend)

(0360) = 19 (most significant digits of subtrahend)
(0361) = 60
(0362) = 28
(0363) = 81
(0364) = 93
(0365) = 15 (least significant digits of subtrahend)

Result:

(0340) = 09 (most significant digits of difference)
(0341) = 74
(0342) = 42
(0343) = 78
(0344) = 66
(0345) = 72 (least significant digits of difference)

that is,

$$\begin{array}{r} 293471605987 \\ -\ 196028819315 \\ \hline 097442786672 \end{array}$$

ARITHMETIC WITH LOOKUP TABLES

More complex arithmetic can often be performed by using lookup tables. Such tables simply contain all the possible results organized in a convenient manner. Now all that you have to do is locate the desired result, just as we did in the seven-segment code conversion presented earlier.

For example, we could form a table from the squares of the decimal digits. The following program uses the table to find the square of the digit in memory location 0041 and places the result in memory location 0042. The lookup procedure is the same one that we used in Programs 5-3 (seven-segment code conversion) and 7-4 (accessing a specific element of

an array). A major advantage of the tabular approach is that we can use the same lookup program in many different applications, thus saving time, effort, and money.

```
LDX     $41             ;GET DATA
LDA     $0340,X         ;GET SQUARE OF DATA FROM TABLE
STA     $42
BRK

*=$0340                 ;SQUARES OF DECIMAL DIGITS
.BYTE   0,1,4,9,16,25,36,49,64,81
```

* = (Origin or Set Origin) is a directive to the assembler (often called a *pseudo-operation* or *pseudo-op*) which indicates that the object code generated from subsequent statements is to be placed in memory beginning at the specified address. .BYTE (Form Byte-Length Data) is another pseudo-operation which indicates that the specified 8-bit data is to be placed in the next available memory locations. Program 9-6 is the hexadecimal version.

PROGRAM 9-6

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A6	LDX	$41
0201	41		
0202	BD	LDA	$0340,X
0203	40		
0204	03		
0205	85	STA	$42
0206	42		
0207	00	BRK	
0340	00	.BYTE	0
0341	01		1
0342	04		4
0343	09		9
0344	10		16
0345	19		25
0346	24		36
0347	31		49
0348	40		64
0349	51		81

Run Program 9-6 with the following sample data:

1) (0041) = 04
   Result: (0042) = 10
2) (0041) = 07
   Result: (0042) = 31

PROBLEM 9-17

Write a program that uses the square table of Program 9-6 to add the squares of the contents of memory locations 0040 and 0041. The program should place the sum in memory location 0042.

Example:

(0040) = 03

(0041) = 06

Result:

(0042) = 2D (hex), since 2D = 09 (3^2) + 24 (6^2) in hexadecimal.

PROBLEM 9-18

Write a program that uses a table to calculate the BCD square of a single BCD digit in memory location 0041 and places the result in memory location 0042.

Examples:

1) (0041) = 06
   Result: (0042) = 36
2) (0041) = 09
   Result: (0042) = 81

The answers are BCD numbers.

PROBLEM 9-19

Write a program that uses a table to calculate the cube of a BCD digit. Allow 2 bytes for each entry in the table, since 7^3, 8^3, and 9^3 are all larger than 256. Assume that the data is in memory location 0041 and place the result in memory locations 0042 and 0043 (MSBs in 0043).

Examples:

1) (0041) = 03
   Result: (0042) = 1B
   (0043) = 00

2) (0041) = 07
Result: (0042) = 57
(0043) = 01

Remember that the results are hexadecimal numbers.

PROBLEM 9-20

Write a program that converts a decimal digit in memory location 0041 into a four-digit square root in memory locations 0042 and 0043 (most significant digits in 0043). Use the following square-root table; place it in memory starting at address 0340 and indicate its placement in your assembly language program with a .WORD (Form Double-Byte Length Data) pseudo-operation.

VALUE	SQUARE ROOT
0	00.00
1	01.00
2	01.41
3	01.73
4	02.00
5	02.24
6	02.45
7	02.65
8	02.83
9	03.00

Examples:

1) (0041) = 03
Result: (0042) = 73
(0043) = 01

2) (0041) = 07
Result: (0042) = 65
(0043) = 02

PROBLEM 9-21

Extend the answer to Problem 9-20 so that it converts the decimal digit in memory location 0041 into a six-digit square root in memory locations 0042, 0043, and 0044 (most significant digits in 0044). Use the following square root table:

VALUE	SQUARE ROOT
0	00.0000
1	01.0000
2	01.4142
3	01.7321
4	02.0000
5	02.2361
6	02.4495
7	02.6458
8	02.8284
9	03.0000

Examples:

1) (0041) = 02
   Result: (0042) = 42
   (0043) = 41
   (0044) = 01

2) (0041) = 06
   Result: (0042) = 95
   (0043) = 44
   (0044) = 02

If the table is very long, you can save memory by storing only some of the entries and interpolating to obtain intermediate values. This method is discussed in T.A. Seim, "Numerical Interpolation for Microprocessor-Based Systems," *Computer Design*, February 1978, pp. 111-116.

KEY POINT SUMMARY

1) The BCD representation is a convenient way to handle decimal numbers, since each decimal digit is coded separately. This representation does, however, require more memory and processing instructions than the ordinary binary representation.

2) Most microprocessors have special instructions for performing decimal arithmetic. The 6502 microprocessor has a decimal mode in which addition and subtraction (ADC and SBC) instructions produce decimal (BCD) results automatically. The processor enters the decimal mode by executing an SED (SET DECIMAL MODE) instruction and returns to the binary mode by executing a CLD (CLEAR DECIMAL

MODE) instruction. The programmer must initialize the DECIMAL MODE (D) flag and keep track of its value. Increment and decrement instructions always produce binary results, regardless of the state of the D flag.

3) Multiple-precision arithmetic requires a succession of 8-bit operations. Extra instructions are needed to update indexes and to move data to and from the 8-bit accumulator.

4) The CARRY flag transfers information (carries or borrows) between 8-bit operations; ADD WITH CARRY and SUBTRACT WITH CARRY instructions handle the carries or borrows appropriately.

5) Rounding simply requires an examination of the most significant digit that is to be dropped. Rounding may be a multiple-precision operation involving carries.

6) Using lookup tables is a simple way to perform complex arithmetic at the cost of extra memory. The lookup procedure depends only on the organization of the table and the length of the elements; it does not depend on the data values or the function involved.

Laboratory A

Subroutines and the Stack

PURPOSE

To learn how to write and use subroutines.

REFERENCE MATERIALS

R. C. Camp et al., *Microcomputer System Principles Featuring the 6502/KIM*, Matrix Publishers, Portland, OR, 1978, Appendix B (particularly pp. 508-511).

M. L. DeJong, *Programming and Interfacing the 6502*, Howard W. Sams, Indianapolis, IN, 1980, pp. 156-157, 172-183.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 57-60, 97-100, 113-115, 120, 220-229.

L. A. Leventhal, *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 9-16 through 9-17, Chapters 10 and 15.

L. J. Scanlon, *6502 Software Design*, Howard W. Sams, Indianapolis, IN, 1980, pp. 35-36, 75-91, 111-113.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.*, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 172-174 (stack pointer), 342-345 (stack transfers), 355-360 (subroutines), 367-368 (indirect addressing), 368-375 (timing loops).

W. J. Weller, *Practical Microcomputer Programming: The 6502*, Northern Technology Books, Evanston, IL, 1980, Chapter 9, Appendixes A, B, and C.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 40-42, Appendix I (KIM-1 monitor listing).

WHAT YOU SHOULD LEARN

1) Why subroutines are useful.
2) How to transfer control to and from subroutines using the JUMP TO SUBROUTINE and RETURN FROM SUBROUTINE instructions.
3) How to call subroutines from other subroutines.
4) The principles of stack management.
5) How to use the stack for temporary data storage.
6) How to use a delay subroutine.
7) How to use I/O routines as subroutines.
8) How to use the monitor subroutines.
9) How to handle the DECIMAL MODE flag in subroutines.
10) How to transfer control to one of a set of subroutines using either indirect jumps or the RTS instruction.

TERMS

Hardware stack—a stack that the computer manages automatically when executing instructions that use it.

Indirect jump—a jump instruction that transfers control to the address stored in a register or memory location, rather than to a fixed address.

Information-hiding principle—a principle of program development whereby each part of a program conceals from all other parts information about its implementation that is not essential to their functions. Following this principle minimizes the interactions between parts of a program and thus makes changes easier to introduce (since they usually affect only one part of the program).

Library program—a program that is part of a collection of programs and is written and documented according to a standard format.

LIFO (last-in, first-out) memory—a memory that is organized according to the order in which elements are entered and from

which elements can be retrieved only in the order opposite from that in which they were entered.

Module—a part or section of a program.

Nesting—constructing programs, subroutines, or interrupt service routines so that one level is contained within another and so on. The *nesting level* is the number of transfers of control required to reach a particular routine without returning to a higher level.

Nibble—a unit of 4 bits. A byte (8 bits) may be described as consisting of a high nibble (4 most significant bits) and a low nibble (4 least significant bits).

Overflow (of a stack)—exceeding the amount of memory allocated to a stack.

Parameter—an item that must be provided to a subroutine or program in order for it to be executed.

Passing parameters—making the required parameters available to a subroutine.

Pop (or **pull**)—to remove an operand from a stack.

Push—to store an operand in a stack.

Reentrant—a program that can be executed correctly even while the same program is being interrupted or preempted.

Software stack—a stack that is managed by means of specific instructions, as opposed to a hardware stack which the computer manages automatically.

Stack—a section of memory that can be accessed only in a last-in, first-out manner. That is, data can be added to or removed from the stack only through its top; new data is placed above the old data and the removal of a data item makes the item below it the new top.

Stack pointer—a register that contains the address of the top of a stack.

Subroutine—a subprogram that can be executed (called) from more than one place in a main program.

Subroutine call—the process whereby a computer transfers control from its current program to a subroutine, while retaining the information required to resume the current program.

Subroutine linkage—the mechanism whereby a computer retains the information required to resume its current program after it completes the execution of a subroutine.

Underflow (of a stack)—attempting to remove more data from a stack than has been entered into it.

6502 INSTRUCTIONS

JSR—jump to subroutine; jump to the specified absolute (direct) address and save the return address (the address of the last byte of the JSR instruction) in the stack.

PHA—store accumulator in stack; store the accumulator at the top of the stack and decrement the stack pointer by 1.

PHP—store status register in stack; store the status register at the top of the stack and decrement the stack pointer by 1.

PLA—load accumulator from stack; increment the stack pointer by 1 and load the accumulator from the top of the stack.

PLP—load status register from stack; increment the stack pointer by 1 and load the status register from the top of the stack.

RTS—return from subroutine; load the program counter from the top two locations in the stack and then add 1 to it. This is equivalent to a jump to the address one larger than the contents of the top two locations in the stack.

TSX—transfer stack pointer to index register X; transfer the contents of the stack pointer to index register X. The stack pointer is not affected. This is the only way to save the contents of the stack pointer.

TXS—transfer index register X to stack pointer; transfer the contents of index register X to the stack pointer. Index register X is not affected. This is the only way to load the stack pointer.

Note: In the instructions JSR, PHA, PHP, PLA, PLP, and RTS, the top of the stack is the address on page 1 given by the contents of the stack pointer. That is, the top of the stack is at address 01ss, where ss is the contents of the 6502's 8-bit stack pointer.

RATIONALE AND TERMINOLOGY

Most of the tasks we have described so far occur repeatedly in real applications. Actual programs often, for example, must perform many time delays, code conversions, and arithmetic functions. Clearly, we would be wasting a large amount of memory (and programming time) if we repeated the same sequences of instructions every time they were needed. What we would like is to employ a single copy of each common sequence.

In order to do this, we need a way for the processor to suspend its current program, transfer control to one of the common sequences of instructions, execute that sequence, and finally resume the current pro-

gram where it left off. Then the processor could use the same copy of a common sequence of instructions at many different points in its overall program. The common sequence of instructions could even be part of the monitor or operating system; in that case, the programmer would not even have to code it or load it into memory.

We use the following terminology in describing such common sequences of instructions:

- The sequence is referred to as a *subroutine*, since it is subordinate to the main or calling program.
- The process of transferring control to the subroutine is referred to as a *subroutine call.*
- A piece of data or an address that a subroutine needs to perform its tasks is called a *parameter.*
- The process of providing parameters to the subroutine is referred to as *passing parameters.*
- The method whereby the computer transfers control to the subroutine and back to the calling program is called a *subroutine linkage.*

6502 CALL AND RETURN INSTRUCTIONS

The 6502 microprocessor has two instructions (JSR and RTS) that are essential for implementing subroutines; these instructions work as follows:

- JUMP TO SUBROUTINE (JSR) places the specified address (only absolute addressing may be used) in the program counter and saves the old program counter (the address of the last byte of the JUMP TO SUBROUTINE instruction) in the stack.
- RETURN FROM SUBROUTINE (RTS) loads the program counter from the top two locations in the stack and adds 1 to it.

So a JSR (JUMP TO SUBROUTINE) instruction in the calling program transfers control to the subroutine that starts at the specified address. JSR allows only absolute (direct) addressing. An RTS (RETURN FROM SUBROUTINE) instruction at the end of the subroutine causes the CPU to resume the calling program where it left off. The subroutine linkage is thus in the stack—the JSR instruction saves the return address there and the RTS instruction retrieves it.

THE RAM STACK

To understand how JSR and RTS work, we must discuss how the 6502 transfers data to and from its stack. The processor performs the transfers as follows:

1) It places data in the stack by first storing the data at the top of the stack and then subtracting 1 from the stack pointer (Figure A-1).
2) It removes data from the stack by first adding 1 to the stack pointer and then loading the data from the top of the stack (Figure A-2).

The CPU increments or decrements the stack pointer automatically when executing instructions that use the stack. We could produce the same effects with an index register (resulting in what is called a *software stack*), but the increment or decrement instructions would require extra time and memory.

Note that the 6502's stack is always on page 1 of memory. The top of the stack (the next empty location) is at address 01ss hex, where ss is the contents of the stack pointer.

Note the following features of the stack:

1) The stack is just an ordinary area of read/write memory. The processor moves the top of the stack up or down by decrementing or incrementing the contents of the stack pointer (examine Figures A-1 and A-2 carefully). There is nothing special about the memory that contributes to its use as a stack.

2) The programmer (or the monitor program) assigns an area on page 1 to the stack by placing an initial value in the 8-bit stack pointer. The sequence LDX (load index register X), TXS (transfer index register X to stack pointer) can be used for this purpose. TXS is the only instruction that loads the stack pointer. The KIM monitor starts its stack at address 01FF. We will start our stack at address 017F so that it will not interfere with the monitor or with our programs. Programs usually do not change the stack pointer explicitly after it has been initialized.

3) The stack grows down in memory (i.e., from higher addresses to lower addresses). If you feel uneasy about this, just try standing on your head and everything will be all right.

4) The stack pointer always contains the next available (empty) stack address. That is, the lowest address actually occupied by the stack is one larger than the address to which the stack pointer refers.

5) The instructions JSR and RTS transfer 16-bit addresses to and from the stack. The 8 least significant bits are obtained first and

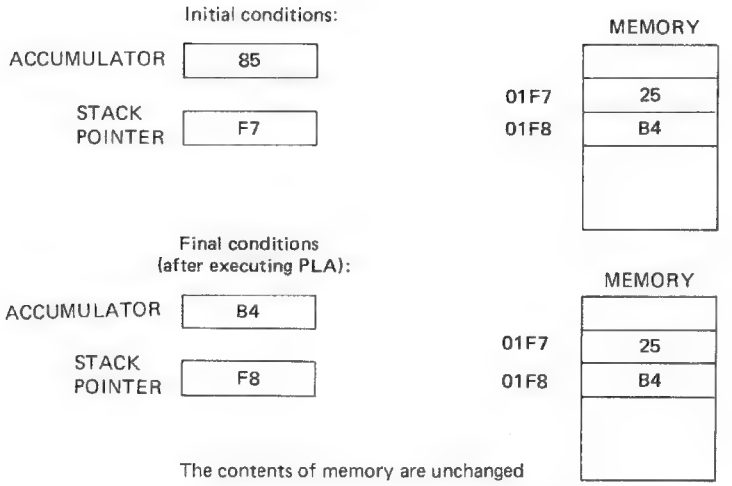


FIGURE A-1. Entering data into the stack.

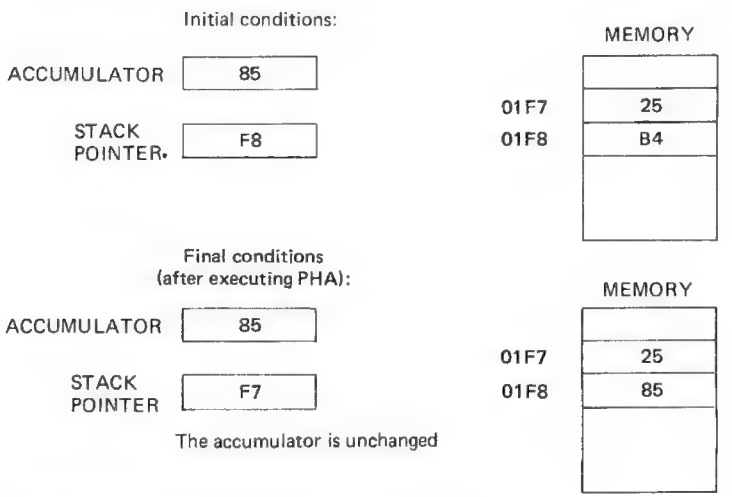


FIGURE A-2. Removing data from the stack.

stored last in accordance with the usual 6502 method for storing addresses. Be careful—the least significant bits are stored last but the stack is growing down, so they end up at the lower address. Remember the curious offset of 1; JSR saves the address of its own last byte in the stack and RTS adds 1 to the address it obtains from the stack. This procedure evidently

allows the 6502 to execute JSR faster, but it is also another quirk for the programmer to remember.

Examples:

a) (S) = 61

(PC) = 021C

After the processor executes JSR $0238 (occupying addresses 021C through 021E),

(S) = (S) – 2 = 5F, since a 2-byte address has been saved in the stack.

(PC) = 0238, the starting address of the subroutine.

(0160) = 1E, the LSBs of the address of the last byte of JSR $0238.

(0161) = 02, the MSBs of the address of the last byte of JSR $0238.

b) (S) = 7C

(017D) = 28

(017E) = 02

After the processor executes RTS,

(S) = 7E, since a 2-byte address has been removed from the stack.

(PC) = (017E) (017D) + 1 = 0228 + 1 = 0229

6) The instructions PLA (load accumulator from stack) and PLP (load status register from stack) load 8 bits of data from the top of the stack into the specified register. The instructions PHA (store accumulator in stack) and PHP (store status register in stack) store 8 bits of data from the specified register at the top of the stack. There is no way to transfer data directly between the stack and an index register; the data must move through the accumulator.

Examples:

a) (S) = 67

(A) = F2

After the processor executes PHA,

(S) = 66

(0167) = F2

The accumulator does not change.

b) (S) = 6F

(0170) = 3B

After the processor executes PLA,

(S) = 70

(A) = 3B

Memory location 0170 does not change, but it is no longer part of the stack. The stack expands and contracts like the tides which alternatively cover and uncover parts of the shoreline.

GUIDELINES FOR STACK MANAGEMENT

Most beginners find the stack confusing and even a little frightening. However, you will have no problems with the stack if you follow these guidelines:

1) Load the stack pointer during system initialization. Start the stack at the highest available address on page 1.

2) Always balance stack operations. Each JUMP TO SUBROUTINE should be balanced by a RETURN FROM SUBROUTINE and each push (PHA or PHP) by a pull (PLA or PLP). This is just like balancing left and right parentheses in arithmetic or in sentences.

3) Don't be fancy. Leave the stack and the stack pointer alone except for JSR, RTS, push, and pull instructions. Simple programs rarely need more than 20 RAM locations for the stack. Leave yourself lots of room so that the stack never overflows; many 6502 programmers simply leave all of page 1 for the stack.

SUBROUTINE LINKAGES IN THE STACK

Let us see how JSR and RTS work in a simple situation. Enter the following program into memory:

```
STARTING AT $0200
        LDX     #$7F        ;INITIALIZE USER STACK POINTER
        TXS
        JSR     $0260       ;GO TO SUBROUTINE
        BRK
STARTING AT $0260
        TSX                 ;SAVE STACK POINTER
        STX     $40
        BRK
```

Program A-1 is the hexadecimal version. Note that we need the sequence TSX, STX to save the current stack pointer in memory.

PROGRAM A-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	20	JSR	$0260
0204	60		
0205	02		
0206	00	BRK	
0260	BA	TSX	
0261	86	STX	$40
0262	40		
0263	00	BRK	

PROBLEM A-1

What are the final values of the stack pointer and memory locations 017E and 017F? What is the final value of memory location 0040? Explain why memory locations 017E and 017F do not contain the address of the next executable instruction after JSR.

Note that BRK is itself a subroutine call. It adds 2 to the program counter, stores the incremented program counter and the status register in the stack, and gets the new program counter from two fixed memory locations (1FFE and 1FFF in the KIM monitor). Those memory locations, in turn, contain the address 1C1F (verify this!); the program starting there directs the processor to the address in memory locations 17FE and 17FF.

Be careful not to press the RS key after running Program A-1. What happens to the stack pointer in memory location 00F2 if you do? Do memory locations 017E and 017F change value?

PROBLEM A-2

What are the final values of the stack pointer and memory locations 017E and 017F if you replace the BRK (00) in memory location 0263 with RTS (60)? Explain what has happened.

PROBLEM A-3

What are the final values of the stack pointer and memory locations 017C through 017F if you place the following instructions in memory? Remember to execute the main program starting at memory location 0200.

```
0260    BA      TSX
0261    86      STX     $40
0262    40
0263    20      JSR     $0280
0264    80
0265    02
0266    60      RTS

0280    BA      TSX
0281    86      STX     $41
0282    41
0283    00      BRK
```

What are the values of the stack pointer in memory locations 0040 and 0041? What happens if you revise the program as follows?

```
0260    BA      TSX
0261    86      STX     $40
0262    40
0263    20      JSR     $0280
0264    80
0265    02
0266    BA      TSX
0267    86      STX     $42
0268    42
0269    00      BRK

0280    BA      TSX
0281    86      STX     $41
0282    41
0283    60      RTS
```

Why is it essential that the stack be organized in a last-in, first-out manner? A subroutine that is called by another subroutine is said to be *nested* within that subroutine.

SAVING REGISTERS IN THE STACK

You can use the stack to save registers before calling a subroutine. Now you need not worry about which registers the subroutine uses. Remember the following:

1) You can save the accumulator with the instruction PHA. You can restore its old value with the instruction PLA.

2) You can save the status register (Figure 2-3) with the instruction PHP. You can restore its old value with the instruction PLP.

In fact, PLP is the only instruction that loads the status register and PHP is the only instruction that saves its value. One of the few situations in which loading or saving the status register is necessary is when an operating system or monitor resumes or suspends a user program. For example, the KIM monitor, as we have noted previously, saves the status register in memory location 00F1 if your program ends with a BRK instruction. It also loads the status register from memory location 00F1 when you press GO.

The easiest way to load the status register with a specific value is by transferring the value through the stack. The sequence

```
LDA     #VALUE      ;LOAD STATUS THROUGH STACK
PHA
PLP
```

will do the job. Remember, there are no instructions that transfer data directly to or from the status register.

3) You can save an index register in the stack only by first transferring its contents to the accumulator. The required sequences of instructions are:

```
TXA         ;SAVE INDEX REGISTER X IN STACK
PHA
```

and

```
TYA         ;SAVE INDEX REGISTER Y IN STACK
PHA
```

You can restore the index registers with the inverse sequences; that is,

```
PLA         ;RESTORE INDEX REGISTER X FROM STACK
TAX
```

and

```
PLA         ;RESTORE INDEX REGISTER Y FROM STACK
TAY
```

Since these sequences use the accumulator, you must save the accumulator before saving the index registers and restore the accumulator after restoring the index registers.

4) You must restore register values in the opposite order from that in which they were saved. If they are saved in the order

```
PHP         ;SAVE STATUS
PHA         ;SAVE ACCUMULATOR
TXA         ;SAVE INDEX REGISTER X
PHA
TYA         ;SAVE INDEX REGISTER Y
PHA
```

they must be restored in the order

```
PLA         ;RESTORE INDEX REGISTER Y
TAY
PLA         ;RESTORE INDEX REGISTER X
TAX
PLA         ;RESTORE ACCUMULATOR
PLP         ;RESTORE STATUS
```

For example, the following subroutine performs a table lookup using the index in the accumulator and the base address in memory locations 0040 and 0041.

```
TAY
LDA         ($40),Y
RTS
```

Here both the index and the base address are variables.

The next program obtains a seven-segment code from the table in the monitor (starting address 1FE7). You must place the data in memory location 0044. We have loaded the status register and index register Y from memory initially to make their starting values easy to observe and change. Program A-2 contains the hexadecimal versions of the main program and the subroutine.

```
LDX    #$7F      ;INITIALIZE USER STACK POINTER
TXS
LDA    $42       ;INITIALIZE STATUS REGISTER
PHA              ;LOAD STATUS THROUGH STACK
PLP
LDY    $43       ;INITIALIZE INDEX REGISTER Y
LDA    #$E7      ;GET BASE ADDRESS OF TABLE
STA    $40
LDA    #$1F
STA    $41
LDA    $44       ;GET DATA
JSR    $0260     ;PERFORM A TABLE LOOKUP
STA    $45       ;SAVE RESULT
BRK
```

We have used the following memory locations for temporary storage:

ADDRESS	CONTENTS
0040	LSBs of base address of table
0041	MSBs of base address of table
0042	Initial value of status register
0043	Initial value of index register Y
0044	Data to be converted
0045	Result (element obtained from table)
00F1	Final value of status register
00F4	Final value of index register Y

The KIM monitor provides the last two locations automatically.

PROGRAM A-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
Main Program			
0200	A2	LDX	#$7F
0201	7F		

PROGRAM A-2 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0202	9A	TXS	
0203	A5	LDA	$42
0204	42		
0205	48	PHA	
0206	28	PLP	
0207	A4	LDY	$43
0208	43		
0209	A9	LDA	#$E7
020A	E7		
020B	85	STA	$40
020C	40		
020D	A9	LDA	#$1F
020E	1F		
020F	85	STA	$41
0210	41		
0211	A5	LDA	$44
0212	44		
0213	20	JSR	$0260
0214	60		
0215	02		
0216	85	STA	$45
0217	45		
0218	00	BRK	
Subroutine			
0260	A8	TAY	
0261	B1	LDA	($40),Y
0262	40		
0263	60	RTS	

Run Program A-2 for the following test cases:

1) (0044) = 03
   Result: (0045) = CF
2) (0044) = 0D
   Result: (0045) = DE

Run each test case four times with the following conditions:

1) (0042) = 7D (NEGATIVE flag = ZERO flag = 0)
   (0043) = 00

2) (0042) = FF (NEGATIVE flag = ZERO flag = 1)
   (0043) = 00
3) (0042) = 7D (NEGATIVE flag = ZERO flag = 0)
   (0043) = FF
4) (0042) = FF (NEGATIVE flag = ZERO flag = 1)
   (0043) = FF

Note that the only flags we have varied are the NEGATIVE flag (bit 7 of the status register) and the ZERO flag (bit 1 of the status register). The other flags are either irrelevant or unaffected by Program A-2, since it contains no arithmetic or shift instructions. We have set all the other flags to 1 arbitrarily in all the conditions.

Do the initial values that we place in the NEGATIVE and ZERO flags and in index register Y affect their final values? JSR and RTS do not affect the flags or index register Y, but the subroutine does. Ignoring the effects of subroutines is a common source of programming errors; a subroutine call may result in the execution of many instructions and may change the values of registers, flags, and memory locations.

We can easily make the main program save the initial value of the status register in the stack. The revised main program is

```
LDX   #$7F      ;INITIALIZE USER STACK POINTER
TXS
LDA   $42       ;INITIALIZE STATUS REGISTER
PHA             ;LOAD STATUS THROUGH STACK
PLP
PHP             ;SAVE STATUS REGISTER IN STACK
LDY   $43       ;INITIALIZE INDEX REGISTER Y
LDA   #$E7      ;GET BASE ADDRESS OF TABLE
STA   $40
LDA   #$1F
STA   $41
LDA   $44       ;GET DATA
JSR   $0260     ;PERFORM A TABLE LOOKUP
STA   $45       ;SAVE RESULT
PLP             ;RESTORE STATUS REGISTER FROM STACK
RTS
```

Program A-3 is the hexadecimal version of the revised main program.

PROGRAM A-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)
0200	A2	LDX #$7F
0201	7F	

PROGRAM A-3 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0202	9A	TXS	
0203	A5	LDA	$42
0204	42		
0205	48	PHA	
0206	28	PLP	
0207	08	PHP	
0208	A4	LDY	$43
0209	43		
020A	A9	LDA	#$E7
020B	E7		
020C	85	STA	$40
020D	40		
020E	A9	LDA	#$1F
020F	1F		
0210	85	STA	$41
0211	41		
0212	A5	LDA	$44
0213	44		
0214	20	JSR	$0260
0215	60		
0216	02		
0217	85	STA	$45
0218	45		
0219	28	PLP	
021A	00	BRK	

Repeat the various test cases with the revised program and show that it preserves the NEGATIVE and ZERO flags.

PROBLEM A-4

Change the main program so that it saves and restores index register Y (save index register Y after saving the status register). Determine the final values of the stack pointer and memory locations 017C through 017F if you replace the RTS instruction at the end of the subroutine (memory location 0263) with BRK and set the initial conditions to:

(0042) = 04 (initial value of status register)

(0043) = 23 (initial value of index register Y)

Saving incidental registers in the stack limits the flow of information between the main program and the subroutine. The way in which the sub-

routine uses those registers will not affect the operation of the main program. The programmer need not understand the details of the subroutine and need not change the main program if the subroutine is revised or replaced.

A DELAY SUBROUTINE

The following subroutine from Program 4-3 provides a 1-ms delay:

```
DLYMS   LDX   #$C8      ;DELAY 1 MS
DLY     DEX
        BNE   DLY
        RTS
```

We can use it in a main program as follows:

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
        JSR   DLYMS     ;DELAY 1 MS
        BRK
```

The hexadecimal version of the main program and subroutine are given in Program A-4. Enter and run this program.

PROGRAM A-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	20		JSR	DLYMS
0204	60			
0205	02			
0206	00		BRK	
0260	A2	DLYMS	LDX	#$C8
0261	C8			
0262	CA	DLY	DEX	
0263	D0		BNE	DLY
0264	FD			
0265	60		RTS	

PROBLEM A-5

How could you make the subroutine preserve the original value of the status register? How much do the additional instructions increase the execution time?

PROBLEM A-6

How could you make the main program wait for the number of milliseconds in memory location 0040?

Example:

(0040) = 07 results in a delay of 7 ms.

How could you make the subroutine wait for the number of milliseconds in index register Y? What are the advantages and disadvantages of this approach as compared to modifying the main program?

PROBLEM A-7

Revise the subroutine so that the delay is in seconds rather than in milliseconds. Have the subroutine preserve the original values of the index registers, status register, and accumulator. Use memory location 0041 for the count in seconds and use the 1-s delay program from Problem 8-10.

Example:

(0041) = 05 results in a delay of 5 s.

AN INPUT SUBROUTINE

The following subroutine identifies which of the eight switches attached to a port has been closed. The data from the port is assumed to be in the accumulator.

```
IDSW    LDY   #$FF     ;SWITCH NUMBER = -1
SRCHS   INY            ;INCREMENT SWITCH NUMBER
        LSR   A        ;IS NEXT SWITCH CLOSED?
        BCS   SRCHS    ;NO, KEEP LOOKING
        RTS
```

Program A-5 is the hexadecimal version. We have used memory locations starting with 0270 to avoid interfering with the 1-ms delay routine in Program A-4. The following program uses Program A-5 to wait until a switch is closed at port A of the user 6530 device and then identify it:

PROGRAM A-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0270	A0	IDSW	LDY	#$FF
0271	FF			
0272	C8	SRCHS	INY	
0273	4A		LSR	A
0274	B0		BCS	SRCHS
0275	FC			
0276	60		RTS	

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
WAITC   LDA   $1700     ;GET DATA FROM SWITCHES
        CMP   #$FF      ;ARE ANY SWITCHES CLOSED?
        BEQ   WAITC     ;NO, WAIT
        JSR   IDSW      ;YES, IDENTIFY CLOSED SWITCH
        STY   $40       ;SAVE SWITCH NUMBER
        BRK
```

The hexadecimal version of the main program is given in Program A-6. Enter and run this program. Where does the subroutine place the switch number?

PROGRAM A-6

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	AD	WAITC	LDA	$1700
0204	00			
0205	17			
0206	C9		CMP	#$FF
0207	FF			
0208	F0		BEQ	WAITC
0209	F9			
020A	20		JSR	IDSW
020B	70			
020C	02			
020D	84		STY	$40
020E	40			
020F	00		BRK	

PROBLEM A-8

How would you make Program A-6 examine the switches once and conclude with either the switch number or FF (if no switches are closed) in memory location 0040? How would you use subroutine DLYMS (see Program A-4) to write a version that waits until two readings taken 1 ms apart give the same result? Note that DLYMS affects index register X but not index register Y.

PROBLEM A-9

Modify Program A-6 so that it waits for the number of separate switch closures specified in memory location 0042 and stores the identification numbers starting at memory location 0340. Use subroutine DLYMS to provide a 1-ms delay for debouncing.

Example:

If (0042) = 03 and you close switches 0, 6, and 5 in that order, the results should be

(0340) = 00

(0341) = 06

(0342) = 05

Assume that you must open all switches between closures.

AN OUTPUT SUBROUTINE

The next subroutine converts a decimal digit in index register X to a seven-segment code, and shows it on the rightmost display. Program A-7 is the hexadecimal version. The subroutine blanks the display if register X does not contain a decimal digit.

```
DSP1    LDA    #$FF      ;MAKE PORT A OUTPUT
        STA    $1741
        LDA    #0        ;GET BLANK CODE
        CPX    #10       ;IS DATA A DECIMAL DIGIT?
        BCS    DONE      ;NO, NO CONVERSION NECESSARY
        LDA    SSEG,X    ;YES, CONVERT TO SEVEN-SEGMENT CODE
DONE    LDX    #$12      ;ACTIVATE RIGHTMOST DISPLAY (#6)
        STX    $1742
        STA    $1740     ;PLACE RESULT ON DISPLAY
        RTS
SSEG    .BYTE  $3F, $06, $5B, $4F, $66
        .BYTE  $6D, $7D, $07, $7F, $6F
```

PROGRAM A-7

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0260	A9	DSP1	LDA	#$FF
0261	FF			
0262	8D		STA	$1741
0263	41			
0264	17			
0265	A9		LDA	#0
0266	00			
0267	E0		CPX	#10
0268	0A			
0269	B0		BCS	DONE
026A	03			
026B	BD		LDA	SSEG,X
026C	77			
026D	02			
026E	A2	DONE	LDX	#$12
026F	12			
0270	8E		STX	$1742
0271	42			
0272	17			
0273	8D		STA	$1740
0274	40			
0275	17			
0276	60		RTS	
0277	3F	SSEG	.BYTE	$3F,
0278	06			$06,
0279	5B			$5B,
027A	4F			$4F,
027B	66			$66,
027C	6D			$6D,
027D	7D			$7D,
027E	07			$07,
027F	7F			$7F,
0280	6F			$6F

PROBLEM A-10

Write a main program that uses Program A-7 and the 1-s delay routine (Problem 8-10) to show the contents of memory location 0041 on the rightmost seven-segment display (#6) for 1 s.

PROBLEM A-11

Modify Program A-7 so that it uses the seven-segment code table in the monitor. The modified program should accept any hexadecimal digit as valid data.

PROBLEM A-12

Make Program A-7 use index register Y to determine the display that it activates. The main program should provide the display number (1 to 6) as a parameter.

Examples:

(Y) = 01 activates the leftmost display.
(Y) = 06 activates the rightmost display.

PROBLEM A-13

Make the display subroutine from Problem A-12 return immediately with (A) = FF if Y originally contains an invalid display number (something other than 1 through 6).

Example:

(Y) = 7C causes an immediate return with (A) = FF.

Note how Program A-7 differs from the subroutine in Program A-2. In Program A-2, both the table address and the element number are parameters. In Program A-7 only the element number is a parameter and the table is part of the program. The choice of parameters is very important because it determines the flexibility of the subroutine and how it is used. Note the tradeoffs we can make. The earlier subroutine requires two parameters and an external table but can handle any function that requires 1 byte of data and produces a 1-byte result. Program A-7 requires only a single parameter but can perform only a specific table lookup.

USING THE MONITOR SUBROUTINES

The JSR instruction also lets us use subroutines that are part of a monitor or operating system. For example, the KIM monitor includes subroutines that handle input and output, perform code conversions, and generate time delays. Using these subroutines can save a large amount of time and effort.

Remember that we have already utilized a code conversion table in the monitor ROM—now we shall investigate the use of entire routines. An easy one to employ is the time-delay routine DELAY, which starts in address 1ED4. This routine counts the contents of memory locations

17F2 and 17F3 (MSBs in 17F3) down to a negative value and then returns control to the calling program. We can use it as follows:

```
LDX     #$7F     ;INITIALIZE USER STACK POINTER
TXS
LDA     #CNTH    ;GET COUNT FOR DELAY (MSB'S)
STA     $17F3
LDA     #CNTL    ;AND LSB'S
STA     $17F2
JSR     DELAY
BRK
```

The hexadecimal version of this program is given in Program A-8.

PROGRAM A-8

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#CNTH
0204	CNTH		
0205	8D	STA	$17F3
0206	F3		
0207	17		
0208	A9	LDA	#CNTL
0209	CNTL		
020A	8D	STA	$17F2
020B	F2		
020C	17		
020D	20	JSR	DELAY
020E	D4		
020F	1E		
0210	00	BRK	

Set CNTL (memory location 0209) to 00 and try the following sequence of values for CNTH (memory location 0204): 80, 40, 20, 10, 08, 04, 02, 01. When can you no longer see the delay? You may want to use the LEDs attached to user 6530 port B to make the length of the delay more obvious. Make the port an output and turn all the LEDs on (with "0" bits) before calling DELAY and off (with "1" bits) afterward.

One problem with using the monitor routines is that you cannot single-step through them. The KIM disables the single-step mode when it is executing instructions in its monitor in order to avoid interference

with normal operating functions. So, if you execute a monitor subroutine in the single-step mode, the KIM will complete the entire subroutine and stop after the first instruction it executes from the user memory.

Even though JSR does not affect the user registers or the flags, the subroutine may affect them. In general, you must preserve any values that you need by saving them in the stack before calling the subroutine. Note the importance of knowing which registers a subroutine affects.

PROBLEM A-14

Determine which of the following registers and flags subroutine DELAY affects by experimenting with their values in the single-step mode.

1) The accumulator (memory location 00F3)
2) Index register Y (memory location 00F4)
3) Index register X (memory location 00F5)
4) NEGATIVE flag (bit 7 of memory location 00F1)
5) ZERO flag (bit 1 of memory location 00F1)
6) CARRY flag (bit 0 of memory location 00F1)

A subroutine may change memory locations as well as registers and flags. Place the value 55 in memory location 17F4 before executing Program A-8. What happens? Try some other initial values.

PROBLEM A-15

Use subroutine DELAY to write a program that waits for you to close and then open a switch attached to bit 0 of port A of the user 6530 device. A value of about 0050 in memory locations 17F2 and 17F3 should be sufficient to debounce most switches.

PROBLEM A-16

Use subroutine DELAY to write a program that flashes the center bar (segment g) of the leftmost display (display #1). Vary the parameter of DELAY until you can easily see the bar go on and off.

USING THE OUTPUT SUBROUTINES

Subroutine CONVD (starting address 1F48) shows the hexadecimal digit in the accumulator on the display selected by the contents of index register X (see Table 5-2) for about 0.5 ms and increments X by 2 (to prepare for activation of the next display to the right). So the following program shows a set of hexadecimal digits on the displays. The number of digits (1 to 6) is assumed to be in memory location 0040 and the starting

address of the display buffer (minus 1) is assumed to be in memory locations 0041 and 0042. We have allowed this routine to run indefinitely so you must reset the KIM to regain control.

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
        LDA   #$FF
        STA   $1741     ;MAKE PORT A OUTPUT
DSPLY   LDY   $40       ;GET NUMBER OF DIGITS
        LDX   #$08      ;START WITH LEFTMOST DISPLAY
DSP1    LDA   ($41),Y   ;GET DATA FOR NEXT DISPLAY
        JSR   CONVD     ;DISPLAY A DIGIT
        DEY             ;ALL DIGITS DISPLAYED?
        BNE   DSP1      ;NO, DISPLAY NEXT DIGIT
        BEQ   DSPLY     ;YES, START OVER AGAIN
```

Program A-9 is the hexadecimal version. Enter it into memory and run it with the following data:

(0040) = 06 (number of digits)
(0041) = 7F (LSBs of base address of buffer)
(0042) = 03 (MSBs of base address of buffer)
(0380) = 06 (rightmost display)
(0381) = 05
(0382) = 04
(0383) = 03
(0384) = 02
(0385) = 01 (leftmost display)

PROGRAM A-9

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#$FF
0204	FF			
0205	8D		STA	$1741
0206	41			
0207	17			
0208	A4	DSPLY	LDY	$40

PROGRAM A-9 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0209	40			
020A	A2		LDX	#$08
020B	08			
020C	B1	DSP1	LDA	($41),Y
020D	41			
020E	20		JSR	CONVD
020F	48			
0210	1F			
0211	88		DEY	
0212	D0		BNE	DSP1
0213	F8			
0214	F0		BEQ	DSPLY
0215	F2			

A final BRK is not necessary, since the program never returns control to the monitor.

PROBLEM A-17

What happens if you place an invalid hexadecimal digit in the display buffer? Try the values 10, 11, 38, 99, and DC. Change Program A-9 so that it displays all invalid hexadecimal digits as blanks.

Table A-1 lists some of the KIM monitor routines with their functions and entry points. Obviously, many of the routines are highly specialized to save memory space and are not generally useful. Table A-2 lists the keyboard codes produced by subroutine GETKEY.

Table A-1

KIM MONITOR SUBROUTINES

SUBROUTINE NAME	STARTING ADDRESS	DESCRIPTION
AK	1EFE	Scans the KIM keyboard to see if any keys are depressed. Returns with (A) = 0 if none is, (A) nonzero otherwise. Requires port A of the keyboard/display 6530 device to be programmed as an input.
CHK	1F91	Adds the contents of the accumulator to the 16-bit number stored in memory locations

Table A-1 (continued)

KIM MONITOR SUBROUTINES

SUBROUTINE NAME	STARTING ADDRESS	DESCRIPTION
		00F6 and 00F7. Used in calculating a checksum for paper tape records.
CONVD	1F48	1) Converts hexadecimal digit in A to a seven-segment code using KIM table. 2) Selects a KIM seven-segment display using the contents of index register X (see Table 5-2). 3) Drives the display for about 0.5 ms. 4) Increments X by 2 to prepare for driving the next display.
CRLF	1E2F	Prints a carriage return and a line feed on the teletypewriter.
DELAY	1ED4	Software delay that counts down from the 16-bit number in memory locations 17F3 (MSBs) and 17F2 (LSBs).
DEHALF	1EEB	Same as DELAY except that it starts the count at half the value in memory locations 17F3 and 17F2.
GETCH	1E5A	Reads one character from the teletypewriter and places it in A.
GETBYT	1F9D	Reads two hexadecimal characters from the teletypewriter, packs them into a single byte, and stores the byte in memory location 00F8. It also moves the previous contents of memory location 00F8 to 00F9 to allow the loading of 16-bit addresses as four hex digits.
GETKEY	1F6A	Scans the KIM keyboard and produces a result in the accumulator according to Table A-2. Requires port A of the keyboard/display 6530 device to be programmed as an input.
HEXTA	1E4C	Prints 4 least significant bits of A as a hexadecimal character on the teletypewriter.
INCPT	1F63	Increments the 16-bit value in memory locations 00FA (LSBs) and 00FB (MSBs).
INITS	1E88	Initializes the KIM, performing the following steps: 1) Puts keyboard in address mode. 2) Makes port A of the keyboard/display 6530 device into an input port. 3) Makes bits 0 through 5 of port B of the keyboard/display 6530 device into outputs and bit 7 into an input for use with teletypewriter. 4) Enables the teletypewriter (as opposed to the cassette tape unit) by clearing bit 5 of port

Table A-1 (continued)

KIM MONITOR SUBROUTINES

SUBROUTINE NAME	STARTING ADDRESS	DESCRIPTION
		B of the keyboard/display 6530 device and places the teletypewriter in its normal (logic 1) state by setting bit 0 of that same port. 5) Puts the 6502 processor in the binary mode by executing CLD.
INIT1	1E8C	Same as INITS except that it omits step 1 and thus does not affect the keyboard mode.
OPEN	1FCC	Moves the contents of memory location 00F8 to 00FA and 00F9 to 00FB.
OUTCH	1EA0	Prints the character in the accumulator on the teletypewriter.
OUTSP	1E9E	Prints a space on the teletypewriter.
PACK	1FAC	Converts the ASCII character in the accumulator into a hexadecimal digit. It then shifts that digit into the 4 least significant bit positions of the 16-bit number in memory locations 00F8 (LSBs) and 00F9 (MSBs), also shifting the previous number left 4 bits. This procedure allows the assembly of 8-bit data items or 16-bit addresses from hexadecimal digits. If the accumulator does not contain a valid digit, the routine returns with A unchanged.
PRTBYT	1E3B	Prints the contents of A as two hexadecimal digits on the teletypewriter.
PRTPNT	1E1E	Prints the contents of memory locations 00FA (LSBs) and 00FB (MSBs) as four hexadecimal digits on the teletypewriter.
PRTST	1E31	Prints on the teletypewriter a string of ASCII characters starting at 1FD5 + (X) and concluding at 1FD5. The string starting at 1FD5 consists of 6 nulls (00), line feed (0A), carriage return (0D), M, I, K, space (20), a control character (13), R, R, E, space (20), and another control character (13).
SCAND	1F19	Drives the KIM display, displaying the address in memory locations 00FA (LSBs) and 00FB (MSBs) and the contents of that address using CONVD. Exits to subroutine AK.
SCANDS	1F1F	Same as SCAND, except that it displays the contents of memory location 00F9 as the data instead of the contents of the address in 00FA and 00FB.

Table A-2

KIM KEYBOARD CODES (FROM SUBROUTINE GETKEY)

KEY PRESSED	CODE IN ACCUMULATOR (HEX)
0	00
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	08
9	09
A	0A
B	0B
C	0C
D	0D
E	0E
F	0F
AD	10
DA	11
+	12
GO	13
PC	14
NONE	15

SUBROUTINES AND THE DECIMAL MODE FLAG

One problem with writing general subroutines is that you may not know the initial state of the DECIMAL MODE flag. If the subroutine uses ADC or SBC instructions, you will want that flag to be in a known state. However, you do not want the subroutine to return control with D changed, since that could create errors in other programs. Thus the D flag can be a nuisance, since you often have to consider its value even in routines that do not perform decimal arithmetic.

The way to avoid problems with the DECIMAL MODE flag is to save its value (with PHP) at the start of any subroutine that uses ADC or SBC instructions. You must then determine the value of the flag for the subroutine (using either CLD or SED). Finally, at the end of the subroutine, you must restore the original value of the D flag (using PLP). The standard sequences are:

1) A subroutine that performs binary arithmetic.

```
PHP             ;SAVE OLD D FLAG
CLD             ;BE SURE OF BINARY MODE
.
.       main body of subroutine
.
PLP             ;RESTORE OLD D FLAG
```

2) A subroutine that performs decimal arithmetic.

```
PHP             ;SAVE OLD D FLAG
SED             ;SET DECIMAL MODE
.
.       main body of subroutine
.
PLP             ;RESTORE OLD D FLAG
```

This is not a tidy solution for the following reasons:

1) It adds extra code to any subroutine that uses ADC or SBC instructions, thus making the subroutine longer and slower.
2) It creates problems in debugging, documentation, and maintenance, since the extra instructions seem unnecessary. Although the DECIMAL MODE flag makes decimal arithmetic easy to implement, overall it causes more problems than it solves.

CALLING VARIABLE ADDRESSES

Since JSR allows only absolute addressing, the programmer needs additional techniques to handle cases involving a choice of subroutines. For example, many interactive systems ask the operator what he or she wishes to have done next (e.g., continue, start over, change parameters, report results, or stop). Many systems also have function keys that the operator uses to initiate common procedures such as mathematical or statistical functions, loading of programs or data, or graphics operations. In either case, the system does not know what subroutine it must execute until the operator makes a selection.

The way to get around the limitations of JSR is to transfer control to an intermediate routine that determines the ultimate destination. The intermediate routine can transfer control using either JMP (the only 6502 instruction that allows true indirect addressing) or RTS (which uses the stack). We will explore both approaches.

Let us assume that we have a table of starting addresses. For example, if our system is a calculator, these addresses might be the entry points for

the sine, cosine, exponential, logarithm, reciprocal, and other routines. If our system is a piece of test equipment, these addresses might be the entry points for the self-test, initial condition setting, or data analysis routines. To transfer control to a particular routine, all that we need to know is the starting address of the table and the entry number.

In the first approach, we use JMP with indirect addressing. In this case, JMP transfers control not to the specified address, but rather to the address stored starting at the specified address. So the intermediate routine must do the following:

1) Obtain the starting address of the routine from the table using indexed addressing. The only special consideration is that we must double the entry number, since addresses always occupy 2 bytes of memory.
2) Store the starting address in memory so that it can be used indirectly. We will store it in memory locations 0040 and 0041.
3) Jump indirectly. We indicate an indirect jump in assembly language by placing parentheses around the address; for example, JMP ($40) transfers control to the address stored in memory locations 0040 and 0041.

The following program (see Program A-10 for a hexadecimal version) will do the job. We have assumed that the entry number is in memory location 0042 and the table of addresses starts at memory location 0340.

```
JCALC   LDA   $42        ;GET ENTRY NUMBER
        ASL   A          ;DOUBLE IT FOR 2-BYTE ENTRIES
        TAX
        LDA   $0340,X    ;GET LSB OF ENTRY
        STA   $40
        INX              ;GET MSB OF ENTRY
        LDA   $0340,X
        STA   $41
        JMP   ($40)      ;TRANSFER CONTROL TO ENTRY
```

We do not need a final BRK instruction, since JMP ($40) is a transfer of control. However, to run the program, we will need a table of addresses and BRK instructions at each of those addresses for testing purposes. For example, if the table consists of four entries arranged as follows:

```
*=$0340
.WORD  $0260, $0280, $02A0, $02C0
```

then we must place BRK instructions at addresses 0260, 0280, 02A0, and 02C0.

Enter and run Program A-10 with the four-entry table above. Check that it works properly for (0042) = 00, 01, 02, and 03.

PROGRAM A-10

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0220	A5	JCALC	LDA	$42
0221	42			
0222	0A		ASL	A
0223	AA		TAX	
0224	BD		LDA	$0340,X
0225	40			
0226	03			
0227	85		STA	$40
0228	40			
0229	E8		INX	
022A	BD		LDA	$0340,X
022B	40			
022C	03			
022D	85		STA	$41
022E	41			
022F	6C		JMP	($40)
0230	40			
0231	00			
0260	00		BRK	
0280	00		BRK	
02A0	00		BRK	
02C0	00		BRK	
0340	60		.WORD	$0260,
0341	02			
0342	80			$0280,
0343	02			
0344	A0			$02A0,
0345	02			
0346	C0			$02C0
0347	02			

PROBLEM A-18

Write a main program that simply initializes the stack pointer to 7F and then calls Program A-10. What are the final contents of the stack pointer and memory locations 017E and 017F? Which registers and flags does Program A-10 affect? What happens if you replace the BRK instructions in memory locations 0260, 0280, 02A0, and 02C0 with RTS instructions (60 instead of 00)?

PROBLEM A-19

Revise Program A-10 so that it exits immediately if the value in memory location 0042 is invalid (i.e., larger than 3). Another approach to error handling is to conclude the table with an error exit and replace all invalid entries with the length of the table. Revise Program A-10 to use this approach.

Hint: For example, in the above-mentioned case, we would place an error exit in memory locations 0348 and 0349. The program would replace any entry above 4 with 4, thus causing a jump to the error exit.

PROBLEM A-20

Revise Program A-10 so that it obtains the base address of the table from memory locations 0043 and 0044. The results should be the same as before if

(0043) = 40

(0044) = 03

An alternative approach is to store the starting address of the subroutine in the stack and use RTS to transfer control to it. This seems strange, since we are using RTS to reach a subroutine, but it works properly. After all, RTS is a jump instruction—it jumps to the address contained in the top two locations in the stack. If the stack contains a starting address rather than a return address, RTS will jump to a subroutine rather than back to a calling program. Although this procedure is perfectly legitimate, it is confusing and the documentation should explain what the program is doing.

PROBLEM A-21

Revise Program A-10 so that it stores the entry from the table in the stack and uses RTS to transfer control. What changes must you make in the table of starting addresses? Remember that RTS always adds 1 to the address it retrieves from the stack before transferring control.

KEY POINT SUMMARY

1) You can make a single copy of a sequence of instructions available from anywhere in a program by making it into a subroutine. The process of transferring control to the subroutine is referred to as a subroutine call and the items the subroutine requires for proper execution are called parameters.

2) On the 6502 microprocessor, a JUMP TO SUBROUTINE (JSR) instruction in the calling program transfers control to a subroutine and saves the address of its own last byte in the stack. A RETURN FROM SUBROUTINE (RTS) instruction at the end of the subroutine restores control to the calling program by loading the program counter from the top of the stack and adding 1 to it.

3) The stack is just an ordinary area of read/write memory on page 1. The 8 least significant bits of the next empty address are in the processor's stack pointer. All that happens as the stack expands or contracts is that the contents of the stack pointer decrease or increase. Note that the stack grows downward (i.e., from higher addresses to lower addresses).

4) The programmer must initialize the stack pointer (using the LDX, TXS sequence) before calling any subroutines or using the stack for other purposes.

5) You can use the stack for temporary storage. This is convenient since the stack is ordered and easy to expand. The processor updates the stack pointer automatically as part of each instruction that transfers data to or from the stack.

6) You can use monitor subroutines just like ones you have written, but you must determine what parameters they require, which registers they use, and where they place their results. The monitor subroutines are not necessarily either general or useful.

7) Subroutines that use ADC or SBC instructions can preserve the DECIMAL MODE flag by executing PHP before establishing their own value for that flag and PLP before returning control.

8) A program can choose one of several subroutines to execute by calling a routine that calculates the proper starting address and stores it in memory. Either an indirect jump using JMP or an RTS instruction can then transfer control to the subroutine. An RTS instruction at the end of the subroutine will return control to the original calling point. JSR itself allows only absolute (direct) addressing.

Laboratory B

Input/Output Using Handshakes

PURPOSE

To learn how to perform input and output using handshake status and control signals.

PARTS REQUIRED

This optional addition will both expand the KIM's I/O section and allow experimentation with the 6520 Peripheral Interface Adapter (PIA):

- A 6520 PIA attached as shown in Figure B-1. We derived this interface from the article by Tenny (see the references in this laboratory).
- Eight switches attached to port A of the PIA as shown in Figure B-2.
- Eight LEDs attached to port B of the PIA as shown in Figure B-3.
- Four switches attached to the PIA control lines as shown in Figures B-4 and B-5.

- Two LEDs attached to the PIA control lines as shown in Figure B-6.

Note: Jumper wires may be used to choose between the alternative attachments to CA2 and CB2 shown in Figures B-5 and B-6.

Note: The 6520 Peripheral Interface Adapter (Device) is exactly the same as the 6820 Peripheral Interface Adapter designed for use with the 6800 microprocessor. The 6520 is also the same, except for minor differences in electrical characteristics, as the 6821 Peripheral Interface Adapter. We will refer to any of these devices (6520, 6820, or 6821) as a Peripheral Interface Adapter or PIA.

This laboratory also uses the switches attached to port A of the user 6530 device as shown in Figure 2-1 and the LEDs attached to port B of the user 6530 device as shown in Figure 3-1.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, Chapter 13.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 337-369, 405-427.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 11.

J. B. Peatman, *Microcomputer-Based Design,* McGraw-Hill, New York, 1977, pp. 489-494.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 180-193, 230-231, 237-241.

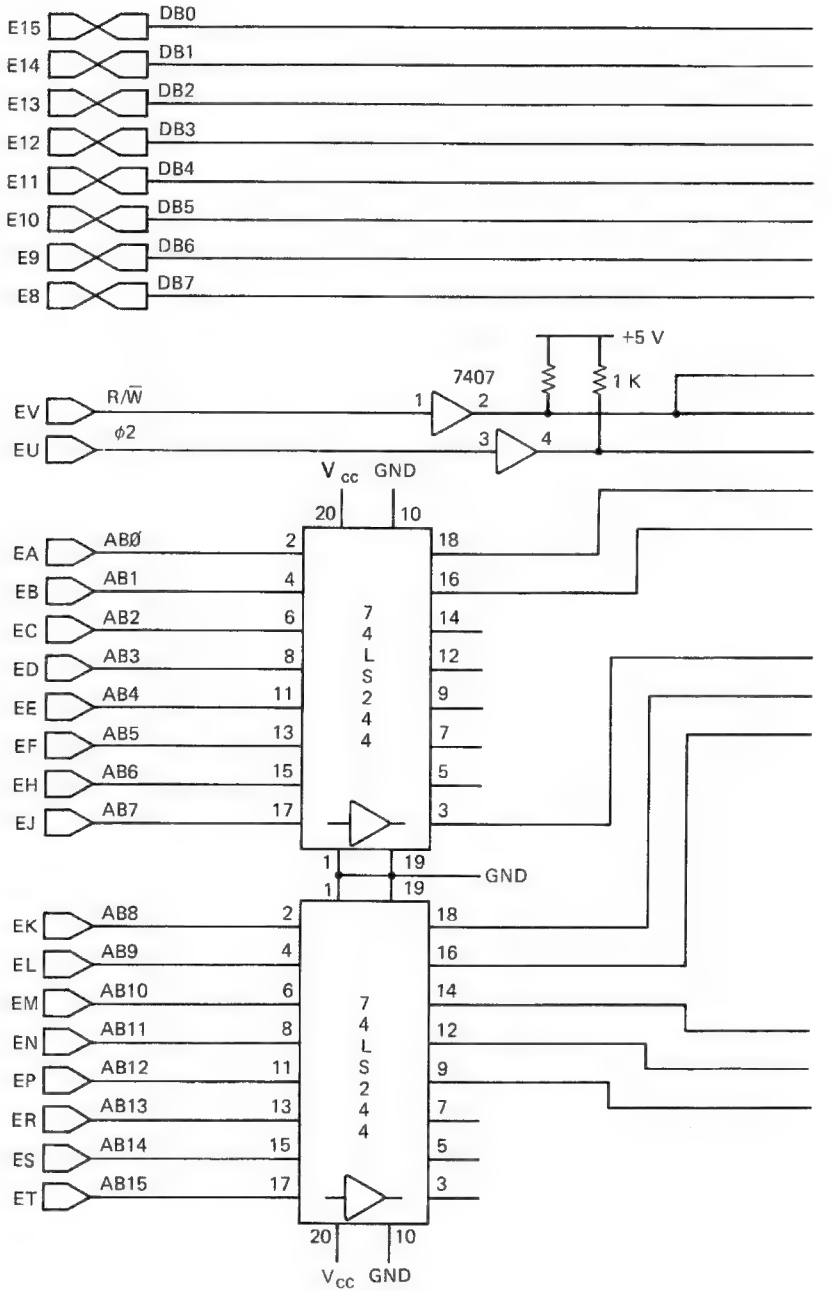
R. Tenny, "Increase KIM-1 Versatility at Low Cost," *Micro,* February 1981, pp. 57-59. This article describes how to add more I/O devices to a KIM. The magazine *Micro* is available from Micro Inc., P.O. Box 6502, Chelmsford, MA 01824.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 36-43 (flip-flops), 43-46 (counters), 46-53 (registers), 119-120 (I/O), 196-197 (I/O terms), 197-199 (I/O examples), 199-200 (I/O methods), 206-212 (polled I/O), 259-265 (6520 PIA).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapter 11.

MCS6500 Microcomputer Family Hardware Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 50-70.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 156-165.


E15
E14
E13
E12
E11
E10
E9
E8
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
+5 V
7407
1 K
EV
R/W
EU
ϕ2
1
2
3
4
V cc
GND
20
10
EA
EB
EC
ED
EE
EF
EH
EJ
ABØ
AB1
AB2
AB3
AB4
AB5
AB6
AB7
2
4
6
8
11
13
15
17
18
16
14
12
9
7
5
3
7
4
L
S
2
4
4
1
19
GND
EK
EL
EM
EN
EP
ER
ES
ET
AB8
AB9
AB10
AB11
AB12
AB13
AB14
AB15
Vcc
GND

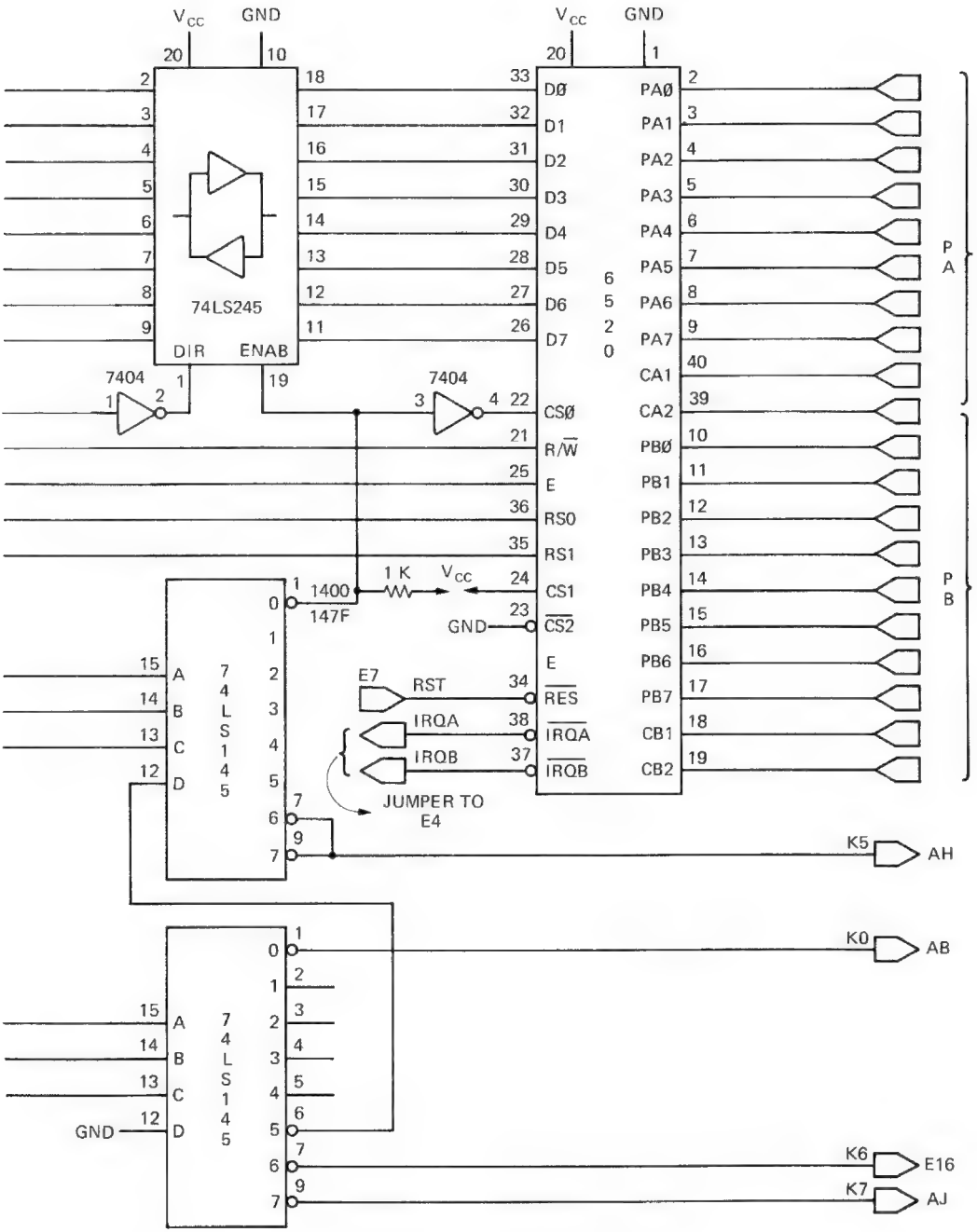


FIGURE B-1. Attachment of a 6520 PIA to the KIM-1 microcomputer.

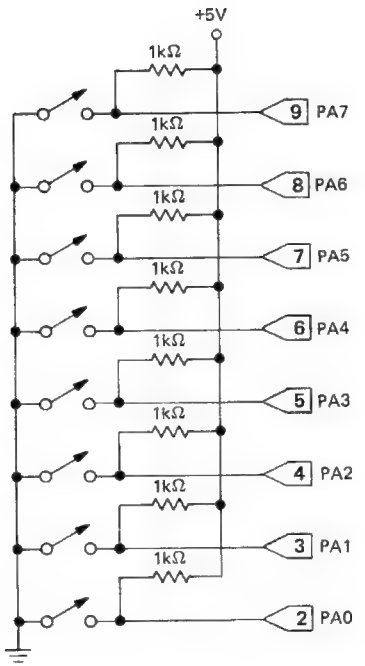


FIGURE B-2. Attachment of switches to port A of the 6520 PIA.

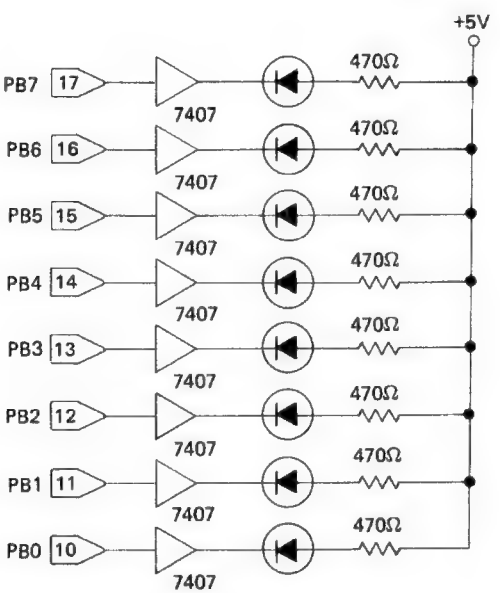


FIGURE B-3. Attachment of LEDs to port B of the 6520 PIA.

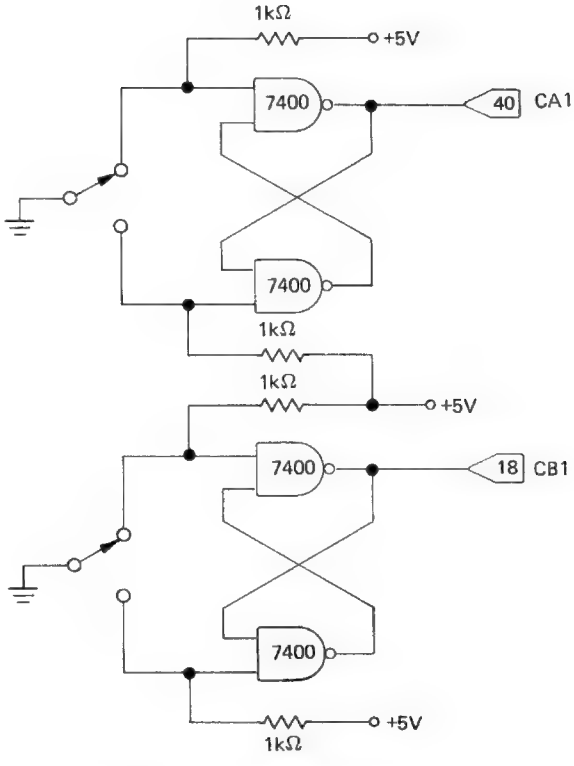


FIGURE B-4. Attachment of switches to control lines CA1 and CB1 of the 6520 PIA.

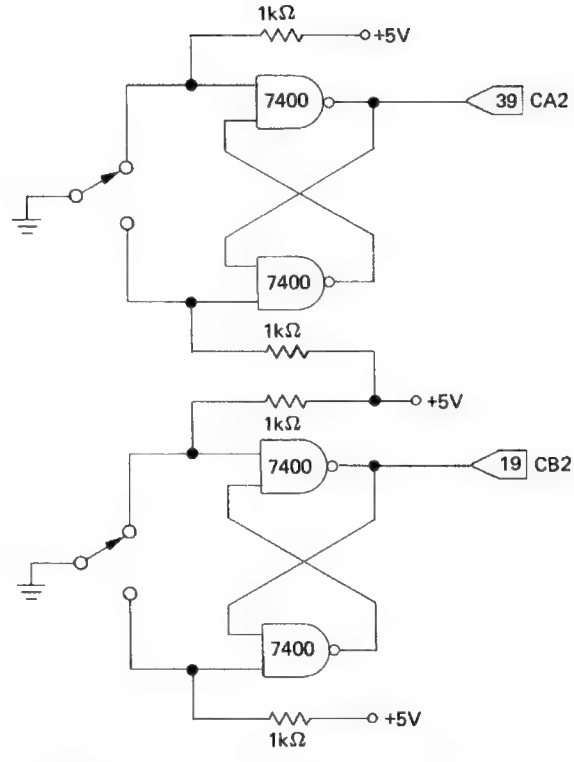


FIGURE B-5. Attachment of switches to control lines CA2 and CB2 of the 6520 PIA.

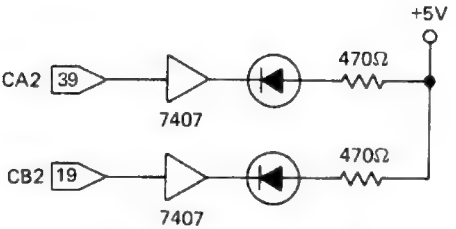


FIGURE B-6. Attachment of LEDs to control lines CA2 and CB2 of the 6520 PIA. (**Note:** Jumper wires are an easy way to select between the circuits in Figures B-5 and B-6; if you are not careful, using CA2 and CB2 as outputs could damage the AND gates in Figure B-5.)

WHAT YOU SHOULD LEARN

1) The information required to complete a data transfer successfully.
2) How synchronous and asynchronous I/O are performed.
3) The status and control signals required by an asynchronous transfer.
4) How to respond to status inputs that indicate when data or a peripheral is ready.
5) How to produce control outputs that indicate when data is available or has been accepted.
6) How to provide a complete handshake in software.
7) The features of a 6520 Peripheral Interface Adapter (PIA).
8) How to determine the operating mode of a PIA.
9) How to perform ordinary I/O using the PIA ports.
10) How to use the PIA's input control lines and interrupt flags in polling.
11) How to implement handshaking with the PIA's control lines.
12) The advantages and disadvantages of using programmable I/O devices.

TERMS

Active transition—in a PIA, the edge on the control line that sets an interrupt flag.

Asynchronous—operating without reference to an overall timing source, that is, at irregular intervals.

Buffer—temporary storage area, generally used to hold data before it is transferred to its final destination.

Buffer empty—a signal that is active when a buffer or register does not contain any data that has not been transferred to its final destination.

Buffer full—a signal that is active when a buffer or register is occupied completely with data that has not been transferred to its final destination.

Clock—a regular series of pulses that controls transitions in a system.

Control (or command) register—a register whose contents determine the state of a transfer or the operating mode of a device.

Control signal—a signal that directs an I/O transfer or changes the operating mode of a peripheral.

Counter—a clocked device that enters a different state after each clock pulse (up to its capacity) and produces an output that reflects the total number of clock pulses it has received. Counters are also called *dividers*, since they divide the input frequency by n, where n is the capacity of the counter.

Data accepted—a signal that is active when the most recent data has been transferred successfully.

Data direction register—a register that determines whether I/O lines are inputs or outputs. Abbreviated as DDR in some diagrams.

Data ready—a signal that is active when new data is available to the receiver. Same as *valid data.*

Data register—in a PIA, the actual input/output port. Also called an *output register* or a *peripheral register.*

Decoder—a device that produces unencoded outputs from coded inputs.

Handshake—a process whereby sender and receiver exchange predetermined signals to establish synchronization and to indicate the status of the data transfer.

Interrupt—a signal that temporarily suspends the computer's normal sequence of operations and transfers control to a special routine.

Interrupt flag—in a PIA, one of the bits in the control register that is set by active transitions on a control line. It is also referred to as an *interrupt request* bit (IRQ).

Interrupt request—a signal that is active when a peripheral is requesting service, often used to cause a CPU interrupt. *See also* Interrupt flag.

IRQ—*see* Interrupt flag; Interrupt request.

Latch—a storage device controlled by a timing signal. The contents of the latch are fixed at their current values by a transition of the timing signal (or clock) and remain fixed until the next transition. A latch retains its contents until they are explicitly changed.

Multiplex—use one component or system for several different purposes on a shared basis.

Output register—in a PIA, the actual input/output port. Also called a *data register* or a *peripheral register.*

Parallel interface—a device that connects a CPU to peripherals that accept or produce data in parallel (i.e., more than one bit at a time).

Peripheral Interface Adapter—a parallel interface device often used with the 6502 and 6800 microprocessors. The 6520, 6820, and 6821 versions are almost interchangeable.

Peripheral ready—a signal that is active when a peripheral can accept more data.

Peripheral register—in a PIA, the actual input/output port. Also called a *data register* or an *output register.*

PIA—*see* Peripheral Interface Adapter.

Polling—determining which I/O devices are in a particular state (e.g., ready) by examining the status of one device at a time.

Programmable I/O device—an I/O device that can have its mode of operation determined by loading registers under program control.

Ready for data—a signal that is active when the receiver can accept more data.

Status register—a register whose contents show the state of a transfer or the operating mode of a device.

Status signal—a signal that describes the current state of a transfer or the operating mode of a device or peripheral.

Strobe—a signal that identifies or describes another set of signals and that can be used to control a buffer, latch, or register.

Synchronous—operating according to an overall timing source or clock, that is, at regular intervals.

Valid data—a signal that is active when new data is available to the receiver.

ADDITIONAL FACTORS IN I/O TRANSFERS

So far we have been concerned with simple I/O. The only problems that we have encountered are ignoring meaningless fluctuations in the input data and making outputs last long enough to satisfy a peripheral or an observer. Factors that we have not considered include:

1) Whether the peripheral is ready to receive data.

The displays are always ready for data. The only question is whether an observer can see it. This is not the case, however, for a printer, teletypewriter, or motor which may be turned off, malfunctioning, or still busy responding to the previous data.

2) Whether new or valid data is available to the peripheral or the computer.

Brief changes on the output lines will not even be visible on the displays. Switch inputs can be disregarded while they are changing. Clearly, most input and output peripherals cannot be treated so casually.

3) Whether the data has been transferred successfully.

Usually, there is no doubt that data sent to a display will appear there. Nor is there a problem with missing switch inputs if the computer checks them at a reasonable rate. Again, most peripherals transfer data more rapidly and cannot be treated so casually.

BASIC I/O METHODS

For data to be transferred successfully, the following conditions must hold:

1) The receiver (computer or peripheral) must be ready.
2) The data must be available (or *valid*).
3) The receiver must take in (*accept*) the data before it changes.

So the sender must know whether the receiver is ready and whether it has accepted the data. The receiver must know whether new data is available.

One approach is to use a clock (i.e., a regular series of pulses) as a reference. The data transfers can then proceed at times determined by the clock. The receiver must be ready, the data must be available, and the

data must be accepted at particular points in the clock cycle (e.g., 100 ns after the rising edge of the first pulse). Transitions on the clock line provide timing information. The only problem is synchronizing (i.e., aligning) the receiver and the transmitter with the clock. This method is called *synchronous transfer.*

Synchronous transfer requires no additional status or control signals. The clock determines the transfer rate. The disadvantages of synchronous transfer are the need for synchronization and the dependence on a fixed clock. The only way one can change the data rate is by changing the clock. Thus synchronous transfer cannot easily accommodate peripherals that operate at varying data rates or that provide data irregularly.

An alternative approach is to use status and control signals to ensure a successful transfer. Typical signals are:

READY FOR DATA—active when the receiver can accept more data.

VALID DATA—active when new data is available.

DATA ACCEPTED—active when the receiver has accepted the most recent data.

Many variations of these signals exist and the signals may be pulses or levels. The sender must provide VALID DATA; the receiver must provide READY FOR DATA and DATA ACCEPTED. No clock is needed and transfers can proceed at any rate. This method is called *asynchronous transfer.*

The advantages of asynchronous transfer are its flexibility (since the devices determine the timing) and its simplicity (no clock or synchronization is necessary). The disadvantages of asynchronous transfer are the increased number of signals and reduced maximum data rates (since the signals must overlap properly).

The terms polling and handshaking are commonly used in describing asynchronous transfers. *Polling* is the examination of a peripheral's status to determine whether it is ready for an I/O transfer. *Handshaking* is the exchange of status and control signals to govern a data transfer; that is, the handshake validates the transfer much as a normal handshake signifies the acceptance by both parties of a contract or agreement. Of course, both sender and receiver must participate to have a true handshake.

TREATING STATUS AND CONTROL SIGNALS AS DATA

One way to handle status and control signals is to treat them as if they were data. Status signals from peripherals then act much like the binary input data we discussed in Laboratory 2. Control signals directed to peri-

pherals act much like the binary output data we discussed in Laboratory 3. The methods we used to interface switches and LEDs can serve also to respond to status inputs and produce control outputs. Since the KIM has limited I/O, we must use bits from the regular I/O ports to illustrate asynchronous transfers governed by status and control signals. In the next few examples, we will use bit 7 of port A of the user 6530 device as an input status signal and bit 7 of port B as an output control signal. This leaves us with only 7 bits of port A and 6 bits of port B for data (remember that pin PB6 of the user 6530 device is not available externally).

Selecting the directions for the 6530 I/O ports requires the following initialization program (as discussed in Laboratory 3):

```
LDA     #0          ;MAKE PORT A INPUT
STA     $1701
LDA     #$FF        ;MAKE PORT B OUTPUT
STA     $1703
```

Program B-1 is the hexadecimal version of the initialization routine.

PROGRAM B-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A9	LDA	#0
0201	00		
0202	8D	STA	$1701
0203	01		
0204	17		
0205	A9	LDA	#$FF
0206	FF		
0207	8D	STA	$1703
0208	03		
0209	17		

PROBLEM B-1

Enter Program B-1 into memory and add the following instructions:

```
LDA     #%01010101      ;LIGHT EVERY OTHER LED
STA     $1702
BRK
```

The hexadecimal additions are

020A	A9	LDA	#%01010101
020B	55		
020C	8D	STA	$1702
020D	02		
020E	17		
020F	00	BRK	

Remember that bit 6 of the output port is not connected.

Run the program. What happens? What happens when you press RS? Remember that resetting a 6530 device makes all the I/O lines inputs. What is the logic level of a 6530 input when viewed from an output peripheral?

Change memory location 0206 from FF to 55 (01010101 binary) and run the program again. What happens? Can you explain the difference?

Without pressing RS, single-step the program until it reaches memory address 020C. What happens to the LEDs? Note that the 6530 output port contains a latch. What happens if you now place FF in memory location 1703? Does changing the data direction register affect the contents of the port's output latch?

Does RESET affect the contents of the output latch? Describe a sequence of operations that will show whether your answer is correct.

When we initialize bidirectional ports, we want to avoid unnecessary transients. That is, we would like to load the port with an initial value before making it into an output. How could you accomplish this with a 6530 device? Assume that the initial value we want is FF. What happens if you do not load the port before making it into an output?

PROBLEM B-2

Although we address I/O ports as memory locations, we must remember that I/O devices behave differently from memories. For example, most I/O devices (e.g., printers, keyboards, or card readers) are either input or output devices but not both. Add the following instructions to Program B-1:

```
LDA  #%01010101   ;GET AN ALTERNATING PATTERN OF 0'S AND 1'S
STA  $1700        ;STORE IT IN THE INPUT PORT
LDA  $1700        ;TRY TO READ IT BACK
STA  $40          ;SAVE WHAT CAME BACK
BRK
```

The hexadecimal additions are

020A	A9	LDA	#%01010101
020B	55		
020C	8D	STA	$1700
020D	00		
020E	17		
020F	AD	LDA	$1700
0210	00		
0211	17		
0212	85	STA	$40
0213	40		
0214	00	BRK	

Open all the switches attached to port A of the user 6530 device and run the program. What do you find in memory location 0040? Explain what has happened. Close all the switches and run the program again. Now what do you find in memory location 0040? Does changing memory location 020B to AA (10101010 binary) have any effect on the results?

Add the following instructions after STA $40:

```
ASL   $1700   ;SHIFT THE INPUT PORT
LDA   $1700   ;NOW READ ITS CONTENTS
STA   $41     ;SAVE WHAT CAME BACK THIS TIME
BRK
```

Open all the switches and run the revised program. What values do you find in memory locations 0040 and 0041? What is the final value of the CARRY flag (bit 0 of memory location 00F1)? What changes occur if you open the switch attached to bit position 7? What switch controls the CARRY flag if you replace ASL $1700 with LSR $1700?

Replace 1700 with 0340 everywhere in the additions. Now run the program and compare memory locations 0040 and 0041. Why are the results different? What are the results if you replace the 55 in memory location 020B with AA? What happens if you replace 0340 with 194C? Explain your results. Which memory address (0340 or 194C) produces results more like those produced by 1700? Why?

Instructions may look reasonable but may not make sense physically. The microprocessor is not aware of the physical limitations of I/O devices or memory locations. Programmers who do not understand the underlying hardware may be equally unaware of those limitations and may write programs that attempt the impossible.

USING DATA LINES FOR STATUS

Let us now use bit 7 of port A as a status signal. For an input port, this signal usually indicates that new data is available (e.g., the operator has pressed a key on a keyboard, a card reader has read another card, or a cassette recorder has advanced its tape to the next position). The following program waits for the switch attached to bit 7 to be closed before reading the data from the port. It then displays the data on the LEDs attached to port B. Program B-1 initializes the 6530 device. We turn the LEDs off initially to make the results more obvious. Program B-2 is the hexadecimal version.

```
        LDA   #$FF    ;TURN OFF THE LEDS INITIALLY
        STA   $1702
WAITR   LDA   $1700   ;IS DATA READY?
        BMI   WAITR   ;NO, WAIT
        EOR   #$FF    ;YES, ACCEPT DATA
        STA   $1702   ;AND SHOW IT ON THE LEDS
        BRK
```

Open the switch attached to bit 7 and run Program B-2. What happens? Does opening or closing any of the other switches have any effect? Open all the other switches and then close the switch attached to bit 7. What happens? The processor accepts only the data that is present when the status signal becomes active. Changes that occur while the status signal is inactive are simply ignored.

PROGRAM B-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
020A	A9		LDA	#$FF
020B	FF			
020C	8D		STA	$1702
020D	02			
020E	17			
020F	AD	WAITR	LDA	$1700
0210	00			
0211	17			
0212	30		BMI	WAITR
0213	FB			

PROGRAM B-2 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0214	49	EOR	#$FF
0215	FF		
0216	8D	STA	$1702
0217	02		
0218	17		
0219	00	BRK	

Note that we actually have only seven data lines available at the input port, since we are using bit 7 for status. A separate port could hold 8 independent status bits, but we would have to mask the bits in the middle as shown in Laboratory 2. How would you change Program B-2 to use bit 5 for status?

PROBLEM B-3

Change Program B-2 so that the status signal is active-high. Status signals in TTL logic are, in fact, usually active-low; can you suggest a reason for this choice? Change Program B-2 so that it waits for the status signal to go low and then back high. Remember to debounce the switch.

PROBLEM B-4

Extend Program B-2 so that it loads data into an array starting at memory location 0340. It should load an item from the switches into the array each time the status signal becomes active (low). Remember to debounce the switch.

PROBLEM B-5

Make the program of Problem B-4 stop after loading 10 items into the array. How would you make the program stop when it finds a zero entry?

For an output port, the status signal indicates that the peripheral is ready for more data (e.g., a printer has finished with the last character or a remote station has accepted the previous transmission). The following program waits for the switch attached to bit 7 of port A to be closed before sending data from memory location 0340. Program B-3 is the hexadecimal version. As before, Program B-1 is necessary to initialize the user 6530 device.

```
        LDA     #$FF    ;TURN OFF THE LEDS INITIALLY
        STA     $1702
WAITR   LDA     $1700   ;IS PERIPHERAL READY?
        BMI     WAITR   ;NO, WAIT
        LDA     $0340   ;YES, SEND DATA
        EOR     #$FF    ;WITH REVERSED POLARITY
        STA     $1702
        BRK
```

PROGRAM B-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
020A	A9		LDA	#$FF
020B	FF			
020C	8D		STA	$1702
020D	02			
020E	17			
020F	AD	WAITR	LDA	$1700
0210	00			
0211	17			
0212	30		BMI	WAITR
0213	FB			
0214	AD		LDA	$0340
0215	40			
0216	03			
0217	49		EOR	#$FF
0218	FF			
0219	8D		STA	$1702
021A	02			
021B	17			
021C	00		BRK	

Enter Program B-3 into memory and run it with (0340) = FF. What happens before you close the switch attached to bit 7 of port A? Does it matter what is in memory location 0340? What happens when you close the status switch? Note that the old data remains on the output lines until the peripheral specifically requests new data or informs the computer that it is ready for more data.

PROBLEM B-6

Change Program B-3 so that the processor waits for the status signal to go high (i.e., the signal is active-high rather than active-low). Change the program so that it waits for the status signal to go low and then back high again. Remember to debounce the switch.

PROBLEM B-7

Make Program B-3 send data from an array starting at memory location 0340. It should send a new item from the array to the LEDs each time the status signal becomes active (low). Remember to debounce the switch. The output procedure differs slightly from the input procedure; an output peripheral usually starts in the ready state (i.e., it can accept data but the computer has not yet produced any), whereas an input peripheral usually starts in the inactive state (i.e., the computer is ready to accept data but the peripheral has not yet produced any).

Sample Data Arrays:

1) Single light moves from left to right, starting in bit position 7.

(0340) = 80
(0341) = 40
(0342) = 20
(0343) = 10
(0344) = 08
(0345) = 04
(0346) = 02
(0347) = 01

2) Start with all lights on and turn one more off each time, starting with bit 0.

(0340) = FF
(0341) = FE
(0342) = FC
(0343) = F8
(0344) = F0
(0345) = E0
(0346) = C0
(0347) = 80
(0348) = 00

Remember that bit 6 of port B is not connected, so there will be a break in both sequences.

PROBLEM B-8

Change your answer to Problem B-7 so that it exits after sending eight items from the array. How would you make your program stop after it sends a zero value?

USING DATA LINES FOR CONTROL

We can also use data lines as output control signals. A control signal intended for an input device could indicate that the computer has read the most recent data. The following program loads memory location 0340 with the data from the input port and then lights the LED attached to bit 7 by clearing bit 7 of address 1702. Remember that a logic 0 lights an LED. Program B-4 is the hexadecimal version.

```
LDA   #$FF          ;TURN OFF THE LEDS INITIALLY
STA   $1702
LDA   $1700         ;GET DATA FROM INPUT PORT
STA   $0340         ;SAVE DATA IN MEMORY
LDA   #%01111111    ;TURN CONTROL LIGHT ON
STA   $1702
BRK
```

PROGRAM B-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
020A	A9	LDA	#$FF
020B	FF		
020C	8D	STA	$1702
020D	02		
020E	17		
020F	AD	LDA	$1700
0210	00		
0211	17		
0212	8D	STA	$0340
0213	40		
0214	03		
0215	A9	LDA	#%01111111
0216	7F		
0217	8D	STA	$1702
0218	02		
0219	17		
021A	00	BRK	

Enter and run Program B-4. Here the light indicates that the computer has accepted the latest data and is ready for more. Such a signal may be referred to as READY FOR DATA, DATA ACCEPTED, or DATA BUFFER EMPTY.

PROBLEM B-9

> Revise Program B-4 so that it leaves the control light lit only long enough to be visible. Use monitor subroutine DELAY (starting address 1ED4); an initial value of 4000 hex in memory locations 17F2 and 17F3 will be adequate. Here the control signal is a pulse rather than a level.

Combining Programs B-2 and B-4 produces a program that waits for the input status signal to become active before accepting the data and then sets the output control signal to indicate that the data has been accepted. Program B-5 is the hexadecimal version of the combined program. Here we have a complete handshake (see Figure B-7); the sender indicates that new data is available and the receiver, in response, reads the data and acknowledges it.

```
        LDA   #$FF          ;TURN OFF THE LEDS INITIALLY
        STA   $1702
WAITR   LDA   $1700         ;IS DATA READY?
        BMI   WAITR         ;NO, WAIT
        STA   $0340         ;YES, ACCEPT DATA
        LDA   #%01111111    ;INDICATE DATA ACCEPTED
        STA   $1702
        BRK
```

PROGRAM B-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
020A	A9		LDA	#$FF
020B	FF			
020C	8D		STA	$1702
020D	02			
020E	17			
020F	AD	WAITR	LDA	$1700
0210	00			
0211	17			
0212	30		BMI	WAITR
0213	FB			
0214	8D		STA	$0340

PROGRAM B-5 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0215	40		
0216	03		
0217	A9	LDA	#%01111111
0218	7F		
0219	8D	STA	$1702
021A	02		
021B	17		
021C	00	BRK	

Enter and run Program B-5. The status signal indicates that new data is available and the control signal indicates that the computer has accepted the data.

An obvious problem with Program B-5 is that the light stays on indefinitely. Clearly, it should go off eventually so that it can be used in the next transfer. There are several ways to determine how long the control signal stays active (on):

1) It can remain active only briefly, thus producing a pulse that can be counted or latched if necessary. The processor must turn the signal off as well as on.

2) It can go off when the status signal becomes active again to begin the next transfer. The control signal then indicates whether the processor has accepted the most recently sent data (i.e., it acts as a BUFFER EMPTY signal).

The processor must turn the control signal off when it finds the status signal active unless there is a hardware connection.

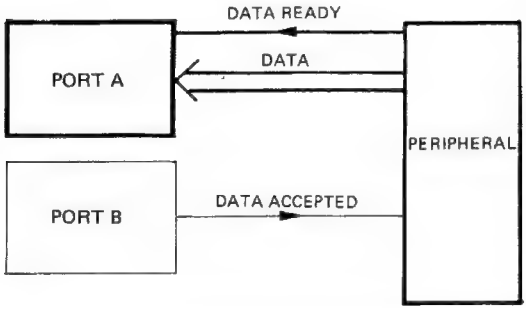
3) It can remain active for an amount of time determined by the program. This provides flexibility but requires more program intervention.

As we shall see later, the 6520 Peripheral Interface Adapter contains the circuitry required to implement any of these approaches. All the user must do is select the proper operating mode by storing the appropriate value in the PIA's control register.

PROBLEM B-10

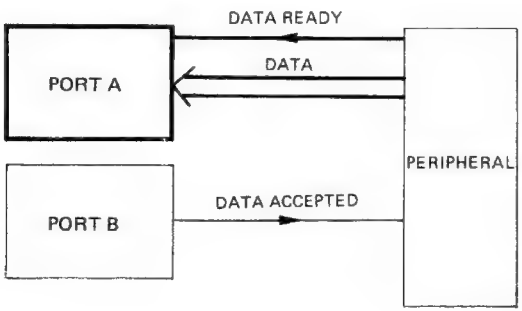
Change Program B-5 so that it starts with the control light lit and explicitly turns it off when it finds the status signal active. Single-step through your program so that you can see the light go on and off.

STEP 1
PERIPHERAL PROVIDES DATA AND ACTIVATES DATA READY.

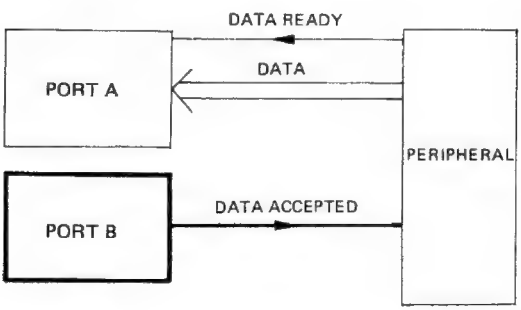


The peripheral provides both the data and an active DATA READY signal.

STEP 2
CPU RECOGNIZES THAT DATA READY IS ACTIVE AND READS THE DATA, THUS PERFORMING THE ACTUAL DATA TRANSFER.



STEP 3
CPU ACTIVATES DATA ACCEPTED, INDICATING THE SUCCESSFUL COMPLETION OF THE TRANSFER.



The peripheral can examine DATA ACCEPTED to determine when it can send more data.

FIGURE B-7. Procedure for a complete input handshake.

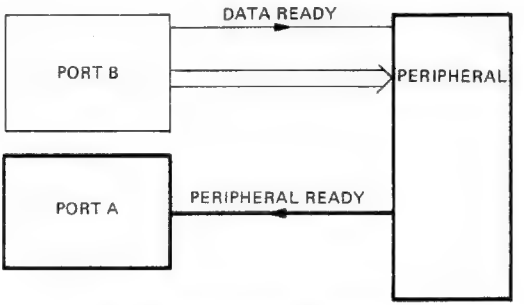
Combining Programs B-3 and B-4 produces a program that waits for the input status signal to become active before sending the data and then sets the output control signal to indicate that the data has been sent. Program B-6 is the hexadecimal version of the combined program. Here again we have a full handshake (see Figure B-8), although the order and meaning of the signals is somewhat different from the input case. The receiver indicates that it is ready to accept data; in response, the sender provides the data and indicates that it is available. Enter and run Program B-6. Change it so that it leaves the control light on only long enough to be visible.

```
        LDA   #$FF          ;TURN OFF THE LEDS INITIALLY
        STA   $1702
WAITR   LDA   $1700         ;IS PERIPHERAL READY?
        BMI   WAITR         ;NO, WAIT
        LDA   $0340         ;YES, SEND DATA
        EOR   #$FF
        AND   #%01111111    ;INDICATE DATA AVAILABLE
        STA   $1702
        BRK
```

PROGRAM B-6

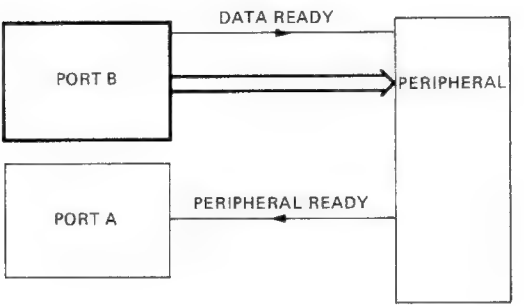
MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
020A	A9	LDA	#$FF
020B	FF		
020C	8D	STA	$1702
020D	02		
020E	17		
020F	AD	WAITR LDA	$1700
0210	00		
0211	17		
0212	30	BMI	WAITR
0213	FB		
0214	AD	LDA	$0340
0215	40		
0216	03		
0217	49	EOR	#$FF
0218	FF		
0219	29	AND	#%01111111
021A	7F		
021B	8D	STA	$1702
021C	02		
021D	17		
021E	00	BRK	

STEP 1
PERIPHERAL ACTIVATES PERIPHERAL READY, INDICATING THAT IT IS ABLE TO ACCEPT DATA.

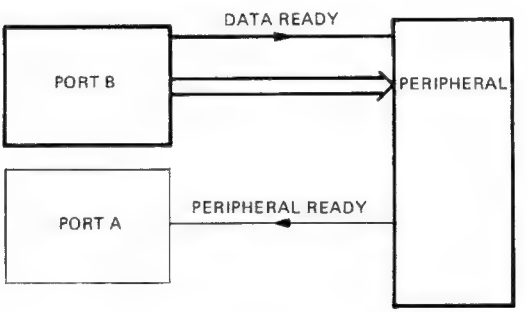


The output peripheral must provide the input status signal PERIPHERAL READY.

STEP 2
CPU RECOGNIZES THAT PERIPHERAL READY IS ACTIVE AND SENDS THE DATA, THUS PERFORMING THE ACTUAL DATA TRANSFER.



STEP 3
CPU ACTIVATES DATA READY, THUS INFORMING THE PERIPHERAL THAT NEW DATA IS AVAILABLE.



The peripheral can examine DATA READY to determine when new data is available.

FIGURE B-8. Procedure for a complete output handshake.

PROBLEM B-11

Change Program B-6 so that it starts with the control light lit and turns the light off when the status signal becomes active. Single-step through your program so that you can see the light go on and off.

6520 PERIPHERAL INTERFACE ADAPTER (PIA)

One way to simplify interfacing and I/O programming is to use LSI I/O devices. Such devices contain the circuitry required to implement common interfaces, much as an LSI microprocessor contains the circuitry required to implement a CPU. The LSI versions are smaller, cheaper, more reliable, and use less power than comparable circuits made from TTL parts or discrete elements.

We have already discussed the 6530 device. This device contains read-only memory, read/write memory, and two bidirectional I/O ports. Other I/O devices contain the circuits required to implement handshaking signals —that is, flip-flops that can be used to hold status signals, output lines that can be used to generate control signals, and connections that can be used to manage the I/O transfers and the status and control signals. A typical example is the 6520 Peripheral Interface Adapter (PIA), a parallel interface that can be used to implement a variety of handshaking circuits. The PIA not only contains two bidirectional I/O ports, but it also contains other circuits that can operate in many different useful ways. The programmer selects an operating mode by setting or clearing bits in a control register. The bits in the control register activate a particular set of circuits in the PIA, much as the bits of an instruction do in the microprocessor. The circuits in the PIA, however, are much simpler than those in the CPU and are intended for use in I/O handshakes.

If you have a PIA attached to your KIM, you should use it to implement the tasks described in Programs B-1 through B-6. Some of the references describe the PIA in detail. The key points to remember are the following:

1) An LSI I/O device can replace a large amount of hardware and software.

2) To understand the operation of LSI I/O devices, you must consider the functions for which they are intended. In the case of the PIA, think of the handshakes illustrated in Figure B-7 and B-8 and implemented in Programs B-1 through B-6.

3) To utilize an LSI I/O device, you must select the proper operating mode by setting and clearing bits in its control register. Often

a little experimentation will help you understand operating modes that manuals may explain poorly. Again, consider what the devices are intended to do. Their designers want to include as many common circuits as possible, since each one increases the potential market for the device and for board designs using the device.

Let us now briefly describe the 6520 PIA. Each PIA, like a 6530 device, has two ports called A and B. These ports are almost identical, but a few minor differences make it preferable to use A as an input port and B as an output port. Each port contains:

- A data register (the actual I/O port, also called a peripheral register) used to transfer data to or from peripherals.
- A data direction register that determines whether the I/O lines are inputs or outputs. This register is inside the PIA and is not connected to peripherals.
- A control register that determines how the PIA operates. This register is also inside the PIA and is not connected to peripherals.
- Two control lines that can be used for status and control signals as determined by the contents of the control register. These lines, like the data lines, are connected to peripherals.

Figure B-9 describes the function of each bit in a PIA's control registers. Each I/O port has its own control register and control lines. Table B-1 lists the memory addresses for the registers in the PIA attached as shown in Figure B-1.

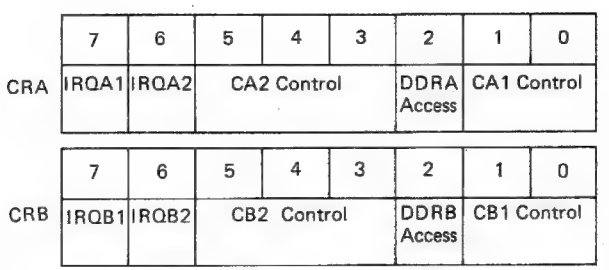


FIGURE B-9. The control registers of the 6520 PIA. (CRA is the control register for port A, CRB for port B. IRQ means interrupt request.)

Table B-1

MEMORY ADDRESSES FOR THE PIA OF FIGURE B-1

ADDRESS (HEX)	FUNCTION
1400	I/O port A or data direction register for port A*
1401	Control register for port A
1402	I/O port B or data direction register for port B*
1403	Control register for port B

*Bit 2 of PIA control register A (B) is 1 to select the I/O port and 0 to select the data direction register.

Bit 2 of the PIA control register is used for addressing; it selects either the data direction register or the I/O port (data or peripheral register) as the other address on one side of the PIA. Bit 2 = 0 to select the data direction register and 1 to select the I/O port.

Remember the following when you use a PIA:

1) You can make each bit of the I/O ports into an input or an output by setting the corresponding bit in the data direction register to 0 (input) or 1 (output).
2) The CPU can read and write the control registers, the data or peripheral registers (the I/O ports), and the data direction registers.
3) RESET clears the control and data direction registers, makes all the data lines inputs, and clears the output latches.
4) The positions (and meanings) of the bits in the control register are arbitrary and can be determined only by reading the manufacturer's documentation or other reference material.

Initializing a PIA is more difficult than initializing a 6530 device because the PIA has more options. Even without handshaking, the initialization program must perform the following steps for each port:

1) Clear bit 2 of the control register to access the data direction register. RESET does this automatically.

2) Load the data direction register with a value that produces the appropriate arrangement of inputs and outputs. The common alternatives are to load the register with zero to produce an 8-bit input port and FF (hex) to produce an 8-bit output port.

3) Set bit 2 of the control register to address the I/O port and allow data transfers to and from peripherals. Remember that the control

and data direction registers are inside the PIA and are not connected to the peripherals.

The sharing of an address by the I/O port and the data direction register is confusing. The reason for sharing is to reduce the number of pins required for internal addressing. In real applications, the shared address is only a minor nuisance; the initialization routine loads the data direction registers with the appropriate values and the rest of the program never refers to them.

The following program is a typical initialization routine for a 6520 PIA. This routine makes port A an input port and port B an output port; it does nothing with the control lines. We use the mnemonic addresses PADD and PBDD for the data direction registers and PAC and PBC for the control registers. Remember that the data direction registers actually have the same addresses as the I/O ports (which we call PAD and PBD, respectively).

```
LDA   #0           ;ADDRESS DATA DIRECTION REGISTERS
STA   PAC
STA   PBC
STA   PADD         ;MAKE PORT A INPUT
LDA   #$FF         ;MAKE PORT B OUTPUT
STA   PBDD
LDA   #%00000100   ;ADDRESS I/O PORTS (DATA REGISTERS)
STA   PAC
STA   PBC
```

Once the computer has executed this program, we can read data from port A by accessing address PAD and send data to port B by storing it in address PBD. For example, if we have switches attached to port A and LEDs attached to port B (by their cathodes), the following program will read the data from the switches and show it on the lights in positive logic (light on = open switch = logic 1, light off = closed switch = logic 0).

```
LDA   PAD          ;READ DATA FROM SWITCHES
EOR   #$FF         ;INVERT POLARITY
STA   PBD          ;SHOW DATA ON LEDS
```

PIA STATUS INPUTS

We can use the PIA's control lines to implement status signals such as DATA READY or PERIPHERAL READY. The key features of the PIA's control register are the following:

will start sending the printer characters as fast as it can obtain them. The result will be chaos, since a mechanical printer cannot keep up with a microcomputer. The problem is that bit 7 of the PIA's control register remains set, so the computer thinks the printer is always ready.

PROBLEM B-12

Revise the input program so that it uses control line CA2 instead of CA1. How would you revise the initialization routine so that bit 7 of the PIA control register is set by a positive transition on CA1? Which transition would you use if you wanted bit 7 to be set when a switch attached to CA1 was closed?

PROBLEM B-13

Extend the output program so that it sends data from an array starting at memory location 0340. Assume that it sends an item to port B each time the peripheral becomes ready. The output program should terminate after it sends a data item with the value 0D (hex), the ASCII carriage return character.

PIA CONTROL OUTPUTS

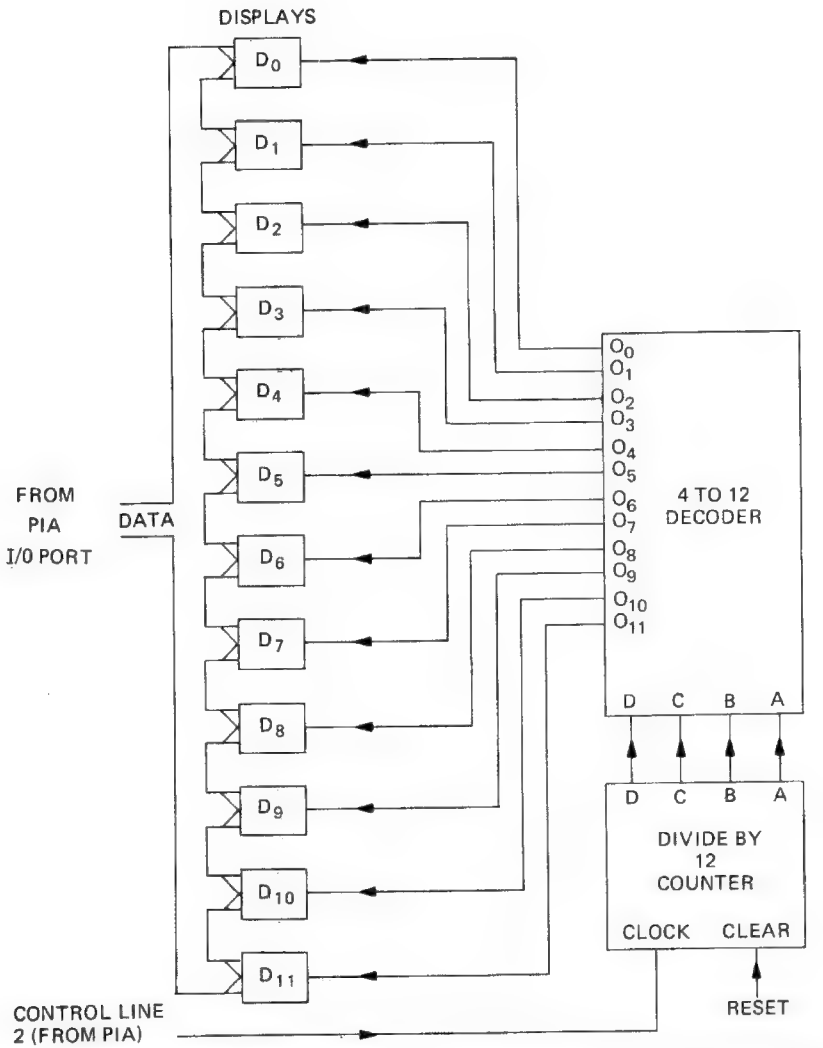
We can also use control line 2 as an output signal to indicate DATA ACCEPTED (for input ports) or DATA AVAILABLE (for output ports). The control register bits governing this mode of operation are:

1) Bit 5 determines whether control line 2 is an input (0) or an output (1).

2) If control line 2 is an output (bit 5 = 1), bit 4 determines whether that line is pulsed automatically (0) or left at a fixed level (1). These alternatives are referred to as an *automatic mode* or a *manual mode.*

3) If control line 2 is operating in the automatic mode, bit 3 determines whether the pulse (a logic 0) lasts one clock cycle (bit 3 = 1) or until the next active transition on control line 1 (bit 3 = 0). The first alternative is often used to produce a brief BYTE OUT pulse for multiplexing displays as shown in Figure B-10. The second alternative is often used to produce an acknowledgment to an intelligent peripheral (such as a printer or a terminal with its own microcomputer), indicating that the computer has accepted the latest input data or has produced new output data that the peripheral has not yet accepted. This alternative can thus be used to manage transfers between two intelligent devices.

4) If control line 2 is operating in the manual mode, bit 3 is its value. In this mode, the program can change control line 2 by setting or clearing bit 3 of the control register.



The decoder controls which display is active. Sending data to the display causes a pulse on control line 2 which clocks the counter.

FIGURE B-10. Twelve-digit multiplexed display using a counter and a decoder.

Thus the following programs produce a complete input handshake in which the processor waits for the peripheral to indicate that new data is available, reads the data, and acknowledges it.

1) Automatic mode:

```
WTRDY   BIT   PAC      ;IS PERIPHERAL READY?
        BPL   WTRDY    ;NO, WAIT
        LDA   PAD      ;YES, READ DATA AND ACKNOWLEDGE
```

The PIA clears the status and sends the acknowledgment line low automatically. No further programming is necessary.

2) Manual mode:

```
        LDA   PAC          ;BRING ACKNOWLEDGE HIGH
        ORA   #%00001000
        STA   PAC
WTRDY   BIT   PAC          ;IS PERIPHERAL READY?
        BPL   WTRDY        ;NO, WAIT
        LDA   PAD          ;YES, READ DATA AND CLEAR STATUS
        LDA   PAC          ;ACKNOWLEDGE MANUALLY
        AND   #%11110111
        STA   PAC
```

Note the use of a logical OR to set bit 3 of the control register and a logical AND to clear it. Obviously, the manual mode requires more programming than the automatic mode. The advantage of the manual mode is that the programmer has complete control over the length and polarity of the pulse on the control line. As you might expect, the automatic modes often do not match the needs of a particular application; then the manual mode comes in handy.

PROBLEM B-14

What values must you place in the control register to have the automatic input program do the following:

1) Respond to a low-to-high transition on control line CA1 and then send CA2 low for one clock cycle.
2) Respond to a high-to-low transition on control line CA1 and then send CA2 low until the next active transition on CA1.

PROBLEM B-15

Change the manual version so that it sends CA2 low initially, waits for an active transition, reads the input data, and then brings CA2 high for one millisecond.

PROBLEM B-16

Revise the automatic version so that it produces a pulse on CB2 that lasts one clock cycle after an output operation. Assume that the peripheral must be ready (indicated by a low-to-high transition on CB1) before the computer can send it data. Remember to clear bit 7 of the control register before concluding the program.

PROBLEM B-17

Change the answer to Problem B-15 so that it transfers data from an array starting at memory location 0340 and terminating with a value of 0D (hex), the ASCII carriage return character. That is, the program should send CB2 low, wait for the peripheral to become ready, send it the next data item from the array, send CB2 high for 1 ms, and then repeat the process if the character is not 0D (hex).

The automatic modes have one limitation that we have not mentioned. Control line CA2 is pulsed automatically only after port A is read, whereas control line CB2 is pulsed automatically only after port B is written. Thus we generally prefer to use port A for input and port B for output, as we remarked earlier. We can, however, overcome this limitation by means of dummy operations. That is, we can produce a pulse on control line CA2 (assuming that A is an output port) by reading back the data we have stored in the port. Similarly, we can produce a pulse on control line CB2 (assuming that B is an input port) by storing back the data we have read from the port.

PROBLEM B-18

Revise the automatic program so that it uses port B as the input port instead of port A.

PROBLEM B-19

Revise your answer to Problem B-17 so that it uses port A as the output port instead of port B.

Figure B-11 contains a summary of a PIA control register. We will discuss the interrupt-related features in Laboratory C. In the description of the output control signal, you can interpret "E" as "system clock"

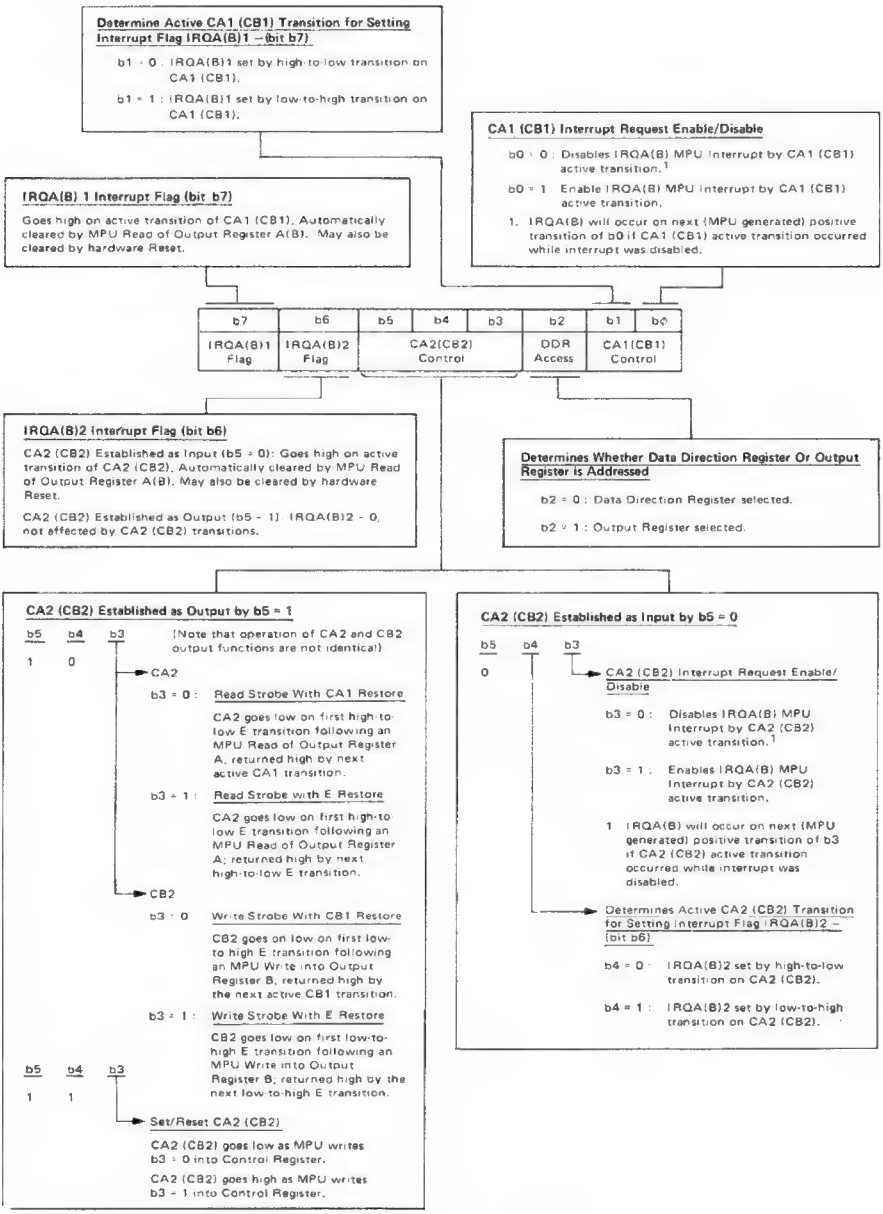


FIGURE B-11. Summary of the 6520 PIA's control register.

in most 6502-based microcomputers. Note also that MPU means "microprocessor" and IRQ means "interrupt request." Bits 6 and 7 of the PIA control register are called *interrupt request* bits or *interrupt flags*, since they are often used to cause CPU interrupts. The PIA control outputs are called *read* or *write strobes*, since they are often used to indicate to a peripheral that the CPU has read data from a particular port or written data into it.

PROGRAMMABLE I/O PORTS

The PIA has numerous operating modes (see the summary in Figure B-11). The programmability of this device means that it can operate in any of these modes, subject only to the program storing the appropriate value in the control register. The advantages of programmability are (as we noted also in Laboratory 3) that the same hardware can be used in many different applications and that changes or corrections can be made in software rather than in hardware. The disadvantages are that extra programming is necessary and that there are no standards for programmable devices. The options that are available and the ways in which they are selected are arbitrarily determined by the manufacturer and vary from device to device. For example, the functions and positions of the bits in the PIA control register are arbitrary. Similar devices from other manufacturers would have completely different registers.

However, the following features are typical of programmable I/O devices:

1) One or more command or control registers that determine how the device operates.
2) One or more status registers that contain information describing the current state of the device and the data transfer. The PIA control register is actually both a control and a status register.
3) Separate status inputs and control outputs.

Many (if not all) of the bits in the command or control registers are set during initialization to implement a particular interface. The main program does not change those bits. In the case of the PIA, most applications programs would not change the arrangement of input and output lines or the operating mode.

Note that programmable I/O devices require careful documentation. The instructions that determine their operating modes and use them are

arbitrary and are seldom described well in books or manuals. The programmer cannot expect that those who must read the documentation will understand how a specific programmable device works.

KEY POINT SUMMARY

1) Input and output transfers can proceed properly only if there is some way to determine when the receiver is ready, when new data is available, and when the receiver has accepted the data.

2) Synchronous transfers proceed according to a clock reference, while asynchronous transfers proceed through the exchange of status and control signals. The exchange of these signals is called a *handshake.*

3) Status and control signals can be implemented by using additional data ports. Such implementations are straightforward in theory, but require a large amount of software (and hardware) to coordinate the signals properly.

4) If status and control signals are treated as data, determining the status of a peripheral (*polling*) is much like determining the state of a simple two-position switch. Managing a control output is much like turning a single LED on and off. A complete input or output handshake requires an appropriate sequence of input and output operations.

5) The 6520 Peripheral Interface Adapter is an LSI device that simplifies the implementation of polling and handshaking. Besides two bidirectional I/O ports (controlled by data direction registers similar to those in the 6530 device), the 6520 also has two status inputs, latches (interrupt flags) that are set by transitions on the status lines, and two bidirectional status or control lines.

6) Bits 6 and 7 of the PIA's control register are set by transitions on the input control lines. Bits 1 and 4 of the control register determine whether positive or negative transitions are recognized. Not only do bits 6 and 7 latch transitions, but they are also cleared automatically when the CPU reads the I/O port. Thus reading the I/O port clears the interrupt flags and prepares the PIA for the next transfer. The only complication is that an output port must be read after it is written (a so-called dummy read operation) to clear the interrupt flags.

7) Control line 2 on each side of the PIA can serve as an output control signal. Bit 5 of the control register determines whether control line 2 is an input or an output. If that line is an output, there are several operating modes. In the automatic modes (bit 4 = 0), the control line is pulsed automatically after each input or output operation (input on port

A, output on port B). The pulse lasts either one clock cycle or until the next active transition on control line 1 as determined by the value of control register bit 3. In the manual mode (bit 4 = 1), the control line takes on the value of control register bit 3. The manual mode requires more programming than the automatic modes, but can provide pulses with any length and polarity.

8) Programmable I/O devices simplify hardware design. However, the lack of standards for these devices makes careful program documentation essential. Each programmable device has its own set of operating modes, ways to select those modes, and programming idiosyncrasies.

Laboratory C

Interrupts

PURPOSE

To learn when and how to use interrupts.

PARTS REQUIRED

A debounced switch attached to the microprocessor's $\overline{\text{IRQ}}$ input as shown in Figure C-1.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, pp. 183-200, 326-359.

R. Grappel, "Technique Avoids Interrupt Dangers," *EDN,* May 5, 1979, p. 88.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, Chapter 9.

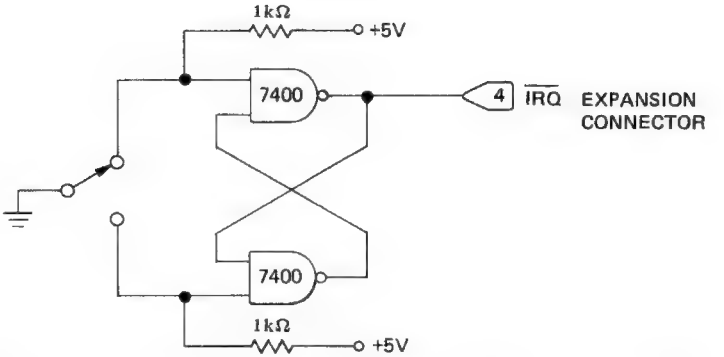


FIGURE C-1. Attachment of a debounced switch to the processor's $\overline{\text{IRQ}}$ input via the Expansion Connector. [**Note:** You should use a push-button (with three terminals) rather than a toggle switch in this circuit so that you never start a program with the interrupt active (low). If you use a toggle switch, always bring $\overline{\text{IRQ}}$ high before executing a program]

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, Chapter 12, pp. 14-1 through 14-5.

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 164-176, 185-192, 203-207.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 199-200 (I/O alternatives), 212-222 (interrupts), 264 (6520 interrupt mode), 343-345 (saving registers during interrupts), 377-378 (address vectors).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapter 18.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 39, 45-46, 75-78.

MCS6500 Microcomputer Family Hardware Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 16-27, 55, 63-64.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapter 9.

WHAT YOU SHOULD LEARN

1) The uses, advantages, and disadvantages of interrupts.
2) The interrupt inputs available on the 6502 microprocessor and the responses they produce.

3) The special interrupt-related instructions available on the 6502 microprocessor.
4) How interrupts are implemented in the KIM microcomputer.
5) How to write a simple interrupt service routine.
6) How to use interrupts to implement handshake I/O.
7) How to use the regular (maskable) interrupt.
8) How to communicate between the main program and the interrupt service routines.
9) How to buffer data that is being transferred to or from input/output devices under interrupt control.
10) When and how to change the interrupt status and the return address that have been saved in the stack.
11) How to use the 6520 PIA in an interrupt-driven mode.
12) How to handle multiple sources of interrupts by means of vectoring and polling.
13) Guidelines for programming with interrupts.

TERMS

Disable (or **disarm**)—prohibit an activity from proceeding or a device from producing data outputs.

Enable (or **arm**)—allow an activity to proceed or a device to produce data outputs.

Interrupt—a signal that temporarily suspends the computer's normal sequence of operations and transfers control to a special routine.

Interrupt-driven—dependent on interrupts for its operation, may idle until it receives an interrupt.

Interrupt mask (or **interrupt enable**)—a bit that determines whether interrupts will be recognized. A mask or disable bit usually must be cleared to allow interrupts, whereas an enable bit must be set.

Interrupt service routine—a program that performs the actions required to respond to an interrupt.

Maskable interrupt—an interrupt that the system can disable.

Nonmaskable interrupt—an interrupt that cannot be disabled within the CPU.

Polling interrupt system—an interrupt system in which a program determines the source of a particular interrupt by examining the status of each potential source separately.

Power fail interrupt—an interrupt that informs the CPU of an impending loss of power.

Priority interrupt system—an interrupt system in which some interrupts have precedence over others—that is, will be serviced first or can interrupt the others' service routines.

Programmed input/output—input/output performed under program control without using interrupts or other special hardware techniques.

Reentrant—can be executed correctly while the same routine is being interrupted or otherwise held in abeyance.

Software interrupt—*see* Trap.

Transparent routine—a routine that operates without interfering with the operations of other routines.

Trap (or **software interrupt**)—an instruction that forces a jump to a specific (CPU-dependent) address, often used to produce breakpoints or to indicate hardware or software errors.

Vectored interrupt—an interrupt that produces an identification code (or *vector*) that the CPU can use to transfer control to the appropriate service routine. The process whereby control is transferred to the service routine is called *vectoring*.

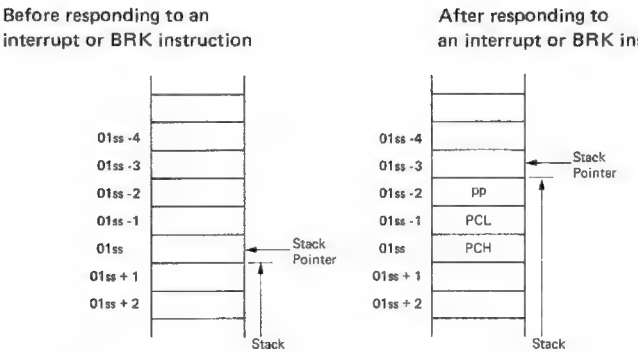
6502 INSTRUCTIONS

BRK—force break; increment the program counter by 2, set the BREAK flag to 1, save the program counter and the status register in the stack as shown in Figure C-2, and load the program counter from addresses FFFF and FFFE.

CLI—clear interrupt disable (enable interrupts); set the INTERRUPT DISABLE (I) flag to zero, thus enabling the maskable interrupt ($\overline{\text{IRQ}}$ input).

RTI—return from interrupt; load the program counter and the status register from the stack in the order shown in Figure C-2.

SEI—set interrupt disable (disable interrupts); set the INTERRUPT DISABLE (I) flag to 1, thus disabling the maskable interrupt ($\overline{\text{IRQ}}$ input).



ss = original contents of stack pointer

pp = contents of status register with INTERRUPT DISABLE flag unchanged but with BREAK flag set if response to BRK

PCH = MSBs of program counter after the execution of the current instruction.

PCL = LSBs of program counter after the execution of the current instruction.

FIGURE C-2. Saving the status of the 6502 microprocessor in the stack. (**Note:** BRK increments the program counter by 2 and sets the BREAK flag before storing anything in the stack.)

FUNCTIONS, ADVANTAGES, AND DISADVANTAGES OF INTERRUPTS

Interrupts are direct inputs to the CPU that can change its sequence of operations. An interrupt informs the CPU that something has happened, much as the ringing of a telephone informs a person that someone is on the line. The program then does not have to check READY flags or other status inputs. Instead, the inputs cause the CPU to suspend its normal operations and respond immediately.

The advantages of interrupts are:

- No need to check the status of I/O devices or determine how often their status must be checked.
- Simple hardware implementation of time delays (see Laboratory D) without involving the processor.
- Fast response because of the direct hardware connection.
- Independent execution of processing routines and input/output routines.

The disadvantages of interrupts are:

- Introduction of a random element into systems, since interrupts can occur at any time.

- Need for extra hardware to control interrupts and simplify their recognition (this hardware acts like a switchboard in a telephone system).
- Introduction of new programming problems, such as deciding when to allow interrupts and how to pass information between the main program and the interrupt service routines.

CHARACTERISTICS OF INTERRUPT SYSTEMS

Interrupt systems vary greatly from processor to processor. Typical characteristics are:

1) Number of inputs.

Each input can produce a different internal response.

2) Priority.

Some interrupts may take precedence over others (i.e., be recognized first or interrupt the others' service routines).

3) How sources are identified.

A system in which the CPU must examine the status of sources until it finds the active one is called a *polling interrupt system;* one in which each source automatically directs the CPU to the proper service routine is called a *vectored interrupt system.*

4) How control is transferred to and from the service routines.

Special CALL (or trap) and RETURN instructions are often involved.

5) How interrupts are enabled and disabled.

Interrupts may have to be disabled during system initialization, critical functions, or service routines. Some events, such as an impending loss of power, may have to be nonmaskable, since they take priority over all other system functions.

6502 INTERRUPT SYSTEM

The 6502 microprocessor has two interrupt inputs:

1) $\overline{\text{NMI}}$ is a nonmaskable interrupt generally used to respond to an impending loss of power. The input is edge-sensitive, so that it will not interrupt its own service routine.

2) $\overline{\text{IRQ}}$ is a maskable interrupt generally used for input/output and other regular system functions. The input is level-sensitive.

The 6502 microprocessor responds to an interrupt by completing its current instruction and then fetching a new value for the program counter from a fixed pair of memory locations (see Table C-1). The processor saves the program counter and the status register in the stack automatically. Note (see Figure C-2) that the status register ends up on top. The value of the program counter is the address of the next instruction in the interrupted program; there is no offset of 1 as there is with JSR.

Table C-1

MEMORY MAP FOR 6502 INTERRUPT VECTORS (COURTESY OF COMMODORE/MOS TECHNOLOGY, INC.)

VECTOR ADDRESS MS	LS	INPUT OR INSTRUCTION
FFFF	FFFE	$\overline{\text{IRQ}}$ input and BRK instruction
FFFD	FFFC	Reset
FFFB	FFFA	$\overline{\text{NMI}}$ input

The 6502 microprocessor also has an INTERRUPT DISABLE (I) flag, which is bit 2 of the status register. If this flag is 1, the maskable interrupt is disabled; if this flag is 0, the maskable interrupt is enabled. The nonmaskable interrupt is always enabled into the processor, although external gates may disable it.

SPECIAL INTERRUPT-RELATED INSTRUCTIONS AND FEATURES

The 6502 microprocessor has several instructions intended for use with interrupts:

- BRK (FORCE BREAK) increments the program counter by 2, sets the BREAK flag, saves the program counter and status register in the stack, sets the INTERRUPT DISABLE flag, and loads the program counter from addresses FFFF and FFFE (see Table C-1). This kind of instruction is often called a *trap* or *software interrupt,* since it has almost the same effect as an external interrupt.
- CLI (CLEAR INTERRUPT DISABLE) clears the INTERRUPT DISABLE flag and thus enables maskable interrupts.
- RTI (RETURN FROM INTERRUPT) loads the program counter and the status register from the stack in the order

shown in Figure C-2. RTI thus restores the registers that an interrupt response or BRK instruction saved, including the previous state of the INTERRUPT DISABLE flag.

- SEI (SET INTERRUPT DISABLE) sets the INTERRUPT DISABLE flag and thus disables maskable interrupts.

You should note the following features of the 6502 interrupt system:

1) RESET sets the INTERRUPT DISABLE flag, disabling the maskable interrupt and giving the program a chance to initialize the system before any interrupts are accepted.

2) Accepting an interrupt sets the INTERRUPT DISABLE flag and thus disables the maskable interrupt. An interrupt will therefore not disturb its own service routine. BRK also sets the I flag.

3) Accepting an interrupt results in the automatic saving of the program counter and the status register in the stack. A PHP instruction is not necessary. Note, however, that the accumulator and index registers are not saved automatically.

4) The $\overline{\text{NMI}}$ input is always enabled. It commonly serves as a power fail interrupt that causes the system to save essential data in a non-volatile memory or in a low-power memory attached to a battery. An impending loss of power (i.e., the supply voltage dropping below a specified level) clearly should take precedence over all other activities, since loss of power will ultimately stop them from proceeding anyway.

5) The BRK instruction has several unusual effects. In the first place, it increments the program counter by 2 even though it is only a 1-byte instruction. Furthermore, it is the only instruction that sets the BREAK flag. Since BRK transfers control to the same address as an $\overline{\text{IRQ}}$ input (see Table C-1), the only way a service routine can tell them apart is by examining the BREAK flag in the stack.

KIM INTERRUPTS

Table C-2 describes how the KIM handles interrupts. The key points to remember are:

- $\overline{\text{NMI}}$ is connected to the ST key and is vectored by the monitor through memory locations 17FB and 17FA. We have previously placed 1C00 in those locations so that pressing ST returns control to the monitor.
- $\overline{\text{IRQ}}$ is vectored by the monitor through addresses 17FF and 17FE. We have previously placed 1C00 in those locations also so that BRK returns control to the monitor.

Before the KIM can respond properly to interrupts, you or your program must:

1) Initialize the stack pointer, since accepting an interrupt causes the CPU to automatically save the program counter and status register in the stack.
2) Load the starting address of the service routine into memory locations 17FB and 17FA ($\overline{\text{NMI}}$) or 17FF and 17FE ($\overline{\text{IRQ}}$).
3) Enable the maskable interrupt with CLI.

Table C-2

KIM INTERRUPT ADDRESSES

INPUT	FUNCTION	LOCATION OF SERVICE ROUTINE
RESET	Reset (RS) key	1C22 in KIM monitor
$\overline{\text{NMI}}$	Nonmaskable interrupt	Address in 17FB and 17FA
BRK	FORCE BREAK instruction	Address in 17FF and 17FE
$\overline{\text{IRQ}}$	Maskable interrupt	Address in 17FF and 17FE

STOP KEY INTERRUPTS

The easiest source of interrupts to use is the stop (ST) key, since it is tied to the nonmaskable input. All that you must do to use it is:

1) Initialize the stack pointer.
2) Place the starting address of your service routine in memory locations 17FB and 17FA.

The following program waits for a stop key interrupt. Program C-1 is the hexadecimal version. Before you run it, you must place the service address (0280) in memory locations 17FA (80) and 17FB (02).

```
        LDX     #$7F    ;INITIALIZE USER STACK POINTER
        TXS
HERE    JMP     HERE    ;WAIT FOR INTERRUPT

        *=$0280         ;INTERRUPT SERVICE ROUTINE
        BRK
```

We do not have to connect the interrupt, since the connection is already part of the KIM. Nor do we have to enable the interrupt, since $\overline{\text{NMI}}$ is

always enabled. This interrupt normally stops program execution and returns control to the monitor. Note that $\overline{\text{NMI}}$ will not interrupt its own service routine, since the processor responds only to edges (1 to 0 transitions) on the $\overline{\text{NMI}}$ line. Since the ST key is debounced with a timer, the microprocessor receives a single pulse each time the key is pressed.

One problem with using $\overline{\text{NMI}}$ in this way is that we can no longer use the ST key to terminate a program. Nor can we use the single-step mode, since it also depends on the nonmaskable interrupt.

PROGRAM C-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	4C	HERE	JMP	HERE
0204	03			
0205	02			
0280	00		BRK	

PROBLEM C-1

What are the final contents of the status register, stack pointer, and index register X? What are the contents of memory locations 017D through 017F? What happens if you replace the BRK instruction in memory location 0280 with RTI (40 instead of 00)?

In most applications, the main program performs other tasks rather than just waiting for interrupts. In the following program, we simulate the main program's workload by counting in index register X. The BRK instruction in the service routine gives you a chance to examine register X (in memory location 0040) before resuming the main program. We insert a NOP after BRK, since BRK increments the program counter by 2 instead of 1. Program C-2 is the hexadecimal version.

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
        LDX   #0        ;INITIALIZE COUNTER TO ZERO
COUNT   INX             ;COUNT (SIMULATE MAIN PROGRAM)
        JMP   COUNT
```

```
*=$0280
STX     $40     ;SAVE CURRENT COUNT IN RAM
BRK             ;TRANSFER CONTROL TO OPERATOR
NOP
RTI             ;KEEP COUNTING
```

Run Program C-2 several times and see what values you find in memory location 0040. We could use this type of program to test an operator's reaction time or to select a random starting point for a game of chance or a series of questions in computer-aided-instruction.

PROGRAM C-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A2		LDX	#0
0204	00			
0205	E8	COUNT	INX	
0206	4C		JMP	COUNT
0207	05			
0208	02			
0280	86		STX	$40
0281	40			
0282	00		BRK	
0283	EA		NOP	
0284	40		RTI	

PROBLEM C-2

Change Program C-2 so that it keeps a 16-bit count in index registers X and Y (MSBs in X). Have the interrupt service routine clear the count after storing its current value in memory locations 0040 and 0041 (MSBs in 0041).

PROBLEM C-3

Change Program C-2 so that it records successive counts in memory starting at address 0040. That is, the service routine should record the count in the next available memory location and return control to the main program. Press the ST key several times and see what values you find in memory.

PROBLEM C-4

Change the program of Problem C-3 so that it displays the latest recorded count on the LEDs attached to port B of the user 6530 device. The displays should change each time you press ST. Remember to make port B an output.

PROBLEM C-5

Change the program of Problem C-4 so that the service routine shows the latest count on the two rightmost seven-segment displays (#5 and #6, see Table 5-2). Use subroutine CONVD (starting address 1F48, see Table A-1) and repeat the display 250 times before returning control to the main program. Be sure to make port A of the KIM's keyboard/display 6530 into an output by storing FF in memory location 1741. Be careful; the KIM's startup routine (entered via BRK) makes the display port an input. If you always find 88 (all segments lit) on the displays, you are probably not assigning the port as an output correctly.

HANDSHAKING WITH INTERRUPTS

Interrupts normally have specific functions in the overall system. They are often used to implement handshake input/output, as described in Laboratory B. In this case, interrupts initiate the input and output procedures shown in Figures B-7 and B-8. The interrupts inform the microprocessor directly that data is available from an input peripheral or that an output peripheral is ready to receive data.

The following program (see Program C-3 for a hexadecimal version) is the interrupt-driven equivalent of Program B-2. It loads an entry from port A of the user 6530 device when you press the ST key. The main program clears the DATA READY flag (memory location 0040) and waits for the interrupt service routine to set it. In response to the interrupt, the service routine loads the data from the input port and sets the DATA READY flag. Note the extra overhead involved in Program C-3 as compared to Program B-2; the interrupt system must be initialized and the DATA READY flag must be cleared, set, and tested. On the other hand, the main program need not examine the interrupt flag and the service routine can proceed independently when it is activated.

PROGRAM C-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#0

PROGRAM C-3 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0204	00			
0205	85		STA	$40
0206	40			
0207	A5	WTRDY	LDA	$40
0208	40			
0209	F0		BEQ	WTRDY
020A	FC			
020B	00		BRK	
0280	48		PHA	
0281	E6		INC	$40
0282	40			
0283	AD		LDA	$1700
0284	00			
0285	17			
0286	85		STA	$41
0287	41			
0288	68		PLA	
0289	40		RTI	

```
        LDX     #$7F        ;INITIALIZE USER STACK POINTER
        TXS
        LDA     #0          ;CLEAR READY FLAG
        STA     $40
WTRDY   LDA     $40         ;IS DATA READY?
        BEQ     WTRDY       ;NO, WAIT
        BRK                 ;YES, DONE

        *=$0280             ;INTERRUPT SERVICE ROUTINE
        PHA                 ;SAVE ACCUMULATOR
        INC     $40         ;SET READY FLAG
        LDA     $1700       ;GET DATA
        STA     $41
        PLA                 ;RESTORE ACCUMULATOR
        RTI
```

The service routine need only save and restore the accumulator. The processor saves the status register automatically in response to the interrupt and the service routine does not use either index register. If the service routine used an index register, we would have to save and restore its original value via the accumulator as discussed in Laboratory A.

Try running Program C-3 for various inputs at port A. What are the final values in memory locations 0040 and 0041? Why do we need the READY flag in memory location 0040? Why couldn't the service routine simply set the ZERO flag before returning control to the main program?

PROBLEM C-6

What is the value of the INTERRUPT DISABLE flag at the beginning of the service routine? Remember that I is bit 2 of the status register. Does it matter if you insert CLI after STA $40 in the main program? Does this insertion affect the value of the I flag that is stored in the stack?

PROBLEM C-7

Change Program C-3 so that it displays the data on the LEDs attached to port B of the user 6530 device each time you press ST.

PROBLEM C-8

Change Program C-3 so that it saves input data values starting at memory location 0340 (hex). That is, each time you press ST, the program should read the data from port A and store it in the next available location. Use memory location 0040 for the current buffer index and make the interrupt service routine save and restore any registers it uses.

PROBLEM C-9

Change Program C-3 so that you can use ST to produce input and output interrupts on an alternating basis. The first time you press ST should cause the processor to load memory location 0041 with the data from port A. The next time should cause the processor to display the contents of 0041 on the LEDs attached to port B, and so on.

Hint: Use memory location 0040 as an input/output flag (0 means input, 1 means output). Have the main program clear 0040 initially and let the service routine use its value to decide what to do and change its value before exiting.

PROBLEM C-10

Make the main part of Program C-3 wait for a 7F input, the synchronization character that we discussed in Laboratory 2. If the input is not 7F, the main program should clear the DATA READY flag and wait for the next input.

PROBLEM C-11

Write an interrupt-driven version of the output routine in Program B-3. The main program should clear memory location 0040 to indicate that data is available for output. When an interrupt occurs, the service routine should send the data from

memory location 0041 and set memory location 0040 to 1 to indicate that the data has been transmitted.

PROBLEM C-12

Change the answer to Problem C-11 so that the service routine sends the data only if it is a synchronization character (7F hexadecimal). Otherwise, the service routine does nothing. Note that all the LEDs will stay off unless (0041) = 7F, assuming that you turn them off initially by storing FF in memory location 1702.

REGULAR (MASKABLE) INTERRUPTS

To use $\overline{\text{IRQ}}$, we must load the service address (0280) into memory locations 17FE and 17FF (see Table C-2), instead of 17FA and 17FB. An added complication is that we must explicitly enable $\overline{\text{IRQ}}$ with a CLI instruction before we check the READY flag. We must also explicitly disable $\overline{\text{IRQ}}$ with an SEI instruction before returning control to the monitor. A further problem is that we cannot use BRK to return control to the monitor since it uses the same vector as $\overline{\text{IRQ}}$. The KIM does, however, provide an alternative terminating instruction: JSR $1C05. If the monitor is entered in this way, the KIM will display the last byte of the JSR instruction (the address JSR saves in the stack—see Laboratory A). The contents of the accumulator are lost.

If we make all these changes, we can revise Program C-3 to use $\overline{\text{IRQ}}$ instead of $\overline{\text{NMI}}$. Program C-4 is the hexadecimal version of the revision. Try running Program C-4 for various inputs at port A. Now you must control the interrupt from the switch attached to $\overline{\text{IRQ}}$, rather than from the ST key. You do not have to reenable the interrupt at the end of the service routine, since RTI restores the original INTERRUPT DISABLE flag automatically. Obviously, that flag must have been 0 or the interrupt would not have been recognized in the first place.

PROGRAM C-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#0
0204	00		
0205	85	STA	$40
0206	40		

PROGRAM C-4 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0207	58		CLI	
0208	A5	WTRDY	LDA	$40
0209	40			
020A	F0		BEQ	WTRDY
020B	FC			
020C	78		SEI	
020D	20		JSR	$1C05
020E	05			
020F	1C			

```
        LDX     #$7F        ;INITIALIZE USER STACK POINTER
        TXS
        LDA     #0          ;CLEAR READY FLAG
        STA     $40
        CLI                 ;ENABLE CPU INTERRUPT
WTRDY   LDA     $40         ;IS DATA READY?
        BEQ     WTRDY       ;NO, WAIT
        SEI                 ;YES, DISABLE CPU INTERRUPT
        JSR     $1C05       ;RETURN TO KIM MONITOR

        *=$0280
        PHA                 ;SAVE ACCUMULATOR
        INC     $40         ;SET READY FLAG
        LDA     $1700       ;GET DATA
        STA     $41
        PLA                 ;RESTORE ACCUMULATOR
        RTI
```

The interrupt service routine is exactly the same as in Program C-3. We must be careful to initialize the stack pointer and the READY flag before allowing interrupts. What would happen in Program C-3 if an interrupt occurred before the READY flag was cleared? This causes no problems in our simple system, since we are producing interrupts at a slow rate. In a real system, however, an early interrupt could be lost or cause the processor to wander off aimlessly. This initialization problem is one reason why the nonmaskable interrupt is seldom used except to force a response to an impending power failure.

The main program knows that new data has been received because the DATA READY flag in memory location 0040 has been set. In a real application, it would clear that flag and process the data. It might also

disable the interrupt so that no more data would be accepted until the current data had been processed.

If we are using the regular interrupt, we can restore the ST key to its normal function by storing 1C00 in memory locations 17FA and 17FB. We still cannot, however, use the single-step mode, since it depends on nonmaskable interrupts that take priority over regular interrupts. The processor will not respond to regular interrupts if we are executing a program in the single-step mode.

PROBLEM C-13

What is the value of the INTERRUPT DISABLE flag at the beginning of the service routine? What is the value that is stored in the stack? What happens if you omit the CLI instruction (i.e., replace it with a NOP)?

PROBLEM C-14

Revise your answer to Problem C-8 so that it uses the regular interrupt rather than the nonmaskable interrupt.

Note: Be careful when you execute your program. Since we have no way to clear the interrupt, you should transfer control back to the monitor after each data transfer (i.e., replace RTI with JSR $1C05). Then bring the interrupt back high before resuming your program. Otherwise, each interrupt will be recognized many times. This iteration does not occur when you use $\overline{\text{NMI}}$, because $\overline{\text{NMI}}$ is edge-sensitive whereas $\overline{\text{IRQ}}$ is level-sensitive.

COMMUNICATING WITH INTERRUPT SERVICE ROUTINES

We generally avoid using registers to transfer data to or from an interrupt service routine. We obviously cannot use the status register since RTI restores its original value. The other registers are seldom available either, since the main program needs them for its own functions and the interrupt service routine cannot depend on finding specific values in them. Furthermore, if a service routine changes the registers, it and the main program become implicitly dependent on each other. The programmer cannot revise or replace one without changing the other. Programming (particularly debugging and maintenance) is much simpler if the service routines are transparent to the main program; that is, the interaction should be explicit and hence easy to understand, correct, and change.

How, then, should the main program communicate with the interrupt service routines? One approach is to assign memory locations that the main program can use to send or receive data. These locations serve the same purpose as the mail drops used in popular spy movies; that is, they provide an indirect means of communication that participants can use

according to a set of established rules. Note how the spy and the informant use a mail drop. The spy places orders, requests, and payments in the drop; the informant picks up the mail and provides the required information. The spy and the informant never talk or meet. Neither needs to (or should) even know who the other is; either person can be replaced without affecting the transfer of information, as long as the substitute follows the rules. Explicit rules for communication and limited interaction make an intelligence network or a program easy to maintain and update.

For example, in Program C-3, the main program clears memory location 0040 and waits for the interrupt service routine to change it. When that happens, the main program exits. Here the interrupt acts like a RUN command, causing the main program to proceed. The flag values we have chosen are arbitrary; we could just as easily have the main program set the READY flag and the interrupt service routine clear it.

PROBLEM C-15

Make Program C-3 use bit 7 of memory location 0040 as the READY flag. This approach leaves the 7 least significant bits of that location available for data (this is particularly convenient if the input data consists of 7-bit ASCII characters).

PROBLEM C-16

Make Program C-3 wait until the value in memory location 0040 is ten. How would you make the main program wait until memory location 0040 has the same value as memory location 0020? This approach is useful when the computer must count a certain number of external events, such as pulses on a clock line or activations of a sensor.

We should note the disadvantages of using specific memory locations to communicate between the main program and the interrupt service routine. The locations are sometimes called a *mailbox*, since messages are transferred through a temporary storage place much as they are in standard postal services. The problems with this approach are:

1) The transfer is indirect and awkward. Everything has to be handled precisely so that the receiver can process the information without any direct communications (i.e., without asking questions or seeking clarification).

2) The mailbox must be checked often enough to avoid missing messages. The receiver may have to acknowledge each message to inform the sender that the information has been transferred properly.

3) The approach may involve a large amount of overhead. The sender must prepare the messages, the receiver must interpret them, and the sender must wait for the messages to be picked up. All of this slows the rate at which information can be transferred.

4) Other programs may accidentally use the mailbox, thus destroying the information in it. Imagine an informant hiding valuable papers in a trash container. Unfortunately, before the spy can retrieve the papers, the sanitation department collects the trash.

BUFFERING INTERRUPTS

Program C-3 handles the input data one character at a time. Clearly, this creates problems if the data rate is high or if only sequences of data are meaningful (as is commonly the case when the inputs are coming from a terminal or communications line). The obvious solution is to buffer the data in the computer's memory. Then the interrupt service routine can fill a buffer and the main program need not be concerned with each character separately. The buffer serves the same purpose as a buffer memory in a printer or terminal. The computer can send the peripheral a large amount of data at one time, and the peripheral can handle the data at its own speed without tying up the computer. In our situation, the interrupt service routine takes on the role of the peripheral with its own local memory.

In the following program (see Program C-5 for a hexadecimal version), the main program waits until the count in memory location 0040 (originally set to 0) reaches 4. It also stores the inputs in successive memory locations starting at address 0340.

```
        LDX     #$7F        ;INITIALIZE USER STACK POINTER
        TXS
        LDA     #0
        STA     $40         ;CLEAR COUNT TO START
        LDA     #4          ;GET REQUIRED COUNT
WTCNT   CMP     $40         ;ENOUGH INPUTS RECEIVED?
        BNE     WTCNT       ;NO, CONTINUE
        BRK                 ;YES, DONE

        *=$0280
        PHA                 ;SAVE ACCUMULATOR, X REGISTER
        TXA
        PHA
        LDX     $40         ;COUNT = BUFFER INDEX
        LDA     $1700       ;GET INPUT DATA
        STA     $0340,X     ;STORE DATA IN BUFFER
```

```
        INC     $40     ;INCREMENT BUFFER INDEX
        PLA             ;RESTORE ACCUMULATOR, X REGISTER
        TAX
        PLA
        RTI
```

PROGRAM C-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#0
0204	00			
0205	85		STA	$40
0206	40			
0207	A9		LDA	#4
0208	04			
0209	C5	WTCNT	CMP	$40
020A	40			
020B	D0		BNE	WTCNT
020C	FC			
020D	00		BRK	
0280	48		PHA	
0281	8A		TXA	
0282	48		PHA	
0283	A6		LDX	$40
0284	40			
0285	AD		LDA	$1700
0286	00			
0287	17			
0288	9D		STA	$0340,X
0289	40			
028A	03			
028B	E6		INC	$40
028C	40			
028D	68		PLA	
028E	AA		TAX	
028F	68		PLA	
0290	40		RTI	

Note that we must save the accumulator and index register X explicitly (remember the sequences in Laboratory A). The processor saves only the program counter and the status register automatically. The service routine need not, however, save and restore index register Y since it does not use Y.

Enter and run Program C-5. Remember to put 1C00 back in memory locations 17FE and 17FF, since we have returned to using the non-maskable interrupt. Set the switches to form the following array:

(0340) = F0 (11110000 binary)
(0341) = 0F (00001111 binary)
(0342) = AA (10101010 binary)
(0343) = 55 (01010101 binary)

PROBLEM C-17

Make Program C-5 fill the buffer until it receives an input of 0D (hex), the ASCII carriage return character. Use memory location 0041 as an END OF LINE flag. The main program should clear the flag initially and then wait for it to be set. The service routine should set the flag when it receives a 0D input. A program like this handles input from a terminal one line at a time.

PROBLEM C-18

Make Program C-5 fill the buffer with a message that starts with an ASCII STX (Start of Text) character (02 hex) and ends with an ASCII ETX (End of Text) character (03 hex). All inputs before the STX are simply ignored. The STX and ETX characters themselves are omitted from the buffer. Such control characters are often used for synchronization.

Example:

If the inputs are (in order of receipt)

67	
B2	
02	ASCII STX
47	ASCII G
5F	ASCII O (letter)
0D	ASCII Carriage Return
03	ASCII ETX

The final buffer contents are

(0340) = 47	ASCII G
(0341) = 5F	ASCII O (letter)
(0342) = 0D	ASCII Carriage Return

The two inputs preceding the STX are ignored and the STX and ETX characters do not appear in the buffer.

Hint: Use memory location 0042 as a TRANSMISSION IN PROGRESS flag. The main program should clear that flag initially and the service routine should set the flag when it receives an STX input.

PROBLEM C-19

A common practice is to allow the interrupt service routine to fill one buffer while the main program processes another. This practice is known as *double buffering*, for obvious reasons. The advantage is that both the main program and the interrupt service routine have their own buffers, thus further reducing how often they must communicate. Extend Program C-5 so that it first fills (with four inputs) the buffer starting in memory location 0340 and then fills the buffer starting in memory location 0360. Use memory location 0041 as a flag that indicates whether the first buffer is full (0 means empty, 1 means full), and memory location 0042 as a similar flag for the second buffer. Use memory locations 0043 and 0044 to hold the address of the buffer that the interrupt service routine is using currently.

Example:

Initially, memory locations 0041 and 0042 are both cleared. If the inputs are (in order of receipt)

FE	(all switches open except #0)
FD	(all switches open except #1)
FB	(all switches open except #2)
F7	(all switches open except #3)
EF	(all switches open except #4)
DF	(all switches open except #5)
BF	(all switches open except #6)
7F	(all switches open except #7)

the first four values are placed in the first buffer and the second four values in the second buffer. Memory location 0041 is set to 1 after the first four inputs have been received and memory location 0042 is set to 1 after the second four inputs have been received.

The values in the first buffer are

(0340) = FE

(0341) = FD

(0342) = FB

(0343) = F7

The values in the second buffer are

(0360) = EF

(0361) = DF

(0362) = BF

(0363) = 7F

PROBLEM C-20

In real applications of double buffering, the main program processes the data in one buffer, while the interrupt service routine fills the other buffer. When those operations have been completed, the main program switches buffers. Revise your answer to Problem C-19 so that the main program operates continuously, switching buffers whenever it finds that the interrupt service routine has filled one. We are thus essentially assuming that the main program always finishes processing the data in one buffer before the interrupt service routine fills the other buffer. Use memory locations 0043 and 0044 to hold the starting address of the current input data buffer and memory locations 0045 and 0046 to hold the starting address of the current processing buffer.

Example:

We start with

(0043) = 40 (input data buffer)

(0044) = 03

After the first four inputs have been received (filling the first buffer), we have

(0043) = 60 (input data buffer)

(0044) = 03

(0045) = 40 (processing buffer)

(0046) = 03

After the second four inputs have been received (filling the second buffer), we have

(0043) = 40 (input data buffer)

(0044) = 03

(0045) = 60 (processing buffer)

(0046) = 03

We are assuming that the main program has processed the previous data by the time the service routine fills the next buffer.

PROBLEM C-21

Write an interrupt-driven output routine that transmits four values from a buffer starting at memory location 0340.

Examples:

1) Single light moves one position to the left with each interrupt.

(0340) = 01

(0341) = 02

(0342) = 04

(0343) = 08

2) Start with all lights on and turn one off with each interrupt, starting with the one attached to bit 0. Remember that bit 6 is not connected.

(0340) = FF

(0341) = FE

(0342) = FC

(0343) = F8

Remember to complement the data before sending it to the LEDs.

PROBLEM C-22

Write an interrupt-driven output routine that continues transmitting data from the buffer starting at memory location 0340 until it encounters a value of 0D (hex), the ASCII carriage return character.

Example:

(0340) = 80

(0341) = 40 (remember, PB6 is not connected)

(0342) = 20

(0343) = 10

(0344) = 08

(0345) = 04

(0346) = 02

(0347) = 01

(0348) = 0D

The output should appear as a single light that moves one position to the right after each interrupt. The program should exit with the displays showing 0D (hex), the carriage return character.

6520 PIA INTERRUPTS

Most 6502 interrupts come from peripherals attached to the processor through programmable devices such as the 6520 Peripheral Interface Adapter. Thus we must discuss how the PIA handles interrupts. The relevant bits in the control register are (see Tables C-3 and C-4):

- Bit 0 determines whether active transitions on control line 1 cause interrupts. Note that a value of 1 allows interrupts; bit 0 is therefore an INTERRUPT ENABLE, opposite in polarity to the microprocessor's INTERRUPT DISABLE flag. Remember that control register bit 1 determines which transitions are active on control line 1.

Table C-3

CONTROL OF INTERRUPT INPUTS CA1 AND CB1*

CRA-1 (CRB-1)	CRA-0 (CRB-0)	INTERRUPT INPUT CA1 (CB1)	INTERRUPT FLAG CRA-7 (CRB-7)	CPU INTERRUPT REQUEST $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$)
0	0	↓ Active	Set high on ↓ of CA1 (CB1)	Disabled—$\overline{\text{IRQ}}$ remains high
0	1	↓ Active	Set high on ↓ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high
1	0	↑ Active	Set high on ↑ of CA1 (CB1)	Disabled—$\overline{\text{IRQ}}$ remains high
1	1	↑ Active	Set high on ↑ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high

* ↑, positive transition (low to high); ↓, negative transition (high to low). The interrupt flag bit CRA-7 is cleared when the CPU reads the A Data Register, and CRB-7 is cleared when the CPU reads the B Data Register. If CRA-0 (CRB-0) is low when an interrupt occurs (interrupt disabled) and is later brought high $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$) occurs after CRA-0 (CRB-0) is written to a "one."

Table C-4

CONTROL OF CA2 AND CB2 AS INTERRUPT INPUTS
CRA5 (CRB5) IS LOW (0)*

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRB-3)	INTERRUPT INPUT CA2 (CB2)	INTERRUPT FLAG CRA-6 (CRB-6)	CPU INTERRUPT REQUEST $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$)
0	0	0	↓ Active	Set high on ↓ of CA2 (CB2)	Disabled—$\overline{\text{IRQ}}$ remains high
0	0	1	↓ Active	Set high on ↓ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high
0	1	0	↑ Active	Set high on ↑ of CA2 (CB2)	Disabled—$\overline{\text{IRQ}}$ remains high
0	1	1	↑ Active	Set high on ↑ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high

* ↑, positive transition (low to high); ↓, negative transition (high to low). The interrupt flag bit CRA-6 is cleared when the CPU reads the A Data Register and CRB-6 is cleared when the CPU reads the B Data Register. If CRA-3 (CRB-3) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{\text{IRQA}}$ ($\overline{\text{IRQB}}$) occurs after CRA-3 (CRB-3) is written to a "one."

- Bit 3 determines whether active transitions on control line 2 cause interrupts. Like bit 0, bit 3 must be 1 to allow interrupts. This bit only enables interrupts if control line 2 is an input (i.e., if control register bit 5 = 0). Remember that control register bit 4 determines which transitions are active on control line 2.

To use the PIA connected as shown in Figure B-1 in an interrupt-driven mode, you must do the following:

1) Initialize the stack pointer.
2) Load the starting address of the service routine into memory locations 17FE and 17FF.
3) Initialize the PIA with its interrupts enabled. This means setting bit 0 to enable interrupts from control line 1 and setting bit 3 to enable interrupts from control line 2.
4) Enable the CPU's maskable interrupt with CLI.

Using $\overline{\text{IRQ}}$ rather than $\overline{\text{NMI}}$ does not cause the difficulties here that it does in Problem C-14. When we use the switch shown in Figure C-1 to

cause an interrupt directly, we have no way to deactivate the interrupt after servicing it (except by hand). When we insert a PIA between the interrupt switch and the CPU, the PIA sets an interrupt flag when the switch is opened or closed and we can clear the flag by reading the PIA's data register (see Laboratory B).

The following program uses port A of the PIA as an interrupt-driven input port. It performs the required initialization, waits for the interrupt to occur, and then exits. The service routine reads the data from the port, thus clearing the interrupt flag.

```
        LDX   #$7F          ;INITIALIZE USER STACK POINTER
        TXS
        LDA   #0            ;ACCESS DATA DIRECTION REGISTERS
        STA   PAC
        STA   PBC
        STA   PADD          ;MAKE PORT A INPUT
        LDA   #$FF          ;MAKE PORT B OUTPUT
        STA   PBDD
        LDA   #%00000101    ;ENABLE DATA TRANSFERS, INTERRUPTS
        STA   PAC
        STA   PBC
        LDA   #0            ;CLEAR READY FLAG
        STA   $40
        CLI                 ;ENABLE CPU INTERRUPT
WTRDY   LDA   $40           ;IS DATA READY?
        BEQ   WTRDY         ;NO, WAIT
        SEI                 ;YES, DISABLE INTERRUPT
        JSR   $1C05         ;EXIT TO KIM MONITOR

        *=$0280             ;INTERRUPT SERVICE ROUTINE
        PHA                 ;SAVE ACCUMULATOR
        INC   $40           ;SET READY FLAG
        LDA   PAD           ;READ INPUT DATA
        STA   $41
        PLA                 ;RESTORE ACCUMULATOR
        RTI
```

Since the interrupt enable for control line 1 is bit 0 of the PIA control register, you can disable that interrupt (if you know it is enabled) with the instruction DEC PIACR or enable it (if you know it is disabled) with INC PIACR. This trick saves time and memory, but the straightforward approach is clearer.

PROBLEM C-23

Change the interrupt-driven input program so that it loads the input data in response to a low-to-high transition on control line CA2.

PROBLEM C-24

Write an interrupt-driven output program for a PIA. The program should respond to a high-to-low transition on CB1 by sending the contents of memory location 0041 to port B. Remember to clear the interrupt flag with a "dummy read" (BIT PBD).

CHANGING VALUES IN THE STACK

Occasionally, the interrupt service routine must modify the main program's registers. The most common reasons are to change the return address or to disable the overall interrupt system. An example of a situation in which the return address must be changed is the response to a BRK instruction inserted by a debugger for breakpointing purposes. Here the service routine must decrement the return address by 2 so that it can replace the breakpoint with the original operation code, display the original code and its address, and resume the program correctly. Note that a debugger typically replaces the original operation code with a BRK instruction, much as we did by hand in Laboratory 8. Other situations in which we may want to modify values in the stack are errors or exceptions that may require special exits and cases in which the maskable interrupts must be disabled. Remember that RTI reenables the maskable interrupts automatically by restoring the status register from the stack.

Disabling the maskable interrupt is simple, since the 6502 stores the status register at the top of the stack in response to an interrupt (see Figure C-2). All we must do, therefore, is load the status register from the stack, set the INTERRUPT DISABLE flag (bit 2), and store the result back in the stack.

```
PLA                 ;GET STATUS REGISTER
ORA  #%00000100     ;DISABLE MASKABLE INTERRUPT
PHA                 ;RETURN STATUS REG TO STACK
```

Changing the return address is more difficult. If we save the accumulator and index register X in memory locations TEMPA and TEMPX, respectively, we can use TSX to move the stack pointer to index register X. We can then access the return address by indexing on page 1 with register X, remembering that the stack pointer contains the address of the next available (empty) stack location. The indexed addresses for the items in the stack are

Status register:	$0101,X
LSBs of return address:	$0102,X
MSBs of return address:	$0103,X

The following program will, for example, replace the return address with 0240, thus providing a special error exit.

```
STA     TEMPA       ;SAVE A AND X
STX     TEMPX
TSX
LDA     #$40        ;PUT ERROR EXIT IN STACK
STA     $0102,X
LDA     #$02
STA     $0103,X
LDX     TEMPX       :RESTORE A AND X
LDX     TEMPA
```

PROBLEM C-25

Revise the latest program so that it reduces the return address by 2, thus allowing the replacement of a BRK instruction. How would you adjust the return address to allow for the replacement of a JSR instruction? That is, assume that a debugger has replaced the actual code with JSR BRKPT, where BRKPT is a service address in the monitor. Remember that JSR (unlike BRK) does not save the status register in the stack.

PROBLEM C-26

Revise Program C-5 so that the main program simply executes an endless loop instruction (jumping to itself). The interrupt service routine should examine the input data and return to the endless loop if the value is not 7F (hex). If the value is 7F hex, the service routine should return control to the instruction following the endless loop. This is another approach to waiting for an initial synchronization character. The program should wait in place (servicing but essentially ignoring interrupts) until the service routine receives an input of 7F (hex) from the switches.

MULTIPLE SOURCES OF INTERRUPTS

So far, we have always assumed a single source of interrupts. In real applications there are normally several sources. At the very least, an interrupt-driven system will have both input and output devices producing interrupts. Other common sources include alarms, timers, control panels, and remote stations. The problem is how to determine which source caused the interrupt. Once that has been done, the processor must execute the appropriate service routine.

One approach is to attach each source to its own interrupt input. Each interrupt input causes a transfer to a particular memory address at which the service routine for that source can begin. This is called a

vectored interrupt system, since each source directs or *vectors* the processor to the appropriate service routine. In the case of the 6502 microprocessor, we could tie one source to $\overline{\text{IRQ}}$ (the maskable interrupt) and one to $\overline{\text{NMI}}$ (the nonmaskable interrupt). Obviously, this approach suffices only when the microprocessor has enough interrupt inputs to handle all the sources.

If there are more sources than inputs, the simplest and least expensive approach is to examine the status of each source separately. This is analogous to answering a telephone that is connected to several different lines by trying one line at a time. The first source found to be active is serviced and the others are handled in the order of examination. This approach (called *polling*) is particularly simple for PIAs since the status of each source is readily available in a PIA control register. The next program waits for an interrupt on either control line CA1 or CB1 of the PIA. The service routine examines the control registers and services the first interrupt that it finds active. If the input (CA1) interrupt is active, the service routine increments memory location 0040 and loads the input data into memory location 0041. If the output (CB1) interrupt is active, the service routine increments memory location 0042, sends the output data from memory location 0043, and clears the interrupt flag.

Note that the main service routine disables further PIA interrupts by clearing bit 0 of both control registers. If this is not done, you will not be able to observe how priority affects the servicing of the interrupts because the second interrupt will be serviced as soon as the first service routine is completed. With the disabling included, the lower-priority interrupt will not be serviced and its interrupt flag will remain set. You can check the interrupt flags by examining the PIA control registers but be careful not to clear the flags by examining the data registers. After all, examining the data registers surely must involve a read operation.

```
            *=0280                ;POLLING ROUTINE
            DEC     PAC           ;DISABLE FURTHER INTERRUPTS
            DEC     PBC           ;BY CLEARING INTERRUPT
                                  ;ENABLE ON BOTH PORTS
            BIT     PAC           ;INPUT (CA1) INTERRUPT ACTIVE?
            BMI     SRVIN         ;YES, GO SERVICE IT
            BIT     PBC           ;OUTPUT (CB1) INTERRUPT ACTIVE?
            BMI     SRVOUT        ;YES, GO SERVICE IT
            RTI                   ;RETURN IF NEITHER ACTIVE

            *=$02C0               ;INPUT (CA1) INTERRUPT SERVICE
SRVIN       PHA                   ;SAVE ACCUMULATOR
            INC     $40           ;SET INPUT READY FLAG
            LDA     PAD           ;FETCH DATA FROM INPUT PORT
```

```
         STA     $41     ;SAVE DATA
         PLA             ;RESTORE ACCUMULATOR
         RTI

         *=$02E0         ;OUTPUT (CB1) INTERRUPT SERVICE
SRVOUT   PHA             ;SAVE ACCUMULATOR
         INC     $42     ;SET OUTPUT READY FLAG
         LDA     $43     ;GET OUTPUT DATA
         EOR     #$FF    ;INVERT POLARITY
         STA     PBD     ;SEND DATA TO LEDS
         BIT     PBD     ;CLEAR INTERRUPT FLAG
         PLA             ;RESTORE ACCUMULATOR
         RTI
```

PROBLEM C-27

If an input interrupt and an output interrupt occur simultaneously, which one will the processor service? Why? How could you change the polling routine to invert the priority?

PROBLEM C-28

Some interrupt systems may ignore low-priority interrupts for a long time if there are many high-priority interrupts. One way to ensure that all interrupts get serviced is to rotate the priorities. Make the polling routine invert the order in which it examines the PIA control registers each time it is executed. Use memory location 0044 as a flag that indicates the current order of examination (00 means "examine control register A first" and FF means "examine control register B first").

PROBLEM C-29

Write a program for a complete interrupt-driven I/O system that initially enables only the input interrupt, waits for input data, disables the input interrupt and enables the output interrupt on receipt of data, waits for the output interrupt, and finally sends the input data, disables the output interrupt, and enables the input interrupt when the output interrupt occurs. Remember that the PIA latches transitions that occur while its interrupt outputs are disabled.

Be careful that you do not service a port where the interrupts have been disabled. Even if the interrupts from one port of a PIA have been disabled, an input on its control lines will still set an interrupt flag. The port will not cause an interrupt, but a polling routine that examines its control register will find an interrupt flag set. Thus, if you disable some PIA interrupts, you should check only the interrupt flags on the ports that are currently enabled. You can determine if a port is enabled by checking the enabling bits in the control register (bit 0 for control line 1, bit 3 for control line 2).

Polling is an adequate method for identifying interrupts as long as the number of sources is small and the required response time is long. Note that the only real difference between normal polling of PIAs and a polling interrupt system is that the latter is activated by an interrupt. As the number of inputs increases, polling becomes slow and cumbersome. If all sources are equally likely to produce an interrupt, then half of them will have to be polled on the average and the time required to identify the source will increase linearly with the number of sources. Therefore, polling cannot handle large interrupt systems. The alternative is to add external hardware in order to create a fully vectored system. For example, an encoder like the one described in Laboratory 4 could produce a numerical value that the processor could read from an input port. The processor could then use that value to determine which service routine to execute.

GUIDELINES FOR PROGRAMMING WITH INTERRUPTS

In writing programs for interrupt-driven systems, the programmer should use the following guidelines:

1) Initialize all parameters before enabling interrupts. In particular, the stack pointer must be loaded since the interrupt response utilizes the stack.

2) Make all interrupt service routines transparent to the programs that they can interrupt. This means that service routines should not change any registers or flags (including interrupt masks and enables) unless such changes are essential and clearly understood. We have described some of the special conditions under which such changes are occasionally made.

3) Provide a well-defined method for communicating between the main program and the interrupt service routines. This method should be flexible and should not depend on special characteristics of the main program or the service routines.

There are many aspects of programming with interrupts that we have not discussed. Among these are the use of reentrant programs that can be interrupted and resumed later even if the same programs have been executed as part of the interrupt service. A reentrant program must use the registers and the stack for temporary storage, not specific memory addresses, since values stored in those addresses would be destroyed. Subroutines that are not reentrant cannot be called by interrupt service routines unless they are executed with the interrupts disabled.

We can deal with subroutines that must run with interrupts disabled in the same way that we dealt with subroutines that require a particular value of the DECIMAL MODE flag (see Laboratory A). That is, the subroutine must save the current value of the INTERRUPT DISABLE flag in

the stack (using PHP), disable the interrupts (using SEI), execute the code that can run only with interrupts disabled, and restore the original value of the INTERRUPT DISABLE flag (using PLP). Grappel has discussed this problem in the article cited in the references at the beginning of the Laboratory. This procedure ensures that the routine executes correctly without either disabling interrupts when they were originally enabled or enabling them when they were originally disabled.

Other issues that we have not discussed include:

1) When to use the nonmaskable interrupt. This input is most commonly used as a power fail interrupt that causes the CPU to save essential data in a backup memory. Such an interrupt should be nonmaskable since loss of power will shut down all activities anyway.

2) When to enable and disable interrupts. Interrupts must be disabled during activities that could not be resumed properly, such as delay loops, command sequences, and updating of multiple-word results that must be used during the interrupt service. If a program is changing data that occupies more than one word, it must complete the task if the interrupt service routine uses the data. Otherwise, the service routine could find the data only partially changed and interpret it incorrectly.

3) How to implement interrupts from sources other than PIAs. Many other devices can be handled much like PIAs, although the specific details depend on the particular device. Other common sources of interrupts include serial interfaces, timers, converters, arithmetic chips, and peripheral controllers.

KEY POINT SUMMARY

1) Interrupts provide a convenient way for a computer to respond to external events such as changes in the status of peripherals, alarms, requests for control or information, or the passage of time. The program does not have to check to see if events have occurred, since the occurrences cause changes in hardware inputs to the CPU. Interrupts provide fast response and simple logic but introduce a random element into programs that makes them difficult to debug and test.

2) The 6502 microprocessor has two interrupt inputs, one that is maskable ($\overline{\text{IRQ}}$) and one that is nonmaskable ($\overline{\text{NMI}}$). In response to these inputs, the processor saves the program counter and status register in the stack, disables the maskable interrupt, and fetches a new value for the program counter from a specified pair of memory locations. An RTI instruction at the end of the service routine restores the old program counter and status register from the stack. The service routine must save and restore the accumulator and index registers explicitly if it uses them.

3) Before interrupts are enabled, the main program must load the stack pointer and initialize any parameters that the service routines use. It must also determine the operating modes for PIAs and other I/O devices. Since RESET disables the CPU interrupt (and the PIA interrupts), startup programs can perform the required initialization without interference as long as $\overline{\text{NMI}}$ is not involved directly.

4) The main program and the interrupt service routines cannot communicate through the registers, because each generally needs the registers for its own purposes. A simple way to communicate is through assigned memory locations, which act like a mailbox. Either program can place information in those assigned memory locations to be picked up by the other program.

5) Buffering reduces how often the main program must communicate with an interrupt service routine. The main program need only concern itself with the handling of an entire buffer's worth of data. Either a large buffer or multiple buffers (so-called double buffering) gives the main program more time to perform its own work without losing data or ignoring requests for service.

6) PIAs can be used in an interrupt-driven mode by setting the interrupt enable bits in the control register. Transitions on the control lines then cause interrupts as well as setting the interrupt flags.

7) Interrupt service routines sometimes must change the register values that have been saved in the stack. The common reasons are to disable interrupts in the main program, adjust the return address (to account for a breakpoint), and provide special exits for error or exception handling.

8) If there is more than one source of interrupts, the program must have some way of differentiating among them. Vectoring means that each source provides a means of identification, either by being attached to a separate interrupt input or by producing a data value that the processor can examine. Polling means that the processor must examine the status of each source separately until it finds one that is active.

9) In polling interrupt systems, the priority of the sources depends on the order in which they are examined. This order can be changed or varied if necessary. However, the time required to identify a source increases linearly with the number of sources, so polling is reasonable only if the number of sources is small.

Laboratory D

Timing Methods

PURPOSE

To learn how to provide timing for input and output operations.

PARTS REQUIRED

- A low-frequency clock input (5 to 200 Hz). One way to produce this input is to divide down the 1-MHz clock that is available on pin U of the Expansion Connector. A clock obtained from a signal generator or from a 555 timer chip (see Figure D-1) will also be satisfactory. The clock should be tied with a jumper wire to pin PA5 of the user 6530 device (pin 5 of the Application Connector) as shown in Figure D-2. Jumper wires will allow you to choose between the clock input and the switch input shown in Figure 2-1.
- A connection between pin PB7 of the user 6530 device (pin 15 of the Application Connector) and the microprocessor's $\overline{\text{IRQ}}$ input (pin 4 of the Expansion connector) as shown in Figure D-3.

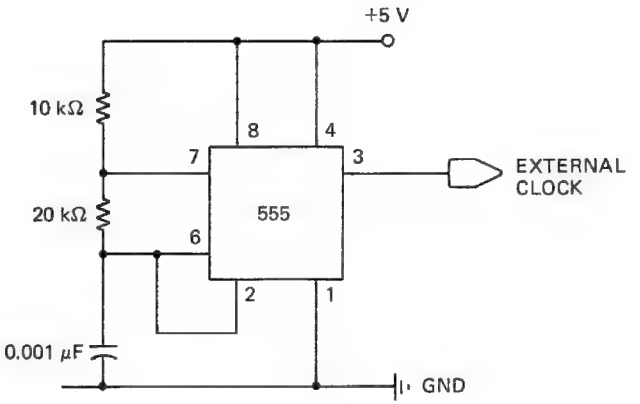


FIGURE D-1. Simple circuit for generating a low-frequency clock from a 555 timer chip. The frequency is approximately 83 Hz.

FIGURE D-2. Connection of the external clock to bit 5 of port A of the user 6530 device. (**Note:** Jumper wires can be used to select either this input or the switch input shown in Figure 2-1.)

FIGURE D-3. Connection of pin PB7 of the user 6530 device (the timer interrupt request) to the microprocessor's $\overline{\text{IRQ}}$ input. (**Note:** Jumper wires can be used to select either this input or the LED output shown in Figure 3-1.)

REFERENCE MATERIALS

G. Cole, "Time Simultaneous Events with a Microprocessor," *EDN*, November 20, 1980, pp. 99-100.

R. H. Cushman, "Successful Systems Combine User Needs and Microcomputer Technology," *EDN*, April 20, 1977, pp. 104-111.

M. L. DeJong, *Programming and Interfacing the 6502*, Howard W. Sams, Indianapolis, IN, 1980, pp. 111-114, 210-215, 326-349.

J. Hemenway, "EDN Microcomputer Operating Systems Directory," *EDN*, November 5, 1980, pp. 275-337.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 343-345, 485-486.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11-8- through 11-11, 11-36- through 11-42, 12-23 through 12-31.

A. Osborne et al., *An Introduction to Microcomputers, Vol. 2: Some Real Microprocessors,* Osborne/McGraw-Hill, Berkeley, CA, 1978, pp. 9-78 through 9-106 (6840 programmable timer), 10-34 through 10-53 (6522 parallel interface/timer), 10-54 through 10-59 (6530 peripheral interface/memory device).

D. L. Ripps, "Help a Real-Time Multitasking Operating System," *Electronic Design,* June 21, 1979, pp. 86-91; continued September 13, 1979, pp. 146-151, and September 27, 1979, pp. 82-86 (description of Industrial Programming's MultiTasking Operating System).

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 83-91, 207-217.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 43-46 (counters), 88-89 (process control), 368-375 (timing loops).

W. J. Weller, *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980, Chapter 18.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Appendix H.

The Linear and Interface Circuits Data Book, Texas Instruments, Inc., Dallas, TX, 1973, pp. 7-53 through 7-61 (555 timer).

MCS6500 Microcomputer Family Hardware Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 71-83.

MCS6500 Microcomputer Family Programming Manual, Commodore/MOS Technology, Norristown, PA, 1976, p. 166.

WHAT YOU SHOULD LEARN

1) The alternative ways to handle timing.
2) How to synchronize with an external clock.
3) How to determine the period of an external clock.
4) How to program the timer in the 6530 peripheral interface/memory device.
5) How to produce and use an elapsed time interrupt.
6) What a real-time clock is.
7) How to produce and use a simple real-time clock.
8) How to schedule tasks with the real-time clock.
9) How to keep calendar time with the real-time clock.
10) What a real-time operating system does.

TERMS

Dead time—a delay between operations required to prevent errors caused by overlap.

Multitasking—executing many tasks during a single period of time, usually by working on each one for a specified part of the period and suspending tasks that must wait for input, output, the completion of other tasks, or external events.

One-shot—a device that produces a single pulse of known length in response to a pulse input. Also called a *monostable multivibrator.*

Operating system (OS)—a computer program that controls the overall operations of a computer and performs such functions as assigning places in memory to programs and data, scheduling the execution of programs, processing interrupts, and controlling the overall input/output system. Also known as a monitor, executive, or master-control program, although the term *monitor* is usually reserved for a simple operating system with limited functions.

Prescaler—a counter that is used to divide the input frequency to another counter. The prescaler thus reduces the rate at which the other counter operates by a factor known as the *divide ratio.*

Programmable timer—a device that can handle a variety of timing tasks, including the generation of delays, under program control.

Real-time—in synchronization with the actual occurrence of events.

Real-time clock—a device that interrupts a CPU at regular time intervals.

Real-time operating system—an operating system that can act as a supervisor for programs that have real-time requirements. May also be referred to as a *real-time executive* or as a *real-time monitor.*

Scheduler—a program that determines when other programs should be started and terminated.

Supervisor—a program that controls the loading and execution of other programs and subroutines.

Suspend (a task)—halt execution and preserve the status of the task until some future time.

Task—a self-contained program that can serve as part of an overall system under the control of a supervisor.

Task status—the set of parameters that specify the current state of a task. A task can be suspended and resumed as long as its status is saved and restored.

Timeout—a period during which no activity is allowed to proceed, an inactive period.

Utility—a general-purpose program, usually supplied by the computer manufacturer or part of an operating system, that executes a common or standard operation such as sorting, converting data from one format to another, or copying a file.

TIMING REQUIREMENTS AND METHODS

Timing is a continual problem in microprocessor applications. Systems must handle inputs and outputs at the proper rate and must derive timing information from external clocks. Delay programs, such as those described in Laboratory 3, can handle simple timing requirements but they occupy the processor completely and are inadequate for applications with complex and varying timing needs.

Many applications, particularly in process and industrial control, involve real-time constraints; the systems must handle certain inputs and outputs at times that are determined externally. Some applications, such as navigation systems and security systems, may even need to maintain calendar time.

We will explore the following methods of handling timing:

1) Varying the parameters of delay routines.
2) Measuring the periods of external clocks and adapting to their frequencies.
3) Using a programmable timer.
4) Using a real-time clock.

The aims of these methods are to provide more flexibility than fixed delay routines while occupying the processor as little as possible.

GENERALIZED DELAY ROUTINES

The simplest way to generalize a delay routine is to have a parameter determine how long it runs. The monitor subroutine DELAY (starting address 1ED4) counts down from the value in memory locations 17F2 and 17F3 (MSBs in 17F3). It uses memory location 17F4 for temporary storage. The routine is

```
DELAY   LDA   $17F3      ;PLACE COUNT IN 17F4 AND A
        STA   $17F4
```

```
        LDA   $17F2
DE2     SEC                ;SUBTRACT 1 FROM LSB OF COUNT
        SBC   #$01
        BCS   DE3          ;IS THERE A BORROW FROM MSB?
        DEC   $17F4        ;YES, SUBTRACT 1 FROM MSB
DE3     LDY   $17F4        ;IS RESULT NEGATIVE?
        BPL   DE2          ;NO, CONTINUE COUNT
        RTS
```

DELAY does not change memory location 17F2 or 17F3, but it does change 17F4.

The total time required by DELAY is (in clock cycles):

- 24 for a JSR (6), RTS (6), and the initial loading of the accumulator and memory location 17F4 (the first three instructions, each of which takes four cycles).
- 14 for each iteration that does not decrement memory location 17F4 (i.e., no borrow from the more significant byte). This path is made up of SEC (2), SBC #$01 (2), BCS involving a branch (3), LDY $17F4 (4), and BPL involving a branch (3).
- 19 for each iteration that decrements memory location 17F4. This path differs from the previous path by including DEC $17F4 (6) and eliminating the branch from BCS (−1). The final (exiting) iteration takes one cycle less since BPL does not cause a branch.

So the time spent (including a JSR instruction and subtracting 1 for the final iteration) is 23 + 14 * COUNT + 5 * (CNTH30 + 1). Here CNTH30 is the initial value of memory location 17F3 and COUNT depends on the initial values of memory locations 17F2 and 17F3. Since the subroutine decrements the accumulator and memory location 17F4 until both become negative, COUNT is equal to CNTH30*256 + CNTL30 + 1, where CNTL30 is the initial value of memory location 17F2. Thus an initial value of 0000 results in 1 iteration, 0007 8 iterations, 0100 257 iterations, and so on. Table D-1 lists some time intervals you can obtain from DELAY, with the length in milliseconds assuming a clock frequency of 1 MHz. You may find it instructive to derive some of these results on your own. Since the program exits if bit 7 of memory location 17F4 is 1, the maximum count is 8000 (hex); any larger count (in the unsigned sense) will cause an exit after the first iteration. (Why?)

Table D-1

TIME INTERVALS FROM SUBROUTINE DELAY (STARTING ADDRESS 1ED4)

TIME INTERVAL [Ms (1-MHz CLOCK)]	INITIAL COUNT (HEX)
1	0044
2	008C
3	00D3
5	0162
10	02C6
20	0590
50	0DEB
100	1BDA
250	45A6
459	8000

The following program produces a delay lasting according to the contents of memory locations 17F2 and 17F3 (MSBs in 17F3). Program D-1 is the hexadecimal version.

```
LDX     #$7F        ;INITIALIZE USER STACK POINTER
TXS
JSR     DELAY       ;WAIT A WHILE
BRK
```

PROGRAM D-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	20	JSR	DELAY
0204	D4		
0205	1E		
0206	00	BRK	

Run Program D-1 for different starting values in 17F2 and 17F3. What is the smallest value for which you can see the KIM displays go on

and off? Be careful of the fact that resetting the KIM changes memory location 17F3 to FF.

PROBLEM D-1

Write a subroutine that uses DELAY to produce a wait of 100 ms without affecting any registers or flags. Save the status register, accumulator, and index register Y in the stack.

PROBLEM D-2

Write a subroutine that uses DELAY to produce a wait lasting the number of milliseconds in the accumulator.

Example:

(A) = 38 (hex)

Result:

The routine should produce a wait of 38 hex (56 decimal) ms.

PROBLEM D-3

Write a subroutine that uses DELAY to produce a wait lasting the number of seconds in the accumulator.

Example:

(A) = 03

Result:

The routine should produce a wait of 3 s.

A generalized delay routine saves memory and programming time, as well as simplifying documentation. However, it still occupies the processor completely. This is acceptable if the system performs tasks sequentially without interruption and does not operate in real time. Software delay routines (or *timeouts*) are often used to handle the initial response time of slow mechanical devices such as printers or motors.

WAITING FOR A CLOCK TRANSITION

We have not yet discussed how to start time intervals and determine how long they should last. Many systems use either hardware (such as synchronizing circuits and one-shots or timer chips) or parameter values stored in ROM to determine both the starting procedures and the interval lengths.

This approach requires very little programming and provides compatibility with systems that are not computer-based, but it lacks flexibility. The resulting systems cannot be modified easily to handle peripherals that require different interfaces or operate at different data rates. This inflexibility reduces the size of the potential markets for the systems, and makes upgrading difficult. The need for easy upgrading has become particularly important in recent years because of rapid improvements in the speed and capabilities of peripherals. A system that cannot be upgraded as peripheral technology changes soon becomes outdated and uncompetitive.

An alternative approach is to have the program determine the time constants required to handle a particular set of I/O devices. That is, the system adapts to its environment. For example, the system could establish synchronization by examining a clock input. Attach a low-frequency (5- to 200-Hz) clock source to bit 5 of port A of the user 6530 device. The following program waits for a low-to-high transition on the clock line.

```
WAITL   LDA   $1700          ;IS CLOCK LINE LOW?
        AND   #%00100000
        BNE   WAITL           ;NO, WAIT
WAITH   LDA   $1700          ;IS CLOCK LINE HIGH?
        AND   #%00100000
        BEQ   WAITH           ;NO, WAIT
        BRK
```

Program D-2 is the hexadecimal version. Note that the program first waits for the clock line to go low and then waits for it to go high (see Figure D-4 for a flowchart).

PROGRAM D-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	AD	WAITL	LDA	$1700
0201	00			
0202	17			
0203	29		AND	#%00100000
0204	20			
0205	D0		BNE	WAITL
0206	F9			
0207	AD	WAITH	LDA	$1700
0208	00			
0209	17			
020A	29		AND	#%00100000

PROGRAM D-2 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
020B	20		
020C	F0	BEQ	WAITH
020D	F9		
020E	00	BRK	

Enter and run Program D-2. Vary the clock rate. How would you make the program wait for a high-to-low transition rather than a low-to-high transition? You may want to test this program and the answers to the next few problems using a debounced switch (like the ones shown in Figure B-4) as the clock input. You will then have complete control over the clock for testing purposes.

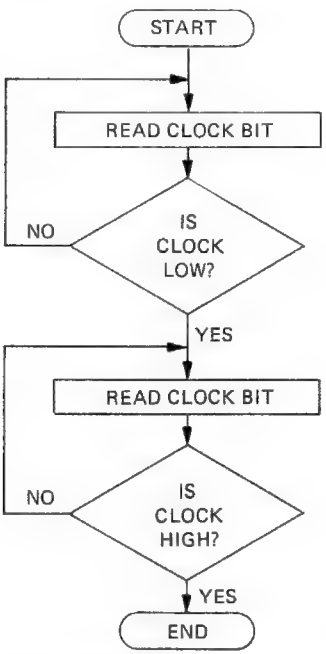


FIGURE D-4. Flowchart of clock synchronization program.

PROBLEM D-4

Change Program D-2 to use BIT instead of AND. Be careful; BIT does not allow immediate addressing, so you will have to load the accumulator with the mask

instead of the data from the switches. Remember the special effects of BIT on the NEGATIVE and OVERFLOW flags. Which bit position (for the clock input) makes the revision of Program D-2 the shortest? Write the programs (using BIT) that handle a clock input attached to bit 6 and to bit 7.

PROBLEM D-5

Make Program D-2 wait for the first full clock pulse (i.e., it should wait for a low-to-high transition followed by a high-to-low transition).

PROBLEM D-6

Make Program D-2 wait for 10 low-to-high transitions.

MEASURING THE CLOCK PERIOD

We can extend Program D-2 to have the processor measure the length of the clock period. This involves:

1) Waiting for a transition.
2) Counting time intervals until the next transition.

Obviously, the period must be many CPU clock cycles in length for this method to be accurate.

The following program (see Figure D-5 for a flowchart) waits for a low-to-high clock transition and then counts the number of milliseconds that elapse until the next such transition:

```
        LDY   #0          ;CLOCK COUNT = ZERO
WTL1    LDA   $1700       ;IS CLOCK LINE LOW?
        AND   #%00100000
        BNE   WTL1        ;NO, WAIT
WTH1    LDA   $1700       ;IS CLOCK LINE HIGH?
        AND   #%00100000
        BEQ   WTH1        ;NO, WAIT
WTL2    INY               ;INCREMENT CLOCK COUNT
        LDX   #$C8        ;WAIT 1 MS
DLYL    DEX
        BNE   DLYL
        LDA   $1700       ;IS CLOCK LINE LOW?
        AND   #%00100000
        BNE   WTL2        ;NO, WAIT
WTH2    INY               ;INCREMENT CLOCK COUNT
        LDX   #$C8        ;WAIT 1 MS
DLYH    DEX
        BNE   DLYH
```

```
LDA    $1700          ;IS CLOCK LINE HIGH?
AND    #%00100000
BEQ    WTH2           ;NO, WAIT
STY    $40            ;SAVE CLOCK COUNT
BRK
```

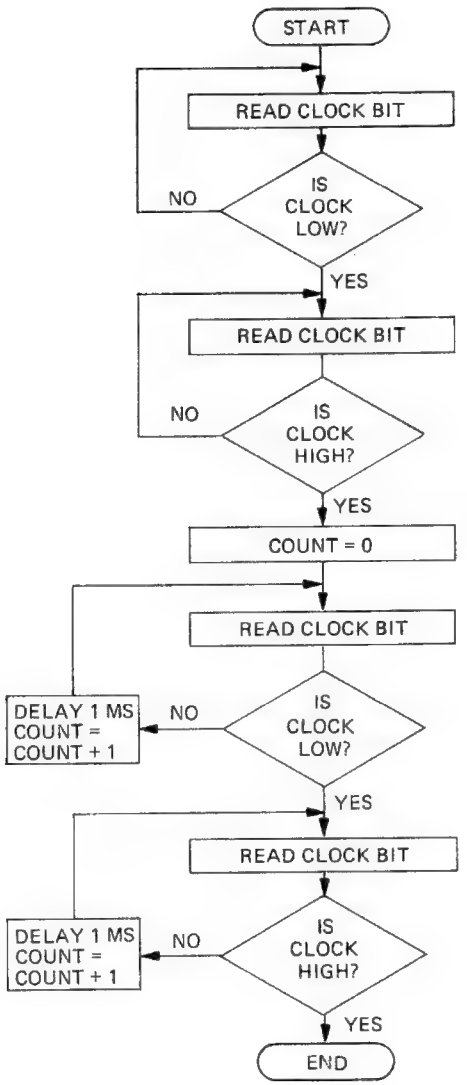


FIGURE D-5. Flowchart of the program that measures a clock period.

Program D-3 is the hexadecimal version; enter it into memory and run it. Check how accurately it measures some low-frequency clock inputs.

PROGRAM D-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A0		LDY	#0
0201	00			
0202	AD	WTL1	LDA	$1700
0203	00			
0204	17			
0205	29		AND	#%00100000
0206	20			
0207	D0		BNE	WTL1
0208	F9			
0209	AD	WTH1	LDA	$1700
020A	00			
020B	17			
020C	29		AND	#%00100000
020D	20			
020E	F0		BEQ	WTH1
020F	F9			
0210	C8	WTL2	INY	
0211	A2		LDX	#$C8
0212	C8			
0213	CA	DLYL	DEX	
0214	D0		BNE	DLYL
0215	FD			
0216	AD		LDA	$1700
0217	00			
0218	17			
0219	29		AND	#%00100000
021A	20			
021B	D0		BNE	WTL2
021C	F3			
021D	C8	WTH2	INY	
021E	A2		LDX	#$C8
021F	C8			
0220	CA	DLYH	DEX	
0221	D0		BNE	DLYH
0222	FD			
0223	AD		LDA	$1700
0224	00			
0225	17			
0226	29		AND	#%00100000
0227	20			

PROGRAM D-3 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0228	F0	BEQ	WTH2
0229	F3		
022A	84	STY	$40
022B	40		
022C	00	BRK	

PROBLEM D-7

Make Program D-3 measure the width of the high phase of the clock.

PROBLEM D-8

The ability of Program D-3 to measure clock periods depends both on how accurately it generates time intervals and on how fine its resolution is. What is the actual time between samples for the section of Program D-3 that measures the clock period? How could you make the interval more accurate? Change the program so that its resolution is 100 μs instead of 1 ms. Measure the period of your clock using both resolutions.

The processor can use the clock period it has measured to time input and output operations. This approach allows a program to handle I/O devices operating at different rates as long as it can determine what rate is being used. A common application of this approach is in handling serial I/O with a terminal, since terminals may operate at any of a common set of data rates (10 or 30 characters per second for low-speed units, 1200 to 19,200 bits per second for higher-speed units). A system that measures the bit rate of the terminal can operate at any of the standard rates without hardware modification.

The KIM monitor uses exactly this approach to determine the data rate of a terminal. All the user must do is connect the terminal, reset the KIM, and type a RUBOUT (DELETE) character. The KIM measures the amount of time between bits for the terminal and stores the appropriate values in memory locations 17F2 and 17F3. Later it can use subroutine DELAY to wait for 1 bit time between serial inputs or outputs. We will discuss serial I/O in greater detail in Laboratory E.

PROGRAMMABLE TIMERS

The previous methods still depend on the processor generating time intervals with delay routines. An alternative approach is to use a hardware

timer under computer control. The processor then only has to determine how the timer will operate, start it, and wait for it to signify the end of the time interval with an interrupt or a change in its status.

The simplest hardware timer is a one-shot (or monostable multivibrator) that produces a single pulse of fixed length in response to a pulsed input. More complex timers contain counters and latches; input controls may determine how many stages are used. Programmable timers are the timing equivalent of the programmable input/output ports that we discussed in Laboratory B. These devices have a variety of operating modes; the program selects a particular operating mode for a timer by clearing or setting bits in one or more control registers. The current state of the timer can be determined by examining the contents of its status registers. Typical options for a programmable timer are binary or decimal (BCD) counts, the shape of output pulses (e.g., square wave or brief pulse on terminal count), whether an interrupt is produced, whether a prescaler is used to divide the clock, and whether the device performs a single timing task or operates continuously (i.e., reloading its counters with their initial values after counting them down to zero).

Among the programmable timers that we can use readily with the 6502 microprocessor are:

1) The 6840 device intended for use with 6800-based microcomputers.
2) The dual timers in the 6522 Versatile Interface Adapter.
3) The timer in the 6530 Peripheral Interface/Memory device.

We will describe the 6530 timer in detail since the KIM has an onboard 6530 device. The references at the start of this laboratory discuss the other timers. Like programmable interfaces, programmable timers simplify hardware design, save parts, and allow the development and use of a standard series of circuit boards in many applications. On the other hand, programmable timers are relatively expensive and difficult to use and document because of their arbitrary features and unique programming requirements.

6530 INTERVAL TIMER

The 6530 Peripheral Interface/Memory device contains an 8-bit timer with a prescaler. A program can determine the starting count by storing it in the timer; the address in which the count is stored determines the ratio by which the prescaler divides the input clock and whether the ending of the time interval (i.e., the count reaching zero) causes an interrupt. The un-

usual feature here is that the 6530 device uses some of its address inputs to set the divide ratio, and enable or disable the timer interrupt.

Table D-2

ADDRESSES FOR THE TIMER IN THE USER 6530 DEVICE

ADDRESS IN WHICH DATA IS STORED	DIVIDE RATIO	INTERRUPT STATUS
1704	1	Disabled
1705	8	Disabled
1706	64	Disabled
1707	1024	Disabled
170C	1	Enabled
170D	8	Enabled
170E	64	Enabled
170F	1024	Enabled

Table D-2 lists the addresses for the timer in the KIM's user 6530 device. Storing data in any of these addresses sets the timer's divide ratio (to 1, 8, 64, or 1024) and enables or disables the timer interrupt. The on-board timer operates as follows:

1) As soon as the starting count is stored in the timer, counting begins at the specified rate. The clock input is just the CPU clock (1 MHz).

2) The status of the timer is available as bit 7 of memory address 1707. That bit (the timer interrupt flag) is cleared when the timer is read or written and is set when the timer counts past zero.

3) When the timer counts past zero, it sets the divide ratio to 1 automatically and starts the count at FF. The user can thus examine the count and determine how many clock cycles have elapsed since the timer reached zero (the number is FF minus the current count).

4) If the timer has not counted past zero, reading address 1706 (170E) will provide the current count and disable (enable) the interrupt.

5) If the timer has counted past zero, reading address 1706 will restore the divide ratio to its previously programmed value and leave the timer with its current count (not the count originally written into it).

Thus the following program produces a delay lasting 6400 clock cycles by storing 100 in the timer at an address that sets the divide ratio to 64 and disables the interrupt. Program D-4 is the hexadecimal version.

```
LDA   #100      ;SET STARTING COUNT TO 100 DECIMAL
STA   $1706     ;AND DIVIDE RATIO TO 64
```

```
WTTIM   LDA   $1707    ;IS INTERVAL OVER?
        BPL   WTTIM     ;NO, WAIT
        BRK             ;YES, DONE
```

PROGRAM D-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A9		LDA	#100
0201	64			
0202	8D		STA	$1706
0203	06			
0204	17			
0205	AD	WTTIM	LDA	$1707
0206	07			
0207	17			
0208	10		BPL	WTTIM
0209	FB			
020A	00		BRK	

PROBLEM D-9

Change Program D-4 to make the initial count 150 and the divide ratio 1024. How long is the delay now?

PROBLEM D-10

Extend Program D-4 so that it reads the final value from the counter and stores that value in memory location 0040 before exiting. What does this final value mean? Remember that the divide ratio is set to 1 automatically when the timer counts past zero.

PROBLEM D-11

Extend Program D-4 so that it waits for 10 intervals given by a starting count of 100 and a divide ratio of 64. How would you make the program obtain the starting count from memory location 0040 and the number of intervals from memory location 0041.

Sample Case:

(0040) = C0

(0041) = 07

Result:

The program waits for seven intervals given by a starting count of C0 hex and a divide ratio of 64.

How could you make your program obtain a code for the divide ratio from memory location 0042? Assume that the code is 3 or less and is related to the divide ratio as follows:

CODE	DIVIDE RATIO
0	1
1	8
2	64
3	256

Sample Case:

(0040) = C8

(0041) = 05

(0042) = 03

Result:

The program waits for five intervals given by a starting count of C8 hex and a divide ratio of 256.

ELAPSED TIME INTERRUPTS

Obviously, we have not yet solved the problem of utilizing the processor efficiently. Program D-4 still forces the processor to wait idly for the time interval to end. However, we can eliminate that requirement by having the timer produce an interrupt at the end of the interval. Now the processor can perform other tasks and need not even check the status of the timer.

To produce interrupts from the 6530's timer, we must connect pin PB7 (pin 15 of the Application Connector) to the microprocessor's $\overline{\text{IRQ}}$ input (pin 4 of the Expansion Connector) as shown in Figure D-3. The timer uses PB7 automatically as its interrupt request line, but the programmer must make that line an input (*not* an output) by clearing bit 7 of the data direction register. The reason why we must make this line an input when the 6530 is using it as an output is beyond our comprehension.

Table D-2 contains the addresses into which we must write the starting count to operate in the interrupt-driven mode. After the timer produces an interrupt, we must clear its interrupt flag by either reading or writing one of the timer addresses.

The following program produces an interrupt after 10 ms. We obtained the value 9C by calculating the count required at an input frequency of 1 MHz and a divide ratio of 64 to make the total period last 10 ms (10,000 clock cycles):

$$X = \frac{1 \cdot 10^4}{64}$$

$$X = 156 \text{ (decimal)} = 9C \text{ (hex)}$$

Program D-5 is the hexadecimal version.

PROGRAM D-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	8E		STX	$1703
0204	03			
0205	17			
0206	A9		LDA	#$9C
0207	9C			
0208	8D		STA	$170E
0209	0E			
020A	17			
020B	A9		LDA	#0
020C	00			
020D	85		STA	$40
020E	40			
020F	58		CLI	
0210	A5	WTTIM	LDA	$40
0211	40			
0212	F0		BEQ	WTTIM
0213	FC			
0214	78		SEI	
0215	20		JSR	$1C05
0216	05			
0217	1C			
0280	E6		INC	$40

PROGRAM D-5 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0281	40		
0282	2C	BIT	$1706
0283	06		
0284	17		
0285	40	RTI	

```
        LDX     #$7F        ;INITIALIZE USER STACK POINTER
        TXS
        STX     $1703       ;MAKE PORT B OUTPUT EXCEPT BIT 7
        LDA     #$9C        ;SET TIMER FOR 10 MS
        STA     $170E       ;START TIMER, ENABLING INTERRUPT
        LDA     #0          ;CLEAR READY FLAG
        STA     $40
        CLI                 ;ENABLE CPU INTERRUPT
WTTIM   LDA     $40         ;IS INTERVAL OVER?
        BEQ     WTTIM       ;NO, WAIT
        SEI                 ;YES, DISABLE CPU INTERRUPT
        JSR     $1C05

        *=0280              ;TIMER SERVICE ROUTINE
        INC     $40         ;SET READY FLAG
        BIT     $1706       ;CLEAR TIMER INTERRUPT FLAG
        RTI
```

The special features of this program are: 1) we made PB7 an input so that we could use it as a timer interrupt, 2) we enabled the timer interrupt by storing the initial count in memory address 170E rather than 1706, and 3) we cleared the timer interrupt flag by reading address 1706 (using BIT to avoid changing the accumulator). We concluded the program with JSR $1C05 instead of BRK, since we are using the $\overline{\text{IRQ}}$ input and therefore need memory addresses 17FE and 17FF for an interrupt vector (see Laboratory C). Remember to set (17FE) = 80 and (17FF) = 02 before executing Program D-5.

PROBLEM D-12

Change Program D-5 to make the time interval 5 ms instead of 10 ms. How would you make the interval 100 ms?

We can extend Program D-5 by letting the service routine reload the timer. The required additions are:

```
PHA                    ;SAVE ACCUMULATOR
LDA     #$9C           ;START TIMER AGAIN
STA     $170E
PLA                    ;RESTORE ACCUMULATOR
```

The hexadecimal version of the service routine is now

0280	48	PHA	
0281	E6	INC	$40
0282	40		
0283	A9	LDA	#$9C
0284	9C		
0285	8D	STA	$170E
0286	0E		
0287	17		
0288	68	PLA	
0289	40	RTI	

If your service routine reloads the timer, you should clear the final timer interrupt (the one that has been activated when your program exits) by hand. The easiest way to do this is to examine memory location 1706 after your program returns control to the KIM monitor. If you do not clear the final interrupt, it could interfere with later programs.

PROBLEM D-13

Use the extended version of Program D-5 to write a program that waits for two clock interrupts. How would you modify your program so that memory location 0030 specifies the number of clock interrupts to wait?

Example:

(0030) = 07 means that the program waits for seven clock interrupts.

PROBLEM D-14

Write a program that waits for the number of timer interrupts specified by memory location 0030 and then lights the LEDs attached to user 6530 port B for the number of timer interrupts specified by memory location 0031.

Example:

(0030) = 50

(0031) = 30

Result:

The program waits for 50 hex (80 decimal) timer interrupts and then lights the LEDs attached to port B for 30 hex (48 decimal) timer interrupts. Finally, it turns the LEDs off. Thus the LEDs are off for five tenths of a second and then on for three tenths of a second. Be careful of the fact that we can use only the 6 least significant bits of port B for data, since PB7 is reserved for the timer interrupt and PB6 is not connected.

REAL-TIME CLOCK

A real-time clock simply produces interrupts at regular intervals. The computer can keep time by counting how many interrupts have occurred. For example, we can have the extended version of Program D-5 (i.e., the version that reloads the timer after each interrupt) keep a count in memory location 0040 by placing an endless loop at the end of the main program. The hexadecimal changes are

0215	4C	HERE	JMP	HERE
0216	15			
0217	02			

Now memory location 0040 contains a count of how many hundredths of a second have elapsed. Run the program a few times and see what values you find in that location when you reset the computer.

Other programs can use the clock count to measure elapsed time, much as you use your watch for that purpose. If your watch now reads 2:33, for example, you can wait for 15 minutes by adding 15 to 2:33 and waiting for your watch to read 2:48. Similarly, a program can produce a delay by reading the clock count, adding the required number of clock periods, and waiting until the sum and the count are equal. The following instructions at the end of Program D-5 will make the computer wait for 50 ms (five clock periods). Here the initial count is zero so the addition is unnecessary.

```
        LDA   #5
WAIT5   CMP   $40        ;HAS CLOCK COUNT REACHED 5?
```

```
BNE     WAIT5       ;NO, WAIT
SEI                 ;YES, DISABLE CPU INTERRUPT
JSR     $1C05
```

The hexadecimal version of the changes is

0215	A9		LDA	#5
0216	05			
0217	C5	WAIT5	CMP	$40
0218	40			
0219	D0		BNE	WAIT5
021A	FC			
021B	78		SEI	
021C	20		JSR	$1C05
021D	05			
021E	1C			

Enter and run the modified program. Make it wait for 10 clock periods.

PROBLEM D-15

Make the modified program wait for five clock periods and then light all the LEDs attached to port B of the user 6530 device for 10 clock periods. Change the program to produce the following on-off periods:

1) OFF-10
   ON-5
2) OFF-1
   ON-1

PROBLEM D-16

Make the program from Problem D-15 operate continuously, turning the LEDs on and off according to the duty cycle given by the contents of memory locations 0030 and 0031. To calculate the count that marks the end of a time interval, add the length of the interval to the current count (memory location 0040). Does it matter if the addition produces a carry?

Run the program for the following test cases and describe what happens.

1) OFF-(0030) = 01
   ON-(0031) = 01
2) OFF-(0030) = 04
   ON-(0031) = 1C

3) OFF-(0030) = 10 (hex)
ON-(0031) = 10 (hex)

4) OFF-(0030) = 1C
ON-(0031) = 04

PROBLEM D-17

Extend the program from Problem D-16 so that it turns the LEDs on and off according to the following duty cycle for a single iteration:

OFF - 10 (hex)

ON - 20 (hex)

OFF - 40 (hex)

ON - 80 (hex)

Clearly, most industrial and process controllers involve complex duty cycles with numerous variations in length and amplitude.

To make the program shorter and more general, place the ON-OFF values in a table. For example, you could use memory locations 0380 through 0383 as follows:

(0380) = 10 (first OFF period)

(0381) = 20 (first ON period)

(0382) = 40 (second OFF period)

(0383) = 80 (second ON period)

You can use index register X to identify the current element and CPX to decide when the program should stop.

HANDLING LONGER TIME INTERVALS

We can handle longer time intervals by using more memory locations for the clock count. The following service routine uses memory locations 0040 and 0041 (MSBs in 0041). You must initialize both locations to zero, either by hand or with instructions in the main program, before execution.

```
        *=$0280
        PHA                     ;SAVE ACCUMULATOR
        INC     $40             ;INCREMENT LSB OF CLOCK COUNT
        BNE     STTIM
        INC     $41             ;AND MSB IF NECESSARY
STTIM   LDA     #$9C            ;START TIMER AGAIN
```

```
STA     $170E
PLA                     ;RESTORE ACCUMULATOR
RTI
```

RTI restores the status register automatically, but we must save and restore the accumulator. Enter and run this program, placing an endless loop at the end of the main program. Program D-6 is the hexadecimal version. Let it run for a while and see what values you find in memory locations 0040 and 0041.

PROGRAM D-6

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0280	48		PHA	
0281	E6		INC	$40
0282	40			
0283	D0		BNE	STTIM
0284	02			
0285	E6		INC	$41
0286	41			
0287	A9	STTIM	LDA	#$9C
0288	9C			
0289	8D		STA	$170E
028A	0E			
028B	17			
028C	68		PLA	
028D	40		RTI	

PROBLEM D-18

Write a main program that uses Program D-6 to wait for 300 (012C hex) clock periods before returning control to the monitor.

PROBLEM D-19

Write a main program that uses Program D-6 to turn all the LEDs attached to port B off for 300 (012C) clock periods and then on for 400 (0190) clock periods. Change your program to produce the following on-off periods.

1) OFF-400 (0190 hex)
   ON-300 (012C hex)
2) OFF-300 (012C hex)
   ON-300 (012C hex)

PROBLEM D-20

Make the program from Problem D-19 operate continuously, turning the LEDs on and off according to the duty cycle specified by the contents of memory locations 0030 and 0031 (OFF period) and 0032 and 0033 (ON period). Try the following test cases:

1) (0030) = 2C (012C hex = 300 decimal)
   (0031) = 01
   (0032) = 58 (0258 hex = 600 decimal)
   (0033) = 02
2) (0030) = F4 (01F4 hex = 500 decimal)
   (0031) = 01
   (0032) = F4 (01F4 hex = 500 decimal)
   (0033) = 01
3) (0030) = C2 (01C2 hex = 450 decimal)
   (0031) = 01
   (0032) = 58 (0258 hex = 600 decimal)
   (0033) = 02

KEEPING TIME IN STANDARD UNITS

We can make the service routine keep time in seconds and minutes rather than in units determined by the computer's word length. The following routine keeps the number of clock periods in memory location 0040, the number of seconds in 0041, and the number of minutes in 0042. As before, the main program must clear the counter locations initially. Program D-7 is the hexadecimal version. Note how the program produces a carry and returns the count to zero when the number of hundredths of seconds reaches 100 or when either the number of seconds or the number of minutes reaches 60.

PROGRAM D-7

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0280	48	PHA	
0281	E6	INC	$40
0282	40		
0283	A5	LDA	$40
0284	40		
0285	38	SEC	
0286	E9	SBC	#100

PROGRAM D-7 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0287	64			
0288	D0		BNE	ENDINT
0289	16			
028A	85		STA	$40
028B	40			
028C	E6		INC	$41
028D	41			
028E	A5		LDA	$41
028F	41			
0290	E9		SBC	#60
0291	3C			
0292	D0		BNE	ENDINT
0293	0C			
0294	85		STA	$41
0295	41			
0296	E6		INC	$42
0297	42			
0298	A5		LDA	$42
0299	42			
029A	E9		SBC	#60
029B	3C			
029C	D0		BNE	ENDINT
029D	02			
029E	85		STA	$42
029F	42			
02A0	A9	ENDINT	LDA	#$9C
02A1	9C			
02A2	8D		STA	$170E
02A3	0E			
02A4	17			
02A5	68		PLA	
02A6	40		RTI	

```
*=$0280
PHA             ;SAVE ACCUMULATOR
INC   $40       ;UPDATE HUNDREDTHS OF SECONDS
LDA   $40
SEC             ;IS THERE A CARRY TO SECONDS?
SBC   #100
BNE   ENDINT    ;NO, DONE (C = 1 IF NO BRANCH)
STA   $40       ;YES, MAKE HUNDREDTHS ZERO
INC   $41       ;UPDATE SECONDS
LDA   $41
```

```
        SBC     #60       ;IS THERE A CARRY TO MINUTES?
        BNE     ENDINT    ;NO, DONE (C = 1 IF NO BRANCH)
        STA     $41       ;YES, MAKE SECONDS ZERO
        INC     $42       ;UPDATE MINUTES
        LDA     $42
        SBC     #60       ;IS THERE A CARRY TO HOURS?
        BNE     ENDINT    ;NO, DONE
        STA     $42       ;YES, MAKE MINUTES ZERO
ENDINT  LDA     #$9C      ;RELOAD TIMER
        STA     $170E
        PLA               ;RESTORE ACCUMULATOR
        RTI
```

PROBLEM D-21

Write a main program that uses Program D-7 to wait for 1 min and 45 s.

PROBLEM D-22

Modify Program D-7 to keep time as pairs of decimal digits in memory locations 0040, 0041, and 0042. Be careful to operate in the decimal mode and remember that INC and DEC always produce binary results.

PROBLEM D-23

Extend the answer to Problem D-22 so it keeps hours as two decimal digits (00 to 23) in memory location 0043.

PROBLEM D-24

Write a main program that uses Program D-7 to turn all the LEDs attached to port B off for 1 min and 30 s and then on for 1 min and 15 s.

Note: In Problems D-25 through D-27, be careful when you calculate the time at which a period should end. Remember that neither the number of seconds in memory location 0041 nor the number of minutes in memory location 0042 can ever exceed 59. Thus you will have to perform your arithmetic mod 60 in handling seconds and minutes.

PROBLEM D-25

Make the program from Problem D-24 operate continuously, turning the LEDs off for 1 min and 30 s and then on for 1 min and 15 s.

PROBLEM D-26

We can handle more complex timing sequences by using tables. Write a program that turns the LEDs attached to port B of the user 6530 device on and off

according to the following table in memory locations 0380 through 0388. Each entry is the length of a period in seconds and the final zero marks the end of the table.

(0380) = 0A	(first OFF period is 10 s)
(0381) = 0F	(first ON period is 15 s)
(0382) = 14	(second OFF period is 20 s)
(0383) = 0F	(second ON period is 15 s)
(0384) = 0A	(third OFF period is 10 s)
(0385) = 14	(third ON period is 20 s)
(0386) = 05	(fourth OFF period is 5 s)
(0387) = 14	(fourth ON period is 20 s)
(0388) = 00	(ending marker)

The LEDs should be off for 10 s (0A hex), on for 15 s, off for 20 s, on for 15 s, off for 10 s, on for 20 s, off for 5 s, and finally on for 20 s. Remember to perform your arithmetic mod 60.

PROBLEM D-27

We can easily extend the table of Problem D-26 to handle values besides ON and OFF. Write a program that operates the LEDs attached to port B of the user 6530 device according to the following table in memory locations 0380 through 0388. Each entry consists of a length in seconds followed by a data value to be sent to the LEDs. The final zero marks the end of the table. Turn all the LEDs off before concluding. Remember that the number of seconds in the clock count can never exceed 59.

(0380) = 14	(first period is 20 s long)
(0381) = 00	(all LEDs on during first period)
(0382) = 1E	(second period is 30 s long)
(0383) = 01	(all LEDs on except bit 0 during second period)
(0384) = 14	(third period is 20 s long)
(0385) = 03	(all LEDs on except bits 0 and 1 during third period)
(0386) = 0A	(fourth period is 10 s long)
(0387) = 07	(all LEDs on except bits 0, 1, and 2 during fourth period)
(0388) = 00	(ending marker)

PROBLEM D-28

If we use Program D-7, what percentage of the processor's time is spent servicing the real-time clock? Determine both the maximum and the average percentage. Note that the 6502 requires seven clock cycles to respond to an interrupt; it uses that time to save the program counter and status register in the stack and to load the new program counter value from memory. In calculating the service time on the KIM, we must also consider the indirect jump (five clock cycles) that vectors the interrupt through memory addresses 17FE and 17FF. How much would the percentages increase if the clock produced an interrupt every 1 ms, instead of every 10 ms? What is the highest clock frequency we can use if we limit the service routine to 10% of the processor time on the average? What if we limit it to 10% maximum during any clock period?

REAL-TIME OPERATING SYSTEMS

The programmer can use a real-time clock to satisfy many timing requirements. Tasks can be scheduled or suspended, delays can be produced, and real-time inputs and outputs can be handled. The programmer must, however, determine the order and priority of tasks and specify exactly how they are to use the real-time clock.

A typical example application is a real-time monitoring system for process or industrial control. The system must regularly record data from the various input points (e.g., from sensors located at various positions in a pipeline, tank, or reactor), respond immediately to alarms, and report status to a central computer. The times at which alarms occur must be listed on a printer to provide permanent records. Note the numerous tasks that this system must perform: regular data logging, alarm recognition, alarm recording, printing, and communications with the central computer. The priority of the various tasks is critical. For example, if an alarm occurs while a message is being printed, the system must suspend the printing, record the time at which the alarm occurred, prepare a new message for later printing, and then resume the suspended task. A similar sequence of suspension and resumption is necessary if the central computer requests a report while the system is busy. The programmer must juggle the computer's resources so that it performs all the tasks in the proper order at the correct times without missing any data, alarms, or requests for reports.

A real-time operating system removes much of the burden of task management from the programmer. This piece of packaged software schedules tasks, handles communications between tasks (e.g., in the monitoring system it would provide routines that allow the alarm recording task to tell the printer task what to print), generates time intervals, and provides real-time interrupt control for I/O devices. The programmer

must learn only how to use the operating system. The articles by D. L. Ripps listed in the references for this laboratory describe a typical real-time operating system. The obvious advantages of such systems are that they can be purchased rather than written, and that they provide standard procedures and formats. The programmer can concentrate on his or her application, rather than on the scheduling and coordination of tasks.

Let us discuss how one would use such an operating system in the monitoring application. The programmer would first have to read the documentation for the operating system to determine its features and requirements. He or she would then write programs in the proper form to handle data logging, alarm monitoring, printing, and status reporting. Each of these programs (or *tasks*) would call subroutines (sometimes called *utilities*) from the operating system. The user tasks and the operating system would thus together control the monitoring system. Note that the programmer could change one task (e.g., attach a new printer, allow more alarms, or add a local data buffer) without changing the other tasks. We would like to thank Bill Renwick of Kadak Products Ltd. (Vancouver, B.C., Canada) for suggesting this example and describing how it would operate under Kadak's AMX Operating System.

KEY POINT SUMMARY

1) You can handle simple timing tasks with software delay routines. A standard routine that provides delays of varied lengths is often useful.

2) Programs can be made more flexible by allowing them to determine their own timing parameters from system inputs. The same program can then be used with peripherals operating at different data rates.

3) A program can easily examine a clock line, synchronize with it, and measure its period as long as its frequency is much lower than the CPU clock frequency.

4) A programmable timer can replace a delay routine. It simply indicates when a starting value loaded into it has been counted down to zero. Programmable timers add flexibility to systems because they can operate in a variety of modes under program control. However, there are no standards for the functions or programming of these timers, so they require careful use and documentation.

5) The 6530 peripheral interface/memory device has an 8-bit timer with a prescaler. This timer can be started at a specific count by storing data in its internal register; the address at which the data is stored determines the prescaler's divide ratio (1, 8, 64, or 1024) and whether reaching zero causes an interrupt.

6) Interrupts are a convenient way to handle timing. A real-time clock is a regular source of interrupts that can be counted to provide a basis for timing and scheduling. Time is specified in terms of the number of counts required.

7) A real-time operating system handles scheduling, coordination, and communications on a real-time basis. It provides a standard supervisor for applications with real-time requirements.

Laboratory E

Serial Input/Output

PURPOSE

To learn how to use the microcomputer to send and receive serial data.

PARTS REQUIRED

A debounced switch attached to the 6502 Set Overflow input (often referred to as S.O. or RO). This input line is available as pin 5 of the Expansion Connector as shown in Figure E-1.

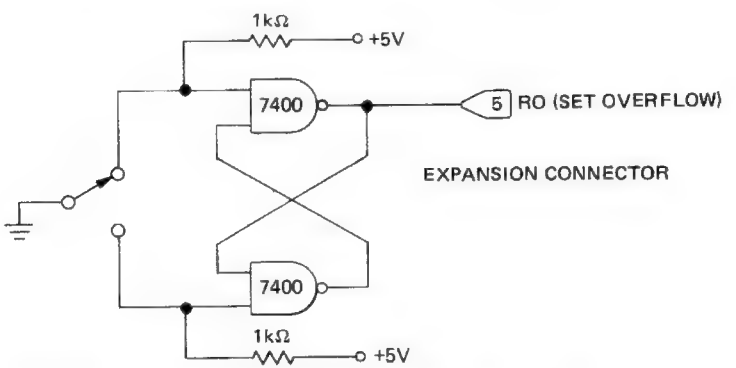


FIGURE E-1. Connections for the 6502 Set Overflow (RO) input.

REFERENCE MATERIALS

M. R. Corder, "6500 Program Automatically Sets Communications Chip Bit Rate," *Electronics,* March 13, 1980, pp. 149-151.

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, pp. 120-123.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 360-363, 385-388, 420-427, 489-492.

L. A. Leventhal, *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979, pp. 11-103 through 11-122, 12-32 through 12-36.

J. E. McNamara, *Technical Aspects of Data Communications,* Educational Services Department, Digital Equipment Corp., Maynard, MA, 1977, Chapters 1-3, 5, 15.

A. Osborne et al., *An Introduction to Microcomputers, Vol. 2: Some Real Microprocessors,* Osborne/McGraw-Hill, Berkeley, CA, 1978, pp. 9-55 through 9-61 (6850 ACIA or UART), 9-61 through 9-70 (6852 USRT), 10-76 through 10-88 (6551 ACIA or UART).

L. J. Scanlon, *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980, pp. 231-237.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 14-16 (alphanumeric codes), 16-17 (parity), 48-49 and 51-52 (shift registers), 237-240 (asynchronous serial I/O), 240-244 (UARTs), 245-254 (6850 ACIA), 255-259 (serial transmission standards), 279-281 (teletypewriters), 335-337 (shift instructions).

A. J. Weissberger, "Data-Link Control Chips: Bringing Order to Data Protocols," *Electronics,* June 8, 1978, pp. 104-112.

J. Wong et al., "Software Error Checking Procedures for Data Communications Protocols," *Computer Design,* February 1979, pp. 122-125.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 40-42 (KIM-1 operating programs).

MCS6500 Microcomputer Family Hardware Manual, Commodore/MOS Technology, Norristown, PA, 1976, p. 44 (Set Overflow input).

WHAT YOU SHOULD LEARN

1) What LSI chips are available to perform serial communications functions.
2) How to convert data between serial and parallel forms.
3) How to provide timing for serial data communications.
4) How to generate and recognize start and stop bits.
5) How to use the 6502's Set Overflow (S.O. or RO) input.

6) How to detect false start bits using majority logic.
7) How to generate and check parity.

TERMS

ASCII—American Standard Code for Information Interchange, a 7-bit character code widely used in computers and communications.

Baud—a measure of the rate at which serial data is transmitted, bits per second but including both data bits and bits used for synchronization, error checking, and other purposes. Common baud rates are 110, 300, 1200, 2400, 4800, and 9600.

Baud rate generator—a device that generates the proper time intervals between bits for serial data transmission.

BSC—Binary Synchronous Communications or BISYNC, an older line protocol often used by IBM computers and terminals.

Checksum—a logical sum that is included in a block of data to guard against recording or transmission errors. Also referred to as longitudinal parity or longitudinal redundancy check (LRC).

Cyclic redundancy check (CRC)—an error-detecting code generated from a polynomial that can be added to a block of data or a storage area.

Data-link control—a set of conventions governing the format and timing of data exchange between communicating systems. Also called a *protocol.*

DDCMP—Digital Data Communications Message Protocol, a widely used protocol that supports any method of physical data transfer (synchronous or asynchronous, serial or parallel).

Error-correcting code—a code that the receiver can use to correct errors in messages; the code itself does not contain any additional message.

Error-detecting code—a code that the receiver can use to detect errors in messages; the code itself does not contain any additional message.

False start bit—a start bit that does not last the minimum required amount of time, usually caused by noise on the transmission line.

Longitudinal parity—*see* Checksum.

Longitudinal redundancy check (LRC)—*see* Checksum.

Majority logic—a combinational logic function that is true when more than half the inputs are true.

Mark—the 1 state on a serial data communications line.

Modem—modulator/demodulator, a device that adds or removes a carrier frequency, thereby allowing data to be transmitted on a high-frequency channel or received from such a channel.

Parallel—more than one bit at a time.

Parity—a 1-bit code that makes the total number of 1 bits in the word, including the parity bit, odd (odd parity) or even (even parity). Also called vertical parity or vertical redundancy check (VRC).

Protocol—*see* Data-link control.

RS-232 (or **EIA RS-232**)—a standard interface for the transmission of serial digital data. It has been partially superseded by RS-449.

SDLC—Synchronous Data Link Control, the successor protocol to BSC for IBM computers and terminals.

Serial—one bit at a time.

Shift register—a clocked device that moves its contents 1 bit position to the left or right during each clock cycle.

Space—the zero state on a serial data communications line.

Standard teletypewriter—a teletypewriter that operates asynchronously at a rate of 10 characters per second.

Start bit—a 1-bit signal that indicates the start of data transmission by an asynchronous device.

Stop bit—a 1-bit signal that indicates the end of data transmission by an asynchronous device.

Synchronization (or **sync**) **character**—a character that is used only to synchronize the transmitter and the receiver.

Teletypewriter—a device containing a keyboard and a serial printer that is often used in communications and with computers. Also referred to as a Teletype (a registered trademark of Teletype Corporation of Skokie, Illinois) or TTY.

Universal asynchronous receiver/transmitter (UART)—an LSI device that acts as an interface between systems that handle data in parallel and devices that handle data in asynchronous serial form.

Universal synchronous receiver/transmitter (USRT)—an LSI device that acts as an interface between systems that handle data in parallel and devices that handle data in synchronous serial form.

6502 INSTRUCTIONS

CLV—clear overflow; set the OVERFLOW flag to zero. Note that there is no "set overflow" instruction.

ROL—rotate left; shift each bit of the accumulator or a memory location left one position as if bits 0 and 7 were connected through the CARRY flag (see Figure E-2).

ROR—rotate right; shift each bit of the accumulator or a memory location right one position as if bits 0 and 7 were connected through the CARRY flag (see Figure E-2). Some early 6502's do not have this instruction.

IMPLEMENTING SERIAL INTERFACES

Most I/O devices transfer data serially rather than in parallel. Serial communications is cheaper to implement than parallel communications, since the serial approach requires only a single data line. However, special interfaces are necessary to connect serial I/O devices to a computer that handles data in parallel.

This laboratory will describe how to interface serial I/O devices using both software and hardware. We will show how to convert data from serial to parallel or parallel to serial, how to provide timing, how to add start and stop bits, how to check and generate parity, and how to use the 6502's Set Overflow input to detect transitions on a serial input line.

Serial interfaces allow numerous tradeoffs between hardware and software. On the one hand, we can perform all the tasks that we have mentioned in software. On the other hand, we can add a serial interface chip at a cost of a few dollars. We may categorize serial interface chips as:

1) Asynchronous devices, called universal asynchronous receiver/transmitters or UARTs. UARTs perform the following functions:

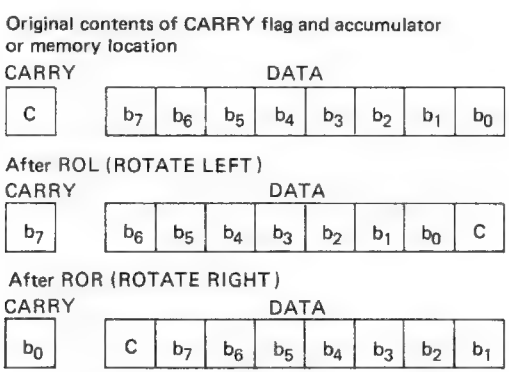


FIGURE E-2. 6502 shift instructions ROL and ROR.

- Serial/parallel conversion.
- Parity checking and generation.
- Start-bit recognition and generation.
- Stop-bit recognition and generation.
- Clocking.
- Buffering.

The references describe the 6850 and 6551 Asynchronous Communications Interface Adapters (ACIAs), UARTs specifically designed for use in 6800 and 6502-based systems.

2) Synchronous devices, called universal synchronous receiver/transmitters or USRTs. USRTs perform most of the UART functions under clock control and also generate and detect synchronization characters.

3) Data-link controllers. These devices perform all or most of the functions required by communications protocols such as BSC, DDCMP, and SDLC. The book by J. E. McNamara (listed in the references) describes all these protocols.

Chips like the 6551 and 6850 Asynchronous Communications Interface Adapters (UARTs), the 6852 Synchronous Serial Data Adapter (a USRT), and the 6854 Advanced Data Link Controller make serial interfaces easy to implement. Unless board space is not available or the number of parts must be minimized, designers prefer to use these chips in most applications. However, programs that perform serial communications functions are occasionally useful as well as instructive.

SERIAL/PARALLEL CONVERSION

Converting data from parallel to serial requires the use of the microprocessor's shift instructions. Since most serial data transmission starts with bit 0, we use the LSR and ROL instructions in the following program to place bit 0 of memory location 0060 on the LED attached to bit 0 of port B of the user 6530 device. Program E-1 is the hexadecimal version. LSR $60 moves one bit of data from memory location 0060 to the CARRY flag. ROL $1702 then moves that bit of data to bit 0 of memory location 1702, the output port for the LEDs. We have complemented the data, so that it will appear on the LEDs in positive logic. Program E-1 turns all the LEDs off initially by storing FF in location 1702.

```
LDX     #$7F      ;INITIALIZE USER STACK POINTER
TXS
```

```
LDA     #$FF    ;MAKE USER 6530 PORT B OUTPUT
STA     $1703
STA     $1702   ;TURN OFF THE LEDS
EOR     $60     ;COMPLEMENT THE DATA, (A) = FF
STA     $60
LSR     $60     ;GET ONE BIT OF PARALLEL DATA
ROL     $1702   ;MOVE BIT TO SERIAL OUTPUT PORT
BRK
```

PROGRAM E-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#$FF
0204	FF		
0205	8D	STA	$1703
0206	03		
0207	17		
0208	8D	STA	$1702
0209	02		
020A	17		
020B	45	EOR	$60
020C	60		
020D	85	STA	$60
020E	60		
020F	46	LSR	$60
0210	60		
0211	2E	ROL	$1702
0212	02		
0213	17		
0214	00	BRK	

Program E-1 transmits one bit of data. Execute it eight times starting with (0060) = AA hex (10101010 binary). After the first time, start at memory address 020F to avoid repeating the initialization instructions. The LED attached to bit 0 should alternately go off and on, since the data consists of alternating 0 and 1 bits, starting with a 0 in bit 0. ROL $1702 shifts the previous serial outputs left so you can see all the bits that have been transmitted (remember, however, that bit 6 is not connected). What does memory location 0060 contain when you are finished? Why? Note how Program E-1 simulates the effects of a shift register on the LEDs.

PROBLEM E-1

Make Program E-1 use bit 7 of port B as the serial output. The data now appears on the LEDs in its normal order, starting with bit 0 at the right, as opposed to the inverted order produced by eight executions of Program E-1. Unfortunately, we cannot use this approach in general, since we need bit 7 of port B as the timer interrupt in some exercises.

Converting inputs from serial to parallel is also simple. The following program (see Program E-2 for a hexadecimal version) fetches a serial input from bit 7 of user 6530 port A and combines it with the data in memory location 0061. We assume that bit 0 is received first. When you execute Program E-2 repeatedly, the data bits move to the right on the LEDs and end up in their normal order.

Clear memory location 0061 initially and execute Program E-2 eight times to assemble a byte of data. After the first time, start at memory address 020B to avoid repeating the initialization. Vary the switch position to produce the result (0061) = AA (hex).

```
LDX   #$7F     ;INITIALIZE USER STACK POINTER
TXS
LDA   #$FF     ;MAKE USER 6530 PORT B OUTPUT
STA   $1703
STA   $1702    ;TURN OFF THE LEDS
ASL   $1700    ;MOVE SERIAL INPUT TO CARRY
ROR   $61      ;AND COMBINE WITH PREVIOUS INPUTS
LDA   $61      ;SHOW DATA ON LEDS
EOR   #$FF     ;IN POSITIVE LOGIC
STA   $1702
BRK
```

PROGRAM E-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#$FF
0204	FF		
0205	8D	STA	$1703
0206	03		
0207	17		
0208	8D	STA	$1702
0209	02		

PROGRAM E-2 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
020A	17		
020B	0E	ASL	$1700
020C	00		
020D	17		
020E	66	ROR	$61
020F	61		
0210	A5	LDA	$61
0211	61		
0212	49	EOR	#$FF
0213	FF		
0214	8D	STA	$1702
0215	02		
0216	17		
0217	00	BRK	

PROBLEM E-2

Make Program E-2 start with bit 7 of the data and use bit 0 of port A as the serial input.

GENERATING BIT RATES

In real applications, the computer must wait the proper amount of time between bits. We can easily make the transmission program send the bits at a rate determined by a delay routine. The next example (see Program E-3 for a hexadecimal version) uses the monitor subroutine DELAY (see Laboratory D, particularly Table D-1). The data is originally in memory location 0060. You must load the delay parameter into memory locations 17F2 and 17F3; 8000 hex (00 in 17F2 and 80 in 17F3) will give you the maximum time between bits (about 0.5 s).

Run Program E-3 with (0060) = AA (hex) and with (0060) = 55 (hex). You can change the data rate by changing the parameter of subroutine DELAY. Try the following sequence of values in memory location 17F3: 80, 40, 20, 10, 08, 04, 02, 01. When can you no longer see the separate serial outputs? You can increase the data rate still further by clearing memory location 17F3 and reducing memory location 17F2 with a similar sequence. Be careful; Program E-3 destroys the data in memory location 0060, so you will have to reload it before you run the program again. Also beware of the fact that resetting the KIM changes location 17F3 to FF. You should check that location, particularly if your time

delay becomes surprisingly short (remember that DELAY exits after one iteration if its parameter is above 8000 hex).

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
        LDA   #$FF      ;MAKE USER 6530 PORT B OUTPUT
        STA   $1703
        STA   $1702     ;TURN OFF THE LEDS
        EOR   $60       ;COMPLEMENT THE DATA, (A) = FF
        STA   $60
        LDX   #8        ;NUMBER OF BITS = 8
OUTB    LSR   $60       ;MOVE SERIAL OUTPUT TO CARRY
        ROL   $1702     ;AND ON TO LEDS
        JSR   DELAY     ;WAIT A BIT TIME
        DEX
        BNE   OUTB      ;COUNT BITS
        BRK
```

PROGRAM E-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#$FF
0204	FF			
0205	8D		STA	$1703
0206	03			
0207	17			
0208	8D		STA	$1702
0209	02			
020A	17			
020B	45		EOR	$60
020C	60			
020D	85		STA	$60
020E	60			
020F	A2		LDX	#8
0210	08			
0211	46	OUTB	LSR	$60
0212	60			
0213	2E		ROL	$1702
0214	02			
0215	17			
0216	20		JSR	DELAY
0217	D4			

PROGRAM E-3 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0218	1E		
0219	CA	DEX	
021A	D0	BNE	OUTB
021B	F5		
021C	00	BRK	

PROBLEM E-3

Write a serial data reception program that waits between bits using the maximum length of subroutine DELAY. Assume that the serial data starts with bit 0 and is received from bit 7 of port A of the user 6530 device. Run the program, setting the input switch so that the final data value is (0061) = AA (hex). Subroutine DELAY will allow you about 0.5 s to move the input switch to the correct position for the next bit; if you need more time, use the single-step mode. Remember that the processor will still execute subroutine DELAY at its normal rate, since the single-step mode does not apply to monitor routines (see Laboratory A).

USING THE REAL-TIME CLOCK

We can also use the real-time clock (see Laboratory D) to wait between bits. The following program initializes the real-time clock and transmits a bit each time the count in memory location 0040 increases by 100 (i.e., at 1-s intervals, since the clock frequency is 100 Hz). Program E-4 is the hexadecimal version. Before executing Program E-4, you must load 0280 into the interrupt vector (memory addresses 17FE and 17FF) provided by the KIM monitor.

Note that we must make some housekeeping changes to use the real-time clock (see Laboratory D). We must designate bit 7 of port B as an input to use that line for timer interrupts; we must also connect bit 7 to the processor's $\overline{\text{IRQ}}$ input (see Figure D-3). We have changed bit 7 from an output to an input by replacing STA $1703 (which stores FF in 1703) with STX $1703 (which stores 7F in 1703); thus we have 8E in memory location 0205 instead of 8D. Remember, however, that bit 7 of port B is no longer an output and no data will ever appear on the LED that was attached to that bit. We must also replace our terminating BRK instruction with JSR $1C05.

Enter and run Program E-4 with (0060) = AA and with (0060) = 55. Note that the interrupt service routine does not affect any registers or

flags. What happens if adding 100 to the current clock count produces a carry? Our procedure is like keeping track of minutes on a watch while ignoring hours. Remember to clear the final timer interrupt by examining memory location 1706 after your program returns control to the monitor.

PROGRAM E-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#$FF
0204	FF			
0205	8E		STX	$1703
0206	03			
0207	17			
0208	8D		STA	$1702
0209	02			
020A	17			
020B	45		EOR	$60
020C	60			
020D	85		STA	$60
020E	60			
020F	A9		LDA	#0
0210	00			
0211	85		STA	$40
0212	40			
0213	A9		LDA	#$9C
0214	9C			
0215	8D		STA	$170E
0216	0E			
0217	17			
0218	58		CLI	
0219	A2		LDX	#8
021A	08			
021B	46	OUTB	LSR	$60
021C	60			
021D	2E		ROL	$1702
021E	02			
021F	17			
0220	A5		LDA	$40
0221	40			
0222	18		CLC	
0223	69		ADC	#100
0224	64			

PROGRAM E-4 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0225	C5	WTCLK	CMP	$40
0226	40			
0227	D0		BNE	WTCLK
0228	FC			
0229	CA		DEX	
022A	D0		BNE	OUTB
022B	EF			
022C	78		SEI	
022D	20		JSR	$1C05
022E	05			
022F	1C			
0280	48		PHA	
0281	E6		INC	$40
0282	40			
0283	A9		LDA	#$9C
0284	9C			
0285	8D		STA	$170E
0286	0E			
0287	17			
0288	68		PLA	
0289	40		RTI	

```
        LDX     #$7F    ;INITIALIZE USER STACK POINTER
        TXS
        LDA     #$FF
        STX     $1703   ;MAKE PORT B OUTPUT EXCEPT BIT 7
        STA     $1702   ;TURN OFF THE LEDS
        EOR     $60     ;COMPLEMENT DATA
        STA     $60
        LDA     #0      ;CLEAR CLOCK COUNT
        STA     $40
        LDA     #$9C    ;SET TIMER FOR 10 MS
        STA     $170E   ;START TIMER
        CLI             ;ENABLE CPU INTERRUPT
        LDX     #8      ;NUMBER OF BITS = 8
OUTB    LSR     $60     ;MOVE SERIAL OUTPUT TO CARRY
        ROL     $1702   ;AND ON TO LEDS
        LDA     $40     ;GET STARTING CLOCK COUNT
        CLC
        ADC     #100    ;CALCULATE TARGET VALUE
```

```
WTCLK   CMP     $40     ;TARGET VALUE REACHED?
        BNE     WTCLK   ;NO, WAIT
        DEX             ;YES, COUNT BITS
        BNE     OUTB
        SEI             ;DISABLE CPU INTERRUPT
        JSR     $1C05

        *=$0280         ;CLOCK SERVICE ROUTINE
        PHA             ;SAVE ACCUMULATOR
        INC     $40     ;INCREMENT CLOCK COUNT
        LDA     #$9C    ;START TIMER AGAIN
        STA     $170E
        PLA             ;RESTORE ACCUMULATOR
        RTI
```

PROBLEM E-4

Make Program E-4 wait between bits for the number of clock interrupts in memory location 0061. The revised program could transmit data at any rate if the system measured the time between bits using the method shown in Program D-3.

PROBLEM E-5

Make the serial reception program wait for 100 real-time clock interrupts between bits. Assume that the serial data starts with bit 0 and that the serial input is bit 7 of port A.

Sample Cases:

1) If you leave the switch open while the program executes, all the inputs will be 1's.

   Result: (0061) = FF

2) If you leave the switch closed while the program executes, all the inputs will be 0's.

   Result: (0061) = 00

PROBLEM E-6

Make Program E-4 use Program D-7 as the interrupt service routine. Now time is being kept in minutes, seconds, and hundredths of seconds. Make the time between bit outputs 1 s.

PROBLEM E-7

Make the answer to Problem E-5 use Program D-7 as the interrupt service routine. The time between serial input operations should be 1 s.

START AND STOP BITS

In the previous discussion, we have assumed that we can start and stop transmission and reception at any time. Of course, this is not generally the case, since external factors such as the availability of data or the rate at which an I/O device can handle data usually control the transfer. In real applications, the receiver must determine when data is available and must identify the beginning and ending of the transmission.

One simple way of marking the beginning and ending of the transmission is to place start and stop bits around the actual data. Figure E-3 shows the standard teletypewriter format in which a start bit (or logic 0) precedes the data and 2 stop bits (or logic 1's) follow the data and separate one character from the next. Note that the data line is normally in the 1 state (called the *mark* state on a teletypewriter). The transition to the 0 (or *space*) state signifies the start of transmission.

Note the advantages of this approach:

1) Transmission can start at any time. No clock is required.
2) The start and stop bits are easy to generate and detect.
3) The start bit can produce an interrupt.
4) The stop bits separate characters.

Of course, the approach also has disadvantages. Noise can easily produce false start bits; we will discuss methods for reducing their frequency later. Adding start and stop bits reduces the actual data rate (since more bits must be transmitted) and increases the overhead for each character. An alternative is to group the characters into blocks and provide synchronization only on a block-by-block basis. Most protocols use this approach in order to provide higher speed and greater reliability.

We can easily modify Program E-3 to produce an initial start bit. All that we must do is clear the CARRY flag initially (with the CLC instruction) and shift memory location 0060 at the end of the loop instead of at the beginning (so bit 0 of the data is actually the second bit transmitted instead of the first). Note that we have also increased the bit count by 1. Program E-5 is the hexadecimal version.

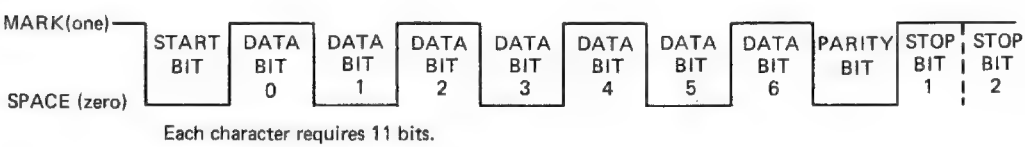


FIGURE E-3. Standard teletypewriter data format.

PROGRAM E-5

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#$FF
0204	FF			
0205	8D		STA	$1703
0206	03			
0207	17			
0208	8D		STA	$1702
0209	02			
020A	17			
020B	45		EOR	$60
020C	60			
020D	85		STA	$60
020E	60			
020F	A2		LDX	#9
0210	09			
0211	18		CLC	
0212	2E	OUTB	ROL	$1702
0213	02			
0214	17			
0215	20		JSR	DELAY
0216	D4			
0217	1E			
0218	46		LSR	$60
0219	60			
021A	CA		DEX	
021B	D0		BNE	OUTB
021C	F5			
021D	00		BRK	

```
        LDX   #$7F     ;INITIALIZE USER STACK POINTER
        TXS
        LDA   #$FF     ;MAKE USER 6530 PORT B OUTPUT
        STA   $1703
        STA   $1702    ;TURN OFF THE LEDS
        EOR   $60      ;COMPLEMENT THE DATA
        STA   $60
        LDX   #9       ;NUMBER OF BITS = 9
        CLC            ;FORM START BIT
OUTB    ROL   $1702    ;MOVE SERIAL OUTPUT TO LEDS
        JSR   DELAY    ;WAIT A BIT TIME
```

```
LSR     $60     ;MOVE NEXT SERIAL OUTPUT TO CARRY
DEX
BNE     OUTB    ;COUNT BITS
BRK
```

Enter Program E-5 into memory. Before you execute it, remove the changes we made to use the real-time clock. That is, put 1C00 in memory locations 17FE and 17FF, remove the timer interrupt connection, and make bit 7 of port B an output again (by placing 8D in memory location 0205 instead of 8E). Run Program E-5 with (0060) = 00 and with (0060) = AA. The start bit will appear as a light in front of the actual data, since a logic 0 lights an LED. By making the following changes to Program E-5, we can also generate 2 stop bits.

1) Make the bit count 11 instead of 9.
2) Replace LSR $60 with the sequence SEC (SET CARRY), ROR $60 so that 1's are automatically shifted in at the left as the data is shifted to the right. The 1's will form the stop bits at the end.

PROBLEM E-8

Write and run a transmission program that generates a start bit and 2 stop bits. How would you modify your program to produce 1 stop bit instead of 2? Many 30-cps (characters per second) terminals (such as the popular Texas Instruments Silent 700) use a 10-bit data format with 1 stop bit.

PROBLEM E-9

A few older terminals use a data format with 1½ stop bits. Modify the program from Problem E-8 to produce data in that format. Assume that one bit time is given by the maximum length of the DELAY subroutine as in Program E-5. You can use subroutine DEHALF (starting address 1EEB) to produce a delay of one half of a bit time.

Receiving data with the start and stop bits included is more difficult than transmitting the data. The program must detect the falling edge (a 1-to-0 or high-to-low transition), which signifies the beginning of a start bit. Remember that the line is normally in the 1 or *mark* state. The program must then wait half a bit time (after detecting the transition) to center the reception. This delay causes the computer to read the data bits near the centers of the pulses rather than at the edges. Reading near the centers avoids errors if the data changes as in Figure E-3. Centering also makes it unnecessary to generate precise time intervals, since a little drift from the center does not matter. Program E-6 is the hexadecimal version of the following program. The KIM monitor provides subroutine DEHALF

(starting address 1EEB) which produces the one-half bit time delay; the teletypewriter monitor uses this routine to receive serial data from a terminal.

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
WTSTB   LDA   $1700     ;WAIT FOR A START BIT (ZERO)
        BMI   WTSTB
        JSR   DEHALF    ;CENTER WITH HALF BIT TIME DELAY
        LDX   #8        ;NUMBER OF BITS = 8
INBIT   JSR   DELAY     ;WAIT A BIT TIME
        ASL   $1700     ;MOVE SERIAL INPUT TO CARRY
        ROR   $61       ;AND COMBINE WITH PREVIOUS DATA
        DEX
        BNE   INBIT     ;COUNT BITS
        BRK
```

PROGRAM E-6

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	AD	WTSTB	LDA	$1700
0204	00			
0205	17			
0206	30		BMI	WTSTB
0207	FB			
0208	20		JSR	DEHALF
0209	EB			
020A	1E			
020B	A2		LDX	#8
020C	08			
020D	20	INBIT	JSR	DELAY
020E	D4			
020F	1E			
0210	0E		ASL	$1700
0211	00			
0212	17			
0213	66		ROR	$61
0214	61			
0215	CA		DEX	
0216	D0		BNE	INBIT
0217	F5			
0218	00		BRK	

Enter and run Program E-6. You can use the single-step mode to give yourself more time to move the switch to the proper position for each data value. The program will not get past the initial loop until you close the switch and form a start bit. It will then execute the two delay routines at full speed, since the single-step mode does not apply to the monitor. Be careful to set the switch at the right time—that is, before you let the computer execute subroutine DELAY. It will treat the JSR DELAY instruction as a single step, but it will then execute subroutine DELAY and the ASL $1700 instruction (moving the serial input to the CARRY) as soon as you press GO. Try the following sample cases:

1) Start with the switch closed and change its position each time the computer enters subroutine DELAY.
   Result: (0061) = 55 (hex)
2) Leave the switch closed all the time.
   Result: (0061) = 00
3) Close the switch to form the start bit and then immediately open it and leave it open.
   Result: (0061) = FF (hex)

PROBLEM E-10

Make Program E-6 check to see if there are 2 stop bits (logic 1's) following the data. The revised program should set memory location 0062 to 00 if the two stop bits are present and to FF if they are not. Lack of the proper number of stop bits is called a *framing error.*

PROBLEM E-11

Make Program E-6 check for the number of stop bits in memory location 0063. Assume that location 0063 always contains 0, 1, or 2.

PROBLEM E-12

Make Program E-5 wait for 100 real-time clock interrupts between bit outputs. Use the interrupt service routine in Program E-4.

PROBLEM E-13

Make Program E-6 wait for 100 real-time clock interrupts between bit inputs. Use the interrupt service routine in Program E-4.

PROBLEM E-14

Change Program E-6 so that it does not need to count bits. **Hint:** Load memory location 0061 with 10000000 (binary) or 80 hexadecimal initially and exit when the program shifts the 1 bit into the CARRY flag.

USING THE SET OVERFLOW INPUT

Another way to detect a start bit is to use the 6502's Set Overflow input. A negative transition on that line sets the OVERFLOW flag. So if we tie the serial data to the Set Overflow line (see Figure E-1), we can wait for a start bit as follows:

1) Clear the OVERFLOW flag.
2) Wait for a 1-to-0 transition to set the OVERFLOW flag.

The following program (see Program E-7 for a hexadecimal version) will wait for a start bit, assuming that the same switch is tied to both bit 7 of user 6530 port A and to the Set Overflow line. Of course, you can simulate the double connection by keeping separate switches in the same positions.

PROGRAM E-7

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	B8		CLV	
0204	50	WTSTB	BVC	WTSTB
0205	FE			
0206	20		JSR	DEHALF
0207	EB			
0208	1E			
0209	A2		LDX	#8
020A	08			
020B	20	INBIT	JSR	DELAY
020C	D4			
020D	1E			
020E	0E		ASL	$1700
020F	00			
0210	17			
0211	66		ROR	$61
0212	61			
0213	CA		DEX	
0214	D0		BNE	INBIT
0215	F5			
0216	00		BRK	

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
        CLV             ;CLEAR THE OVERFLOW FLAG
WTSTB   BVC   WTSTB     ;AND WAIT UNTIL A START BIT SETS IT
        JSR   DEHALF    ;CENTER WITH HALF BIT TIME DELAY
        LDX   #8        ;NUMBER OF BITS = 8
INBIT   JSR   DELAY     :WAIT A BIT TIME
        ASL   $1700     ;MOVE SERIAL INPUT TO CARRY
        ROR   $61       ;AND COMBINE WITH PREVIOUS DATA
        DEX             ;COUNT BITS
        BNE   INBIT
        BRK
```

Using special hardware facilities such as the Set Overflow input is convenient occasionally, but it also creates many problems. Programs that depend on those facilities are often difficult to understand; Program E-7, for example, seems to contain an endless loop, since BVC WTSTB branches to itself until the external transition sets the OVERFLOW flag. Even with careful documentation, programs like E-7 are awkward to maintain and update. Furthermore, the special hardware facilities lack generality. There is only one Set Overflow line and a second serial input would require a different approach. The special facilities may also interfere with normal program execution. If you use the Set Overflow line to detect start bits, you must be careful that ADC or SBC instructions do not interfere with detection by changing the OVERFLOW flag. Incorrect use of the OVERFLOW flag could be difficult to trace and correct. Designers normally use special hardware facilities only in simple, low-performance applications. Reducing the number of parts by using such facilities can save a lot of money in high-volume products such as electronic games, small instruments, and low-cost printers.

PROBLEM E-15

Make Program E-7 use the real-time clock. Allow 100 clock interrupts between bit inputs. Remember to change memory locations 17FE and 17FF to 0280 before running the interrupt-driven program. Also remember to change those locations back to 1C00 when you return to using BRK.

DETECTING FALSE START BITS

Many errors can occur in data communications, particularly if the connections are noisy (like the ordinary telephone network) or if the distances are long. One problem is that noise may make the input briefly a logic 0, thus producing a *false start bit.* The receiver can distinguish be-

tween a short noise pulse and a real start bit by sampling the line several times and requiring that a majority of the samples be 0's. That is, the receiver uses majority logic to decide whether to accept the start bit.

The following program samples the data at one quarter, one half, and three quarters of a bit time after the initial detection of a logic 0. It requires that at least two samples be 0's. Program E-8 is the hexadecimal version. If the computer accepts the start bit, it must then wait one quarter of a bit time to reach the beginning of data bit 0 (see Figure E-3).

Run Program E-8 in the single-step mode so that you can control the switch and trace the sampling. Majority logic works like voting; the value that the computer finds most often "wins." Try the following cases (starting with the switch closed to form the initial logic 0):

1) Change the switch's position every time the computer executes JSR DELAY (thus reaching address 1ED4). The sample values will be 1, 0, and 1. Since only one sample is 0, the computer should reject the start bit and return to address WTSTB (0208).
2) Leave the switch closed until the computer enters subroutine DELAY for the second time and then change the switch's position after each entry. The sample values will be 0, 1, and 0. Since two samples are 0's, the computer should accept the start bit and return control to the monitor.

```
        LDX   #$7F     ;INITIALIZE USER STACK POINTER
        TXS
        LDA   #$20     ;SET DELAY FOR 1/4 BIT TIME
        STA   $17F3
WTSTB   LDA   $1700    ;WAIT FOR A START BIT (ZERO)
        BMI   WTSTB
        LDX   #0       ;CLEAR ZERO COUNT TO START
        LDA   #3       ;NUMBER OF SAMPLES = 3
        STA   $30
CHBIT   JSR   DELAY    ;WAIT 1/4 BIT TIME
        LDA   $1700    ;IS DATA ZERO?
        BMI   CSAMP
        INX            ;YES, INCREMENT ZERO COUNT
CSAMP   DEC   $30      ;COUNT SAMPLES
        BNE   CHBIT
        CPX   #2       ;WAS MAJORITY OF SAMPLES ZERO?
        BCC   WTSTB    ;NO, FALSE START BIT
        JSR   DELAY    ;YES, WAIT 1/4 BIT TIME TO FINISH
        LDA   #$80     ;RESTORE FULL BIT TIME
        STA   $17F3
        BRK
```

PROGRAM E-8

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#$7F
0201	7F			
0202	9A		TXS	
0203	A9		LDA	#$20
0204	20			
0205	8D		STA	$17F3
0206	F3			
0207	17			
0208	AD	WTSTB	LDA	$1700
0209	00			
020A	17			
020B	30		BMI	WTSTB
020C	FB			
020D	A2		LDX	#0
020E	00			
020F	A9		LDA	#3
0210	03			
0211	85		STA	$30
0212	30			
0213	20	CHBIT	JSR	DELAY
0214	D4			
0215	1E			
0216	AD		LDA	$1700
0217	00			
0218	17			
0219	30		BMI	CSAMP
021A	01			
021B	E8		INX	
021C	C6	CSAMP	DEC	$30
021D	30			
021E	D0		BNE	CHBIT
021F	F3			
0220	E0		CPX	#2
0221	02			
0222	90		BCC	WTSTB
0223	E4			
0224	20		JSR	DELAY
0225	D4			
0226	1E			
0227	A9		LDA	#$80
0228	80			
0229	8D		STA	$17F3
022A	F3			
022B	17			
022C	00		BRK	

PROBLEM E-16

Revise Program E-8 to check the input at intervals of one eighth of a bit time. Now, at least four of the seven samples must be zero for the start bit to be accepted. Your program should wait for the end of the start bit and restore the maximum delay constant before exiting.

PROBLEM E-17

Write a reception program that checks each received bit at one fourth, one half, and three fourths of a bit time and determines the actual value by majority logic. That is, the value of at least two of the samples is taken as the value of the bit. Your program should start at the beginning of data bit 0, assuming that the initialization routine has detected a start bit but has not centered the reception.

GENERATING AND CHECKING PARITY

Still another way to reduce the number of errors is to include error-detecting or correcting codes with the data. These codes show whether the data was received correctly and, if not, where the errors were; they contain no additional information and thus reduce the rate at which actual data can be sent.

Parity is a simple error-detecting code. This is a single bit added to each word, which makes the total number of 1 bits even (if even parity) or odd (if odd parity). Note the following examples:

1) Data = 01101101: Even parity = 1, since the data contains an odd number (5) of 1 bits.
2) Data = 00010001: Even parity = 0, since the data contains an even number (2) of 1 bits.

Parity has the following features:

1) It allows the receiver to detect single but not double errors. Two bits received incorrectly will result in the same parity as that generated from the original data.
2) It does not allow for error correction. If the parity of the received data is wrong, there is no way of telling which bit is in error. All the receiver can do is ask the transmitter to send the data again.

Parity is particularly easy to implement for data consisting of 7-bit ASCII characters, since it can be stored in bit 7. Most UARTs and other communications chips, as we have mentioned, will automatically generate parity for transmission and check it on reception. The user can generally

choose whether the chip implements parity, which type of parity (even or odd) it implements, and how many bits it includes in each character.

The easiest way to generate parity with the 6502 microprocessor is to count the 1 bits. The least significant bit of the count will be 1 if the data contains an odd number of 1 bits and 0 if it contains an even number. Thus that bit is an even parity bit, since it makes the total number of 1 bits in the word (including itself) even. We can easily combine the counting of 1 bits with serial transmission (Program E-3). The following program (Program E-9 is the hexadecimal version) generates even parity and sends it after the actual 8-bit data. We have assumed that the data is originally in memory location 0060. Since Program E-9 does not complement the data (to avoid confusion in generating parity), the serial outputs will appear on the LEDs in negative logic (0 = light on, 1 = light off).

```
        LDX   #$7F      ;INITIALIZE USER STACK POINTER
        TXS
        LDA   #$FF      ;MAKE USER 6530 PORT B OUTPUT
        STA   $1703
        STA   $1702     ;TURN OFF THE LEDS
        LDA   #0        ;START PARITY AT ZERO
        STA   $50
        LDX   #8        ;NUMBER OF BITS = 8
OUTB    LSR   $60       ;MOVE SERIAL OUTPUT TO CARRY
        BCC   SENDB     ;WAS SERIAL OUTPUT 1?
        INC   $50       ;YES, ADD TO BIT COUNT
SENDB   ROL   $1702     ;SEND SERIAL OUTPUT TO LEDS
        JSR   DELAY     ;WAIT A BIT TIME
        DEX
        BNE   OUTB      ;COUNT BITS
        LSR   $50       ;FORM PARITY BIT FROM BIT COUNT
        ROL   $1702     ;TRANSMIT EVEN PARITY
        JSR   DELAY     ;WAIT A BIT TIME
        BRK
```

PROGRAM E-9

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)	
0200	A2	LDX	#$7F
0201	7F		
0202	9A	TXS	
0203	A9	LDA	#$FF
0204	FF		
0205	8D	STA	$1703
0206	03		

PROGRAM E-9 (continued)

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0207	17			
0208	8D		STA	$1702
0209	02			
020A	17			
020B	A9		LDA	#0
020C	00			
020D	85		STA	$50
020E	50			
020F	A2		LDX	#8
0210	08			
0211	46	OUTB	LSR	$60
0212	60			
0213	90		BCC	SENDB
0214	02			
0215	E6		INC	$50
0216	50			
0217	2E	SENDB	ROL	$1702
0218	02			
0219	17			
021A	20		JSR	DELAY
021B	D4			
021C	1E			
021D	CA		DEX	
021E	D0		BNE	OUTB
021F	F1			
0220	46		LSR	$50
0221	50			
0222	2E		ROL	$1702
0223	02			
0224	17			
0225	20		JSR	DELAY
0226	D4			
0227	1E			
0228	00		BRK	

Enter and run Program E-9 for the following examples:

1) (0060) = 41 ASCII A
   Result: Even parity bit = 0, since the data has two 1 bits (41 hex = 01000001 binary).

2) (0060) = 43 ASCII C
Result: Even parity bit = 1, since the data has three 1 bits (43 hex = 01000011 binary).

The results of Program E-9 are confusing, since bit 0 has been lost off the end and the bit positions are inverted. You can restore positive logic by complementing the final data as follows:

0228	AD	LDA	$1702
0229	02		
022A	17		
022B	49	EOR	#$FF
022C	FF		
022D	8D	STA	$1702
022E	02		
022F	17		
0230	00	BRK	

Regardless of whether you complement the data, the final values on the LEDs should be (left to right or bit position 7 to bit position 0): data bit 1, nothing (the KIM's unconnected bit), data bit 3, data bit 4, data bit 5, data bit 6, data bit 7, and the even parity bit. In example 1 above, the values will be 0, X, 0, 0, 0, 1, 0, 0, since 41 hex = 01000001 binary (the X represents the unconnected bit). In example 2, the values will be 1, X, 0, 0, 0, 1, 0, 1, since 43 hex = 01000011 binary.

PROBLEM E-18

Many computers and peripherals use 7-bit ASCII characters and reserve the most significant bit (bit position 7) for parity. Make Program E-9 transmit 7-bit characters followed by an even parity bit.

Examples:

1) (0060) = 41 ASCII A
Result: Transmitted data should be 41, since its parity is even.

2) (0060) = 43 ASCII C
Result: Transmitted data should be C3, since the parity of 43 is odd.

PROBLEM E-19

Write a serial reception program that calculates the parity of the received data as it is being converted to parallel form. The program should place the parallel data

in memory location 0061 and set memory location 0062 to 0 if the parity is even and to 1 if the parity is odd.

Examples:

1) Received data is 41 (hex) = 01000001 (binary).
Result: (0061) = 41 (parallel data)
(0062) = 00 since 41 hex has an even number of 1 bits.

2) Received data is C1 (hex) = 11000001 (binary).
Result: (0061) = C1 (parallel data)
(0062) = 01 since C1 hex has an odd number of 1 bits.

PROBLEM E-20

Make the answer to Problem E-19 check to see if bit 7 of the data is actually an odd parity bit. The program should place the parallel data in memory location 0061 and set memory location 0062 to 0 if the parity is correct and to 1 if the parity is wrong.

Examples:

1) Received data is FF (hex) = 11111111 (binary)
Result: (0061) = FF (parallel data)
(0062) = 01 since the parity of 11111111 is even.

2) Received data is C1 (hex) = 11000001 (binary)
Result: (0061) = C1 (parallel data)
(0062) = 00 since the parity of 11000001 is odd.

Example 1 shows why odd parity is more commonly used than even parity with 7-bit data. The reason is that a string of 1's or 0's caused by a hardware fault shows up as an error if odd parity is being used, but not if even parity is being used. Think of what the data would be if the external input were open-circuited or short-circuited.

KEY POINT SUMMARY

1) Serial I/O requires such interfacing functions as parallel/serial conversion, the addition and detection of start and stop bits, clocking, and parity generation and checking. Either hardware (UARTs, USRTs, and data-link control chips) or software can perform these functions.

2) Serial/parallel conversion can easily be performed with shift instructions. Only a few changes in the initial and final conditions are necessary to generate or detect start and stop bits.

3) Serial data can be clocked in or out by any of the timing methods that have been discussed previously. Software delay loops, programmable timers, or a real-time clock can do the job.

4) You can reduce the number of errors in serial communications by centering the reception, by sampling bits several times and using majority logic, or by including an error-detecting or correcting code such as parity. Parity can be generated by counting the number of 1 bits in the data; even parity is the least significant bit of the count.

5) Special serial I/O lines such as the 6502's Set Overflow input are occasionally useful, particularly in high-volume applications in which the number of parts must be minimized.

6) Serial I/O involves numerous tradeoffs. Simple synchronization methods are easy to implement, but require extra overhead for each character. They are typically sensitive to noise and provide only limited error detection and correction facilities. More complex methods are more difficult to implement, but can handle higher data rates and provide more reliable communications. New protocols usually require the grouping of characters into blocks and management on a block-by-block basis.

Laboratory F

Microcomputer Timing and Control

PURPOSE

To learn how the 6502 microprocessor executes instructions and how the addresses in the memory and I/O sections of 6502-based microcomputers are decoded.

PARTS REQUIRED

A dual-trace oscilloscope with a bandwidth of at least 5 MHz.

REFERENCE MATERIALS

M. L. DeJong, *Programming and Interfacing the 6502,* Howard W. Sams, Indianapolis, IN, 1980, Chapters 11 through 13.

P. Dittman et al., "Logic Analyzers Simplify System Integration Tasks," *Computer Design,* March 1981, pp. 119-126.

E. S. Donn and M. D. Lippman, "Efficient and Careful Microcomputer Testing Requires Careful Preplanning," *EDN,* February 20, 1979, pp. 97-107.

L. A. Leventhal, *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 284-316.

G. J. Lipovski, *Microcomputer Interfacing,* D. C. Heath (Lexington Books), Lexington, MA, 1980, Chapter 2.

R. Lorentzen, "Logic Analyzers Finish What Development Systems Start," *Electronic Design,* March 29, 1980, pp. 81-85.

J. McLeod, "Special Report: Logic Analyzers–Smart Fault-Finders Getting Smarter," *Electronic Design,* March 29, 1980, pp. 48-56.

C. A. Ogdin, "Setting Up a Microcomputer Design Laboratory," *Mini-Micro Systems,* May 1979, pp. 87-94.

A. Osborne, *An Introduction to Microcomputers, Vol. 2: Some Real Microprocessors,* Osborne/McGraw-Hill, Berkeley, CA, 1978, Chapter 10.

J. B. Peatman, *Microcomputer-Based Design,* McGraw-Hill, New York, 1977, Chapter 3.

R. J. Tocci and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 35-36 (tristate logic), 38-39 (clock signals), 39-50 (flip-flops), 52-53 (tristate registers), 53-58 (data bus), 58-59 (decoders), 60-62 (multiplexers and demultiplexers), 63-64 (memory devices), 65-73 (semiconductor memories), 73-79 (combining memory chips), 99-101 (instruction words), 101-105 (program example), 105-109 (operating cycles), 109-112 (instruction word formats), 127-137 (microcomputer structure), 137-142 (read and write operations), 142-146 (address allocation), 146-154 (address decoding), 160-162 (timing and control section).

M. J. Weisberg, "Designer's Guide to Testing and Troubleshooting Microprocessor-Based Products," *EDN,* March 20, 1980, pp. 177-214.

KIM-1 Microcomputer Module User Manual, Commodore/MOS Technology, Norristown, PA, 1976, Chapter 6.

MCS6500 Microcomputer Family Hardware Manual, Commodore/MOS Technology, Norristown, PA, 1976, pp. 41-45, 108-122, 123-151, Appendix A.

WHAT YOU SHOULD LEARN

1) Why a logic analyzer is necessary for troubleshooting microprocessor-based systems.
2) What kind of clock the 6502 microprocessor uses.
3) When the 6502 processor changes addresses and what the Synchronization (SYNC) signal means.
4) How the 6502 microprocessor executes instructions.
5) What part of the 6502 instruction cycle is used to transfer data.
6) How the address lines are decoded to activate memories and I/O devices.
7) How the designer can make tradeoffs between the memory capacity of a computer and the number of parts required to decode addresses.
8) How to decode I/O addresses efficiently using linear select.

Access time—the delay between the time when a memory receives an address and the time when the data from that address is available at the outputs.

Address—the identification code that distinguishes one memory location or input/output port from another and that can be used to select a specific one.

Address bus—the bus that the CPU uses to select a particular element of the memory or input/output section.

Address space—the total range of addresses to which a particular computer may refer.

Bidirectional—capable of transporting signals in either direction.

Bus—a group of parallel lines that connect two or more devices.

Bus contention—a situation in which two or more devices are trying to place data on a bus at the same time.

Clock—a regular timing signal that governs transitions in a system.

Decoder—a device that produces unencoded outputs from coded inputs. Also may refer to a device that converts data from one code to another.

Dual inline package (DIP or **bug)**—a semiconductor chip package having two rows of pins in a plane perpendicular to the edges of the package. Sometimes called a *bug,* since it appears to have legs.

Dynamic memory—a memory that loses its contents gradually without any external causes. The contents must be rewritten periodically if they are to be retained; the rewriting process is referred to as *refresh.*

Enable—allow an activity to proceed or a device to produce data outputs.

High-impedance state—*see* Tristate.

Hold time—the amount of time after the end of an activity signal during which some other signal must be stable (constant) to ensure the achievement of the correct final state.

Instruction—a group of bits that defines a computer operation and is part of the instruction set.

Instruction cycle—the process of fetching, decoding, and executing an instruction.

Instruction execution—the process of performing the operations indicated by an instruction.

Instruction execution time—the time required to fetch, decode, and execute an instruction.

Instruction fetch—the process of addressing memory and reading an instruction into the CPU for decoding and execution.

Instruction length—the amount of memory needed to store a complete instruction.

Instruction set—the set of general-purpose instructions available on a given computer—that is, the set of inputs to which the CPU will produce a known response during an instruction cycle.

Latch—a temporary storage device controlled by a timing signal. The contents of the latch are fixed at their current values by a transition of the timing signal (clock) and remain fixed until the next transition.

Linear select—using coded bus lines individually for selection purposes rather than decoding the lines. Linear select requires no decoders but allows only n separate devices to be connected rather than 2^n, where n is the number of lines.

Logic analyzer—a piece of test equipment that detects, stores, and displays the states of digital signals; usually has at least 8 and as many as 32 inputs.

Memory capacity—the total number of separate memory locations (usually specified in terms of bytes) that may be attached to a particular computer.

Multiplex—to use one functional unit for several different purposes on a shared basis, to interleave two or more different signals on the same channel.

Page—a subdivision of the memory. In 6502 terminology, a page is a 256-byte section of memory in which all addresses have the same page number (i.e., the same 8 most significant bits). For example, page C6 consists of memory addresses C600 through C6FF.

Page boundary—the boundary between one page of memory and another. A branch is said to cross a page boundary if the destination address is on a different page than the instruction that the processor would execute next in its normal (consecutive) sequence.

Page number—the identifier that characterizes a particular page of memory. In 6502 terminology, the 8 most significant bits of a memory address.

Paged address—the identifier that characterizes a particular memory address on a known page. In 6502 terminology, the 8 least significant bits of a memory address.

Refresh—the process of rewriting the contents of a dynamic memory before they are lost.

Tristate (or **three-state**)—logic outputs with three possible states—high, low, and an inactive (high-impedance or open-circuit) state that can be combined with other similar outputs in a busing structure.

Tristate enable—an input that, if not active, forces the outputs of a tristate device into the inactive or high-impedance state.

SPECIAL PROBLEMS IN MICROCOMPUTER HARDWARE DESIGN

Describing or investigating the flow of signals in a microcomputer is not simple. Not only does data move through the computer in parallel (typically 8 or 16 bits at a time), but the clock rate is high and there are few periodic sequences available for examination. These characteristics not only make meaningful laboratory experiments difficult to create, but they also make microprocessor-based products difficult to debug, maintain, and repair. In practice, many engineers prefer to purchase board-level computers rather than design their own boards and many companies prefer to replace entire circuit boards rather than have maintenance personnel attempt to identify the cause of a malfunction.

Of course, the designer must understand how a microcomputer operates and how its parts are connected. This laboratory can provide only the briefest overview of hardware design. We assume that you have a dual-trace oscilloscope with a bandwidth of at least 5 MHz. Unfortunately, even a good oscilloscope is usually inadequate for design or troubleshooting. To diagnose problems in system operation, you must be able to simultaneously examine the clock, data bus, address bus, and control signals. This requires an instrument that can display the states of many lines simultaneously in a comprehensible form. Test instruments called *logic analyzers* provide the required features, but they are expensive. We will content ourselves here with examining signals on a less expensive oscilloscope.

TIMING AND CONTROL FUNCTIONS

To design or understand a microcomputer, we must answer the following questions:

1) How does the processor transfer data to or from memory and I/O ports? Clearly timing is a critical factor here.

2) How does the processor decode and execute instructions? This is an internal function, but an understanding of it is important in microcomputer design, since the instruction cycle governs the operation of the computer.
3) How does the processor distinguish different types of cycles? The designer must use the signals that the processor provides to control external hardware and to monitor system operation.
4) How are particular memory addresses or I/O ports selected? The address lines and control signals must be decoded to select the correct device.
5) How does the busing structure allow many memories, input/output ports, and other devices to share the same buses? Most microprocessors have a tristate data bus; only one memory or input port is enabled at a time and the ones that are disabled are in the high-impedance state and do not affect the data bus.

The designer must use the microprocessor's timing and control signals to construct a microcomputer that will meet the requirements of a particular application. Factors that the designer must consider are cost, speed, expandability, consistency with other applications, testability, and ease of updating and maintenance.

SYSTEM CLOCK

Let us now look at some of the processor's signals on the oscilloscope. Figure F-1 and Tables F-1 and F-2 contain the pin assignments for the 6502 microprocessor, the KIM's Application Connector, and the KIM's Expansion Connector, respectively. Attach the oscilloscope ground to pin 22 of the Expansion Connector or to pin 1 of the Application Connector; either is a convenient ground point. +5 V is available on pin 21 of the Expansion Connector or on pin A of the Application Connector if you need it. Put your oscilloscope in the CHOP mode so that it will maintain the timing relationships rather than retriggering when you switch channels; do not use the ALTERNATE mode.

Attach one of your probes to pin U of the Expansion Connector (or to pin 3 of the 6502 CPU). This is one phase (ϕ_1) of the system clock (see Figure F-2) that controls the operations of the microprocessor. Remember that Figure F-1 contains the 6502's pin assignments. Attach

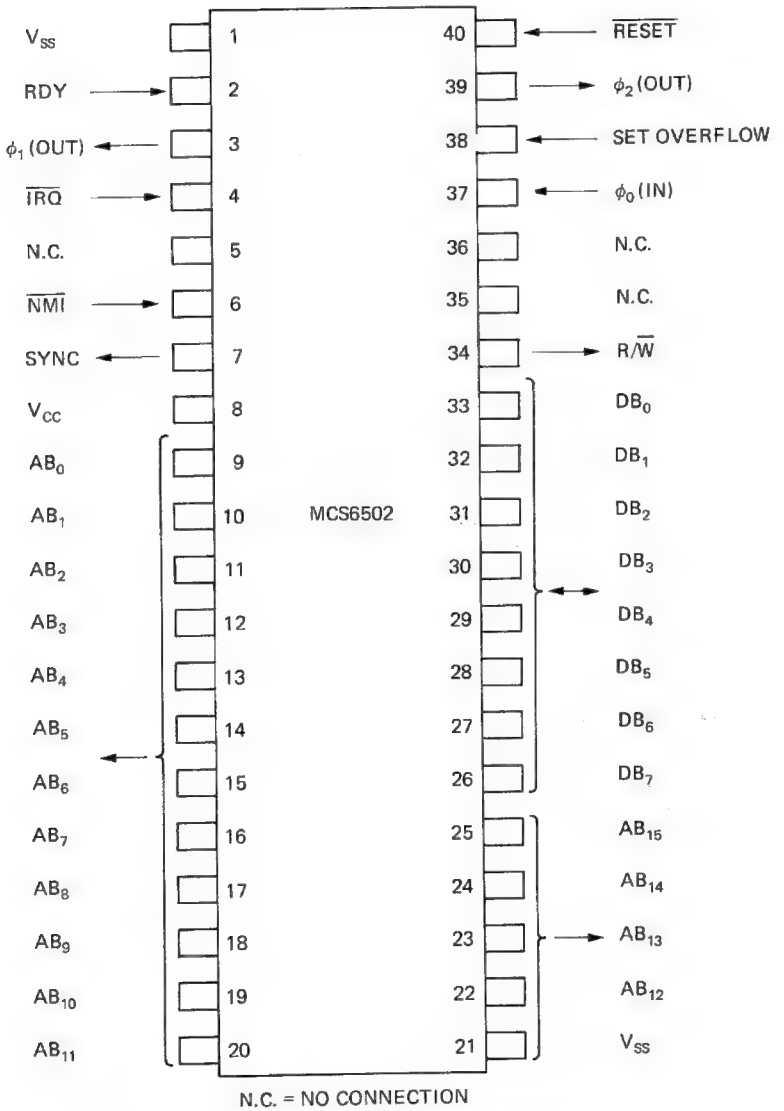


FIGURE F-1. Pin assignments for the 6502 microprocessor.

Table F-1

PIN ASSIGNMENTS FOR THE KIM'S APPLICATION CONNECTOR

22	KB Col D
21	KB Col A
20	KB Col E
19	KB Col B
18	KB Col F
17	KB Row 0
16	PB5
15	PB7
14	PA0
13	PB4
12	PB3
11	PB2
10	PB1
9	PB0
8	PA7
7	PA6
6	PA5
5	PA4
4	PA1
3	PA2
2	PA3
1	V_{SS} (GND)

Z	KB Row 1
Y	KB Col C
X	KB Row 2
W	KB Col G
V	KB Row 3
U	TTY PRINTER
T	TTY KEYBOARD
S	TTY PRINTER RETURN
R	TTY KEYBOARD RETURN
P	AUDIO OUT HIGH
N	+12V
M	AUDIO OUT LOW
L	AUDIO IN
K	DECODER ENABLE
J	K7
H	K5
F	K4
E	K3
D	K2
C	K1
B	K0
A	V_{CC} (+5V)

Table F-2

PIN ASSIGNMENTS FOR THE KIM'S EXPANSION CONNECTOR

22	V_{SS} (GND)
21	V_{CC} (+5V)
20	
19	
18	
17	SINGLE STEP OUT
16	K6
15	DB0
14	DB1
13	DB2
12	DB3
11	DB4
10	DB5
9	DB6
8	DB7
7	RESET
6	NMI
5	RO
4	IRQ
3	ϕ_1
2	RDY
1	SYNC

Z	$\overline{\text{RAM/R/W}}$
Y	ϕ_2
X	$\overline{\text{PLL}}$ TEST
W	$\overline{\text{R/W}}$
V	R/W
U	ϕ_2
T	AB15
S	AB14
R	AB13
P	AB12
N	AB11
M	AB10
L	AB9
K	AB8
J	AB7
H	AB6
F	AB5
E	AB4
D	AB3
C	AB2
B	AB1
A	AB0

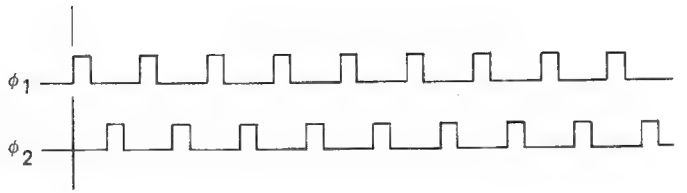
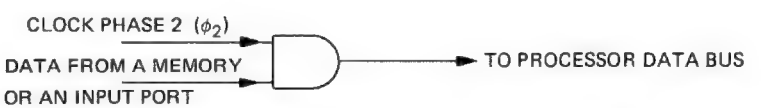


FIGURE F-2. Two-phase 6502 system clock.

your other probe to pin U of the Expansion Connector or to pin 39 of the 6502 CPU. This is the other phase (ϕ_2) of the clock. During phase 1, the processor places a new address on the address bus and determines the values of all the control signals. This phase is a setup period that is necessary because of the finite switching times of the devices that make up the computer. During phase 2, the processor actually transfers data to or from the memory or I/O ports. All control and address signals must be constant (stable) during phase 2.

Many devices must share the microcomputer's data bus. Memories, input ports, output ports, and the microprocessor itself must place data on the bus or receive data from it. Control signals must ensure that no more than one of the potential sources ever tries to put data on the bus at a given time. An attempt by more than one source to put data on the bus is known as *bus contention*. Bus contention can easily occur when the microprocessor changes its address outputs (that is, when it finishes reading one memory location or input port and begins reading another one). Since electronic devices have finite switching times, both the previous address and the current address will be active for a brief time after the change. The designer can eliminate bus contention during this period by gating the outputs from each memory or input port with phase 2 of the clock as shown in Figure F-3. Then no memory or input port can control the data bus except during phase 2. Remember, however, that the processor changes its address outputs during phase 1; by the time phase 2 begins, the new address will be stable and the old address will be inactive.



Note: The output of the gate is always 0 except when clock phase 2 is high.

FIGURE F-3. Gating phase 2 of the clock with data from a memory or an input port.

PROBLEM F-1

Determine the frequency and pulse width of both phases of the processor clock.

PROBLEM F-2

What are the minimum and maximum pulse widths that the input time base (crystal or *RC* network) must provide to operate the 6502 processor? These numbers are part of the processor's specifications.

TRACING INSTRUCTION EXECUTION

Place the following program in memory locations 0200 through 0202:

```
HERE    JMP    HERE
```

This instruction transfers control to itself, thus producing a repetitive pattern of signals. Program F-1 is the hexadecimal version.

PROGRAM F-1

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	4C	HERE	JMP	HERE
0201	00			
0202	02			

Attach one of your probes to phase 2 of the clock (pin 39 of the CPU or pin U of the Expansion Connector). Attach the other probe to SYNCHRONIZATION (SYNC), which is available either at pin 7 of the CPU or at pin 1 of the Expansion Connector. This signal goes high at the start of a cycle during which the processor is fetching an operation code from memory and stays high for the remainder of that cycle. Thus SYNC marks the beginning of each execution of the JMP instruction. Note that SYNC is high roughly one third of the time while Program F-1 is executing. Thus the processor is spending one third of its time fetching the operation code for JMP from memory and two thirds of its time fetching the destination address and performing internal operations.

Now attach your second probe to address line AB_0 (pin 9 of the CPU or pin A of the Expansion Connector). This line is high during the second clock cycle of JMP, when the processor is fetching the less signifi-

cant byte of the address from memory location 0201. Similarly, if you examine address line AB_1 (pin 10 of the CPU or pin B of the Expansion Connector), you will find that it goes high during the third clock cycle of the instruction, when the processor is fetching the more significant byte of the address from memory location 0202.

Note how the 6502 microprocessor executes instructions:

1) Each instruction is divided into a series of clock cycles that are used to transfer data to or from memory or I/O ports and execute internal operations.
2) Each clock cycle consists of one phase which is used to change and stabilize addresses and one phase which is used to transfer data.

Appendix A of the *MCS6500 Microcomputer Family Hardware Manual* describes the execution of all instructions in detail.

The 6502 executes a JMP instruction with absolute addressing in three clock cycles as follows:

1) During the first cycle, the CPU fetches the operation code (4C hex) and places it in the instruction register. The instruction register is inside the 6502 microprocessor and the programmer cannot access it. The processor fetches the operation code by placing the contents of the program counter (0200 hex) on the address bus and thus fetching the data from that address. The processor adds 1 to the program counter after each cycle in which it is used.

2) During the second cycle, the CPU fetches the less significant byte (00) of the destination address and places it in a temporary register. Here again, the processor places the contents of the program counter (now 0201 hex) on the address bus and thus fetches the data from that address. As in the first cycle, the processor adds 1 to the program counter, making its final value 0202.

3) During the third cycle, the CPU fetches the more significant byte (02) of the destination address and places it in another temporary register. Finally, the CPU loads the program counter from the two temporary registers, thus performing the required operation and completing the execution of JMP.

Can you identify an instruction cycle on the oscilloscope? Note that SYNC is high during the cycle in which the CPU is fetching the operation code.

PROBLEM F-3

Determine how long address line AB_0 remains high. How long does address line AB_1 remain high? Explain your result.

PROBLEM F-4

Measure how long SYNC remains high. Can you suggest some uses for this signal?

The CPU always reads data at the end of clock phase 2. The memory address is always stable before the end of clock phase 1. How much time does this allow to access the memory? The only way to allow more time is to reduce the clock frequency. Some processors have a READY input that forces them to wait additional clock cycles before they read the data. Such processors are easily interfaced to memories with long access times. The 6502 has a READY (RDY) input, but it examines that input only during cycles in which it is fetching an operation code. Thus the READY input can be used to implement a single-step mode (i.e., a halt after each instruction cycle) but not to force the processor to wait for a slow memory. Obviously, it would make little sense for the processor to wait for an operation code but not for other data from the same memory.

OTHER ADDRESSING MODES

PROGRAM F-2

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	EA	HERE	NOP	
0201	4C		JMP	HERE
0202	00			
0203	02			

Examine SYNC while Program F-2 is executing. How has it changed from Program F-1?

The CPU executes NOP in two cycles:

1) In the first cycle, the CPU reads the operation code from memory and places it in the instruction register. SYNC is high to indicate the operation code fetch.

2) In the second cycle, the CPU executes the instruction. SYNC is low since nothing is being fetched. The CPU simply ignores the memory.

PROBLEM F-5

What happens to address lines AB_0 and AB_1 during the execution of Program F-2? What happens if you place 01 in memory location 0202 instead of 00?

PROBLEM F-6

What does the processor place on the address bus during the second cycle of NOP? Remember the answer to Problem F-5. Can you think of some problems that this value might create? Can you tell whether the processor is actually reading the memory?

Place Program F-3 in memory and execute it.

PROGRAM F-3

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)	INSTRUCTION (MNEMONIC)		
0200	A9	HERE	LDA	#0
0201	00			
0202	4C		JMP	HERE
0203	00			
0204	02			

PROBLEM F-7

What happens to SYNC while Program F-3 is executing? Explain its appearance.

The 6502 microprocessor executes all instructions with immediate addressing in two clock cycles:

1) During the first cycle, the processor fetches the operation code from memory using the program counter. The operation code is placed in the instruction register and the program counter is incremented.
2) During the second cycle, the processor fetches the data from memory using the program counter. The operation is performed and the program counter is incremented again.

SYNC is high during the first cycle to indicate the operation code fetch.

PROBLEM F-8

What happens to SYNC if you replace the A9 (LDA immediate) in memory location 0200 with A5 (LDA zero page)? Explain the result. What instruction is the processor executing?

The 6502 microprocessor performs all register instructions with zero page (direct) addressing in three clock cycles as follows:

1) During the first cycle, the processor fetches the operation code from memory using the program counter. The operation code is placed in the instruction register and the program counter is incremented.

2) During the second cycle, the processor fetches the address on page zero from memory using the program counter. The address is stored in a temporary register and the program counter is incremented. Note that there are really two temporary registers—one for the eight MSBs of the address and one for the eight LSBs. In the zero page addressing mode, the processor clears the temporary register that holds the eight MSBs.

3) During the third cycle, the processor transfers the data to or from memory using the address in the temporary registers. The operation is performed and the processor is then ready to fetch the next instruction. Note that the program counter is not used in the third cycle and is therefore not incremented.

PROBLEM F-9

What happens to SYNC if you replace LDA #0 with LDA $0380? Describe the execution of a register instruction with absolute (direct) addressing in the same way that we described the execution of register instructions with immediate and zero page (direct) addressing.

PROBLEM F-10

What happens to SYNC if you replace LDA #0 with INC 0? Describe the execution of an INC instruction with zero page (direct) addressing.

Instructions that store data in memory must produce a signal that indicates the direction in which data is traveling and that can be used as a write pulse. The READ/WRITE line (CPU pin 34 or Expansion Connector pin V) serves this purpose. In the programs we have run so far, this line should always be in the READ state (logic 1). Examine the READ/WRITE line during the execution of the program with LDA #0 and verify this.

PROBLEM F-11

What happens to SYNC and R/$\overline{W}$ if you replace the A9 in memory location 0200 with 85? What instruction is now in memory locations 0200 and 0201? Demonstrate that your answer is correct by loading the accumulator and memory location 0000 initially and showing that the instruction has the expected effects.

In relative addressing, the processor must add the relative offset to the program counter to obtain the destination address. The 6502 is designed to minimize the execution time of conditional branches that use relative addressing. It does this by not performing any calculation at all if the branch condition is false and by performing a 16-bit calculation only if the branch condition is true and the branch crosses a page boundary.

Enter Program F-4 into memory. The CLC instruction is executed only once; it simply ensures that the branch condition is true. Examine the SYNC line and determine how many cycles are required to execute the BCC instruction. Here the branch is from address 0203 to address 0201; no page boundary is crossed.

PROGRAM F-4

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	18		CLC	
0201	90	HERE	BCC	HERE
0202	FE			

PROBLEM F-12

What happens to SYNC if you move Program F-4 to memory addresses 02FE through 0300? How many cycles are required to execute BCC now? Explain the difference.

PROBLEM F-13

Change the instruction in memory address 0201 from BCC (90) to BCS (B0). Put the instruction BCC HERE (90 followed by FC) in memory locations 0203 and 0204. Examine SYNC and determine how many cycles are required to execute BCS and BCC. Explain why the numbers are different. What happens if you move the program to memory addresses 02FE through 0302?

DECODING ADDRESS LINES

The KIM microcomputer decodes address lines AB_{12}, AB_{11}, and AB_{10} as described in Table F-3. A 74145 BCD-to-decimal decoder (see Figure F-4 and Table F-4) performs the required logic function; the decoder is integrated circuit U4 on the KIM board (not to be confused with U24, which is used to control the on-board LED displays). Execute Program F-3 and examine line K0 (pin 1 of the decoder or pin B of the Application Connector). What happens to K0? Note that all the instructions in Program F-3 are being executed from the 1K user RAM. K0 (decoder pin 1) is the output signal used to enable (activate) the 1K user RAM.

PROBLEM F-14

What happens to K0 if you place Program F-3 in memory addresses 1780 through 1784? Remember to change the destination address in JMP to 1780. What happens to K5 (pin 5 of the decoder or pin H of the Application Connector)? What happens to K0 and K5 if you execute the following program?

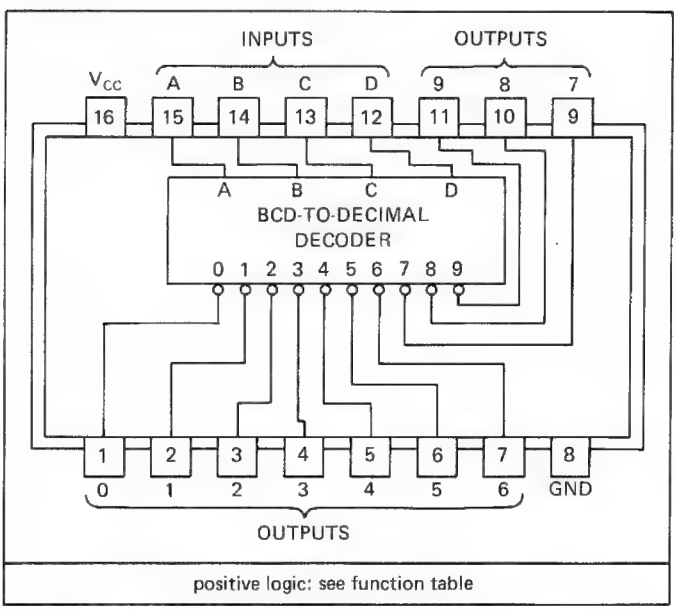


FIGURE F-4. Pin assignments for the 74145 BCD-to-decimal decoder/driver.

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	AD	HERE	LDA	$1780
0201	80			
0202	17			
0203	4C		JMP	HERE
0204	00			
0205	02			

Table F-3

KIM ADDRESS DECODING

AB_{12}	AB_{11}	AB_{10}	SYMBOL	DEVICE ACTIVATED
0	0	0	K_0	1K user RAM
0	0	1	K_1	Not used, available for expansion
0	1	0	K_2	Not used, available for expansion
0	1	1	K_3	Not used, available for expansion
1	0	0	K_4	Not used, available for expansion
1	0	1	K_5	RAM and timers on 6530 devices
1	1	0	K_6	ROM on keyboard/display 6530 device
1	1	1	K_7	ROM on user 6530 device

Table F-4

FUNCTION TABLE FOR THE 74145 BCD-TO-DECIMAL DECODER/DRIVER*

NO.	INPUTS				OUTPUTS									
	D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	L	L	L	L	L	H	H	H	H	H	H	H	H	H
1	L	L	L	H	H	L	H	H	H	H	H	H	H	H
2	L	L	H	L	H	H	L	H	H	H	H	H	H	H
3	L	L	H	H	H	H	H	L	H	H	H	H	H	H
4	L	H	L	L	H	H	H	H	L	H	H	H	H	H
5	L	H	L	H	H	H	H	H	H	L	H	H	H	H
6	L	H	H	L	H	H	H	H	H	H	L	H	H	H
7	L	H	H	H	H	H	H	H	H	H	H	L	H	H
8	H	L	L	L	H	H	H	H	H	H	H	H	L	H
9	H	L	L	H	H	H	H	H	H	H	H	H	H	L
Invalid	H	L	H	L	H	H	H	H	H	H	H	H	H	H
	H	L	H	H	H	H	H	H	H	H	H	H	H	H
	H	H	L	L	H	H	H	H	H	H	H	H	H	H
	H	H	L	H	H	H	H	H	H	H	H	H	H	H
	H	H	H	L	H	H	H	H	H	H	H	H	H	H
	H	H	H	H	H	H	H	H	H	H	H	H	H	H

*H, high level (off); L, low level (on).

PROBLEM F-15

The following program uses the seven-segment code table in the monitor (starting at address 1FE7). This table is in the ROM on the user 6530 device.

```
        LDX   #0          ;DIGIT = ZERO
HERE    LDA   $1FE7,X     ;CONVERT TO SEVEN-SEGMENT CODE
        JMP   HERE
```

In hexadecimal, this is

MEMORY ADDRESS (HEX)	MEMORY CONTENTS (HEX)		INSTRUCTION (MNEMONIC)	
0200	A2		LDX	#0
0201	00			
0202	BD	HERE	LDA	$1FE7,X
0203	E7			
0204	1F			
0205	4C		JMP	HERE
0206	02			
0207	02			

Examine K7 (pin 9 of the decoder or pin J of the Application Connector). This is the output signal used to activate the ROM on the user 6530 device (starting at address 1C00). How long is this signal active? Describe the execution of a register instruction using absolute indexed addressing. Describe the behavior of SYNC.

PROBLEM F-16

Replace LDA $1FE7, X with LDA $1780, X. Are there any changes on lines K5 or K7? Explain what has happened.

MULTIPLE ADDRESSES AND MEMORY EXPANSION

The 6502's three most significant address lines (AB_{13}, AB_{14}, and AB_{15}) are not even connected to the KIM's on-board decoder. The result is that the decoding of the on-board memory does not depend on the values of those three lines. After all, signals that are not connected to the decoder cannot affect its operation. Thus each on-board memory location actually responds to 8 addresses, one for each possible value of AB_{13}, AB_{14}, and AB_{15}. This redundancy seems odd, but it does not cause any operating

problems, any more than the practice of giving a large building several different street addresses (corresponding to different entrances) causes problems in mail delivery. What causes operating problems is not one location responding to several different addresses, but rather one address activating several different locations. That causes bus contention if all the locations try to control the data bus at the same time.

Multiple addresses, however, may result in errors if the programmer or designer is not aware that all the addresses are occupied and that different addresses may actually refer to the same memory location. For example, load Program F-1 into memory locations 0200 through 0202. Now examine memory locations 2200 through 2202. Change memory location 0200 to AA. What happens to memory location 2200?

PROBLEM F-17

> What other memory addresses actually refer to the same location as 0200 and 2200? This situation in which a memory location occupies multiple addresses is often referred to as a *foldback*, since the memory appears to be arranged like a piece of paper or cloth that is folded back over itself.

Thus one problem with multiple addresses is that the programmer may try to use them for different purposes, not realizing that they are actually the same memory location. A more serious problem occurs if the designer tries to add memory to the computer. For example, one might expect the on-board KIM memory to occupy only the lowest 8K of the 6502's address space and that one could add another 56K of memory externally. In fact, however, the on-board KIM memory occupies all 64K of the 6502's address space. You cannot add any memory externally unless you add a connection that disables the on-board memory whenever the external memory is being used. Otherwise, both the on-board memory and the external memory will be activated and bus contention will occur. Figure F-5 (taken from the *KIM-1 User Manual*) shows an expansion scheme that not only decodes AB_{13}, AB_{14}, and AB_{15} but also directs the lowest signal ($8K_0$) back into the KIM to activate the on-board memory. Now each on-board memory location occupies a single address with $AB_{13} = AB_{14} = AB_{15} = 0$.

There is an obvious tradeoff here. Decoding all the address lines allows us to attach the maximum amount of memory, thus facilitating expansion. On the other hand, it also requires additional parts (decoders). If our application is a simple one that does not require a large amount of memory, the additional parts represent extra cost with no compensating benefit. Thus designers typically decode all the address lines in large systems or systems that might be expanded substantially (such as a home computer, a graphics processor, or an industrial robot). On the other hand, designers typically leave address lines undecoded in small systems or

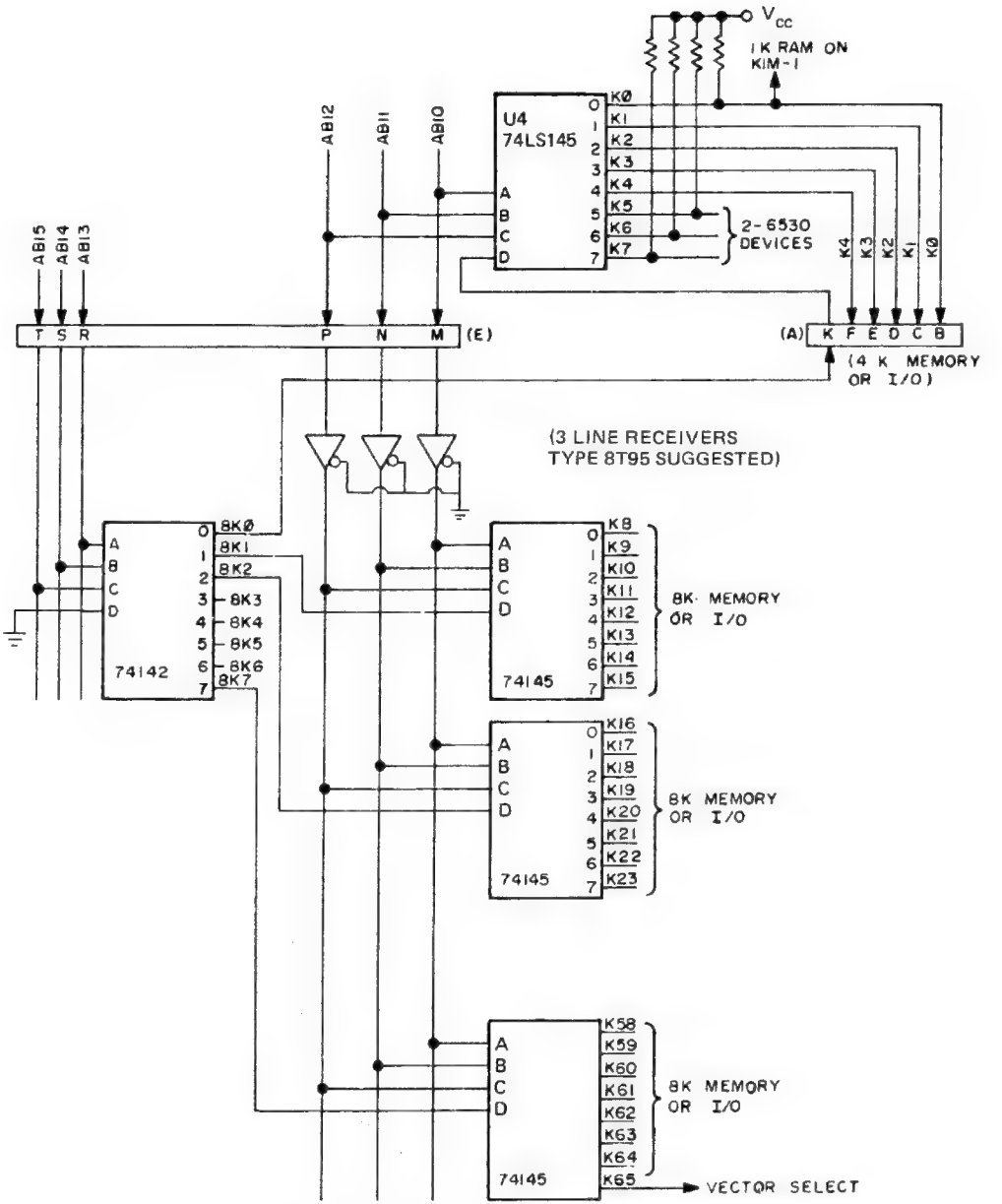


FIGURE F-5. Memory expansion scheme for the KIM-1 microcomputer. (Courtesy of Commodore/MOS Technology, Inc.)

systems that will never be expanded (such as a low-cost printer, a hand-held electronic game, or a simple voltmeter).

PROBLEM F-18

In Figure F-5, each output signal from the decoder at the left corresponds to 8K bytes of address space. Remember that the entire address space is 64K and the decoder is dividing it into eight equal pieces. The decoders at the right further divide 8K of address space into 1K units. Assume that our memory is divided into units of 1K and that we will not subdivide 8K of address space further unless we must. That is, if we have enough address space available, we will simply allow a 1K unit of memory to occupy 8K of address space. What is the minimum number of external decoders that we need if we plan to add 7K of memory? Construct a table showing the minimum number of decoders we need to add amounts of memory ranging from 1K to 56K.

ADDRESSING I/O DEVICES

Even when fully implemented, the decoding scheme in Figure F-5 divides the address space only into 1K sections. This still leaves us with the problem of addressing individual I/O devices. For example, if we are using 6520 PIAs (each of which occupies four memory addresses), 1K of address space could hold 256 devices, far more than most microcomputers need.

As long as we have so much address space, we do not need to decode it fully. If, for example, we tried to fully decode 256 PIAs with 74145 devices, we would need at least 32 chips (since each 74145 has eight outputs), not even considering the additional gates and control signals that such a large decoding system would require. We can, however, avoid the need for any decoders at all by simply using each address line to select a PIA. Figure F-6 shows the pin assignments for a 6520 PIA. The RS (register select) lines decode the internal registers of the PIA and are normally tied to address lines AB_0 and AB_1. The CS (chip select) lines can be used to select a particular PIA. One line could, for example, be tied to the output signal from one of the 74145's shown in Figure F-5, so that the PIA is selected only when that signal is active. Another could then be tied directly to an undecoded address line (AB_2 through AB_9 are still available). This kind of decoding (called *linear select*) lets us attach a total of 8 PIAs (remember that address lines AB_{10} through AB_{15} are decoded in Figure F-5 and AB_0 and AB_1 are used to select internal PIA registers). Eight PIAs are more than enough for most small systems, since each PIA has 20 I/O lines (two 8-bit I/O ports and four control lines).

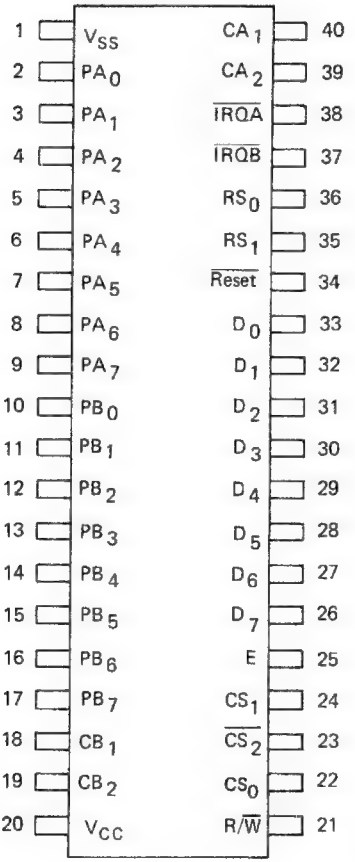


FIGURE F-6. Pin assignments for the 6520 Peripheral Interface Adapter (PIA).

PROBLEM F-19

A 6522 Versatile Interface Adapter occupies 16 memory addresses (i.e., address lines AB_0 through AB_3 are normally used for internal decoding). How many 6522 devices could we place in 1K of address space using linear select? How much address space would we need to set aside if we wanted to decode eight 6522 devices using linear select?

PROBLEM F-20

One problem with linear select is that it results in a curious, discontinuous set of addresses. Each 1 bit in the selection lines activates an I/O device, so only addresses with one 1 bit in those lines are occupied. Which addresses do PIAs occupy if we attach 8 of them using linear select and activate all 8 of them with

signal K20 from Figure F-5? That is, we tie K20 to a chip select line on each PIA.

PROBLEM F-21

An interesting (and occasionally useful) outgrowth of the discontinuous addresses in Problem F-20 is that we can actually send data to more than one PIA at a time by storing the data in an address with 1 bits in all the required selection lines. This multiple transmission is sometimes called a broadcast, since the analogy to a general broadcast on a communications network is obvious. Assuming that we have eight PIAs addressed as in Problem F-20, write a program that clears all the control registers, makes all the A ports inputs and all the B ports outputs, and then stores 04 (hex) in all the control registers. Assume that, if BASE is the lowest address occupied by a PIA, the registers occupy the following addresses:

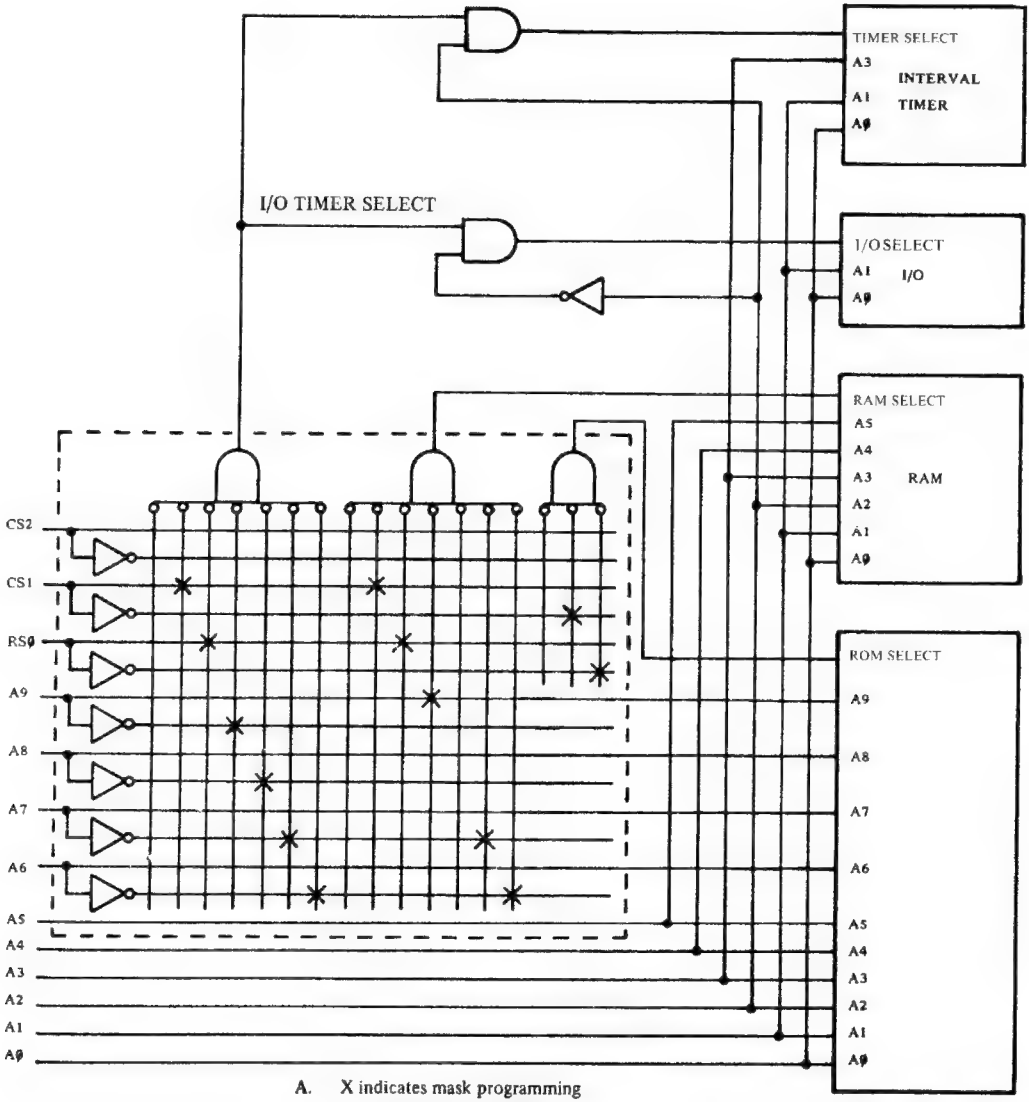
BASE:	Data or data direction register A
BASE+1:	Control register A
BASE+2:	Data or data direction register B
BASE+3:	Control register B

We have not described the addressing of 6530 devices here, since these devices include both memory and I/O ports. Fitting a 6530 device into the processor's address space is much like finding a place on a shelf for an item shaped like a wedding cake. After all, the device has 1K bytes of ROM, 64 bytes of RAM, two 8-bit I/O registers, two data direction registers, and two registers associated with the 8-bit timer. Another complicating factor is that some of the address lines are used to set the divide ratio and the interrupt enable for the timer (remember Laboratory D). The manufacturer has simplified the addressing of 6530 devices by providing optional on-chip decoding for the ROM, RAM, I/O ports, and timer. The user specifies the internal decoding connections together with the contents of the ROM. Figure F-7 shows the decoding diagram for a 6530 device along with the connections specified by a particular user; the X's indicate connections that the user wants the manufacturer to make as part of the final production step (known as *masking*).

KEY POINT SUMMARY

1) A logic analyzer is necessary to fully understand or debug the hardware in microprocessor-based systems. The analyzer can display the states of many simultaneous signals in a comprehensible format.

2) The 6502 microprocessor executes its instructions as a series of clock cycles during which it transfers data to or from memory and performs internal operations.



A. X indicates mask programming
i.e. ROM select = $\overline{CS1}$•RSO
RAM select = $\overline{CS1}$•$\overline{RSO}$•$\overline{A9}$•A7•A6
I/O TIMER SELECT = $\overline{CS1}$•$\overline{RSO}$•A9•A8•A7•A6

B. Notice that A8 is a don't care for
RAM select

C. CS2 can be used as PB5 in this example.

FIGURE F-7. Address decoding diagram for the 6530 Peripheral Interface/Memory device. (Courtesy of Commodore/MOS Technology, Inc.)

3) The 6502 microprocessor differentiates between operation code fetches and other cycles by means of the SYNCHRONIZATION (SYNC) signal. This signal is high when the processor is fetching an operation code from memory.

4) The execution of an instruction involves at least two clock cycles. During the first cycle, the CPU fetches an operation code by placing the program counter on the address bus and loading the contents of the accessed memory location into the instruction register. The CPU adds 1 to the program counter each time it is used.

5) The various addressing modes result in different methods of instruction execution. In immediate addressing, the CPU uses the program counter to fetch the data. In zero page and absolute (direct) addressing, the CPU uses the program counter to fetch the address of the data and then employs that address in a subsequent memory access. In indexed and relative addressing, the processor must perform an addition to calculate the effective address. The designers of the 6502 optimized the speed with which it executes instructions by having it perform full 16-bit operations (taking an extra clock cycle) only if it must. That is, the processor adds the less significant bytes and deals with the more significant byte only if there is a carry. In the case of relative branches, the processor does not perform the calculation at all if the condition does not hold and no branch will occur.

6) The more significant address lines are generally decoded to form enabling signals. These signals allow particular memories or I/O devices to send or receive data. In general, only one memory or I/O device can send or receive data at a time.

7) The designer can make tradeoffs between the memory capacity of the microcomputer and the complexity of the decoding system. Full decoding of addresses maximizes memory capacity but increases parts count. Partial decoding of addresses is often sufficient in small systems.

8) I/O addresses are seldom decoded fully in microcomputers because few applications require even a substantial fraction of the available capacity. Linear select (using an address line directly to select a particular I/O device) is a convenient way to provide a reasonable I/O capacity without using any decoders.

9) Devices that combine functions, such as ROM, RAM, I/O ports, and timers, are often convenient in small systems because they reduce the total parts count. They may, however, be difficult to place in the address space in a way that permits easy expansion while maintaining continuous addresses.

Appendixes

APPENDIX 1—6502 INSTRUCTION SET*

Table A1-1

MICROPROCESSOR OPERATION CODES AND OBJECT CODES

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM			IMPLIED			(IND, X)			(IND), Y			Z PAGE, X			ABS, X			ABS, Y			RELATIVE			INDIRECT			Z PAGE, Y			PROCESSOR STATUS CODES	
MNEMONIC	OPERATION	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V • B D I Z C	MNEMONIC
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										N V • • • • Z C	ADC
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										N • • • • • Z •	AND
ASL	C ← 7 6 5 4 3 2 1 0 ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													N • • • • • Z C	ASL
BCC	BRANCH ON C = 0 (2)																															90	2	2								• • • • • • • •	BCC
BCS	BRANCH ON C = 1 (2)																															B0	2	2								• • • • • • • •	BCS
BEQ	BRANCH ON Z = 1 (2)																															F0	2	2								• • • • • • • •	BEQ
BIT	A∧M				2C	4	3	24	3	2																																M_7 M_6 • • • • Z •	BIT
BMI	BRANCH ON N = 1 (2)																															30	2	2								• • • • • • • •	BMI
BNE	BRANCH ON Z = 0 (2)																															D0	2	2								• • • • • • • •	BNE
BPL	BRANCH ON N = 0 (2)																															10	2	2								• • • • • • • •	BPL
BRK	BREAK (See Fig.)																																								• • • 1 • 1 • •	BRK
BVC	BRANCH ON V = 0 (2)																															50	2	2								• • • • • • • •	BVC
BVS	BRANCH ON V = 1 (2)																															70	2	2								• • • • • • • •	BVS
CLC	0 → C																18	2	1																						• • • • • • • 0	CLC
CLD	0 → D																D8	2	1																						• • • • 0 • • •	CLD
CLI	0 → I																58	2	1																						• • • • • 0 • •	CLI
CLV	0 → V																B8	2	1																						• 0 • • • • • •	CLV
CMP	A - M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										N • • • • • Z C	CMP
CPX	X - M	E0	2	2	EC	4	3	E4	3	2																																N • • • • • Z C	CPX
CPY	Y - M	C0	2	2	CC	4	3	C4	3	2																																N • • • • • Z C	CPY
DEC	M - 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													N • • • • • Z •	DEC
DEX	X - 1 → X													CA	2	1																									N • • • • • Z •	DEX
DEY	Y - 1 → Y													88	2	1																									N • • • • • Z •	DEY
EOR	A ∨ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										N • • • • • Z •	EOR
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													N • • • • • Z •	INC
INX	X + 1 → X													E8	2	1																									N • • • • • Z •	INX
INY	Y + 1 → Y													C8	2	1																									N • • • • • Z •	INY
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3				• • • • • • • •	JMP
JSR	JUMP SUB (See Fig. A1-2)				20	6	3																																		• • • • • • • •	JSR
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										N • • • • • Z •	LDA
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							86	4	2	N • • • • • Z •	LDX
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													N • • • • • Z •	LDY
LSR	0 → 7 6 5 4 3 2 1 0 → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0 • • • • • Z C	LSR
NOP	NO OPERATION													EA	2	1																									• • • • • • • •	NOP
ORA	A ∨ M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										N • • • • • Z •	ORA
PHA	A → Ms S - 1 → S													48	3	1																									• • • • • • • •	PHA
PHP	P → Ms S - 1 → S													08	3	1																									• • • • • • • •	PHP
PLA	S + 1 → S Ms → A													68	4	1																									N • • • • • Z •	PLA
PLP	S + 1 → S Ms → P													28	4	1																									(RESTORED)	PLP
ROL	7 6 5 4 3 2 1 0 ← C				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													N • • • • • Z C	ROL
ROR	C → 7 6 5 4 3 2 1 0				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													N • • • • • Z C	ROR
RTI	RTRN INT (See Fig. A1-1)													40	6	1																									(RESTORED)	RTI
RTS	RTRN SUB (See Fig. A1-2)													60	6	1																									• • • • • • • •	RTS
SBC	A - M - C → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V • • • • Z (3)	SBC
SEC	1 → C													38	2	1																									• • • • • • • 1	SEC
SED	1 → D													F8	2	1																									• • • • 1 • • •	SED
SEI	1 → I													78	2	1																									• • • • • 1 • •	SEI
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										• • • • • • • •	STA
STX	X → M				8E	4	3	86	3	2																													96	4	2	• • • • • • • •	STX
STY	Y → M				8C	4	3	84	3	2													94	4	2																	• • • • • • • •	STY
TAX	A → X													AA	2	1																									N • • • • • Z •	TAX
TAY	A → Y													A8	2	1																									N • • • • • Z •	TAY
TSX	S → X													BA	2	1																									N • • • • • Z •	TSX
TXA	X → A													8A	2	1																									N • • • • • Z •	TXA
TXS	X → S													9A	2	1																									• • • • • • • •	TXS
TYA	Y → A													98	2	1																									N • • • • • Z •	TYA

(1) ADD 1 to "N" IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE, Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X	INDEX X	+	ADD
Y	INDEX Y	-	SUBTRACT
A	ACCUMULATOR	∧	AND
M	MEMORY PER EFFECTIVE ADDRESS	∨	OR
Ms	MEMORY PER STACK POINTER	∨	EXCLUSIVE OR
M_7	MEMORY BIT 7		
M_6	MEMORY BIT 6		
n	NO. CYCLES		
#	NO. BYTES		

*Courtesy of Commodore/MOS Technology, Inc., Norristown, PA.

Table A1-2

MNEMONIC EQUIVALENTS OF HEXADECIMAL OPERATION CODES

LSD / MSD	0	1	2	3	4	5	6	7
0	BRK	ORA-IND, X				ORA Z PAGE	ASL Z PAGE	
1	BPL	ORA-IND, Y				ORA Z PAGE, X	ASL Z PAGE, X	
2	JSR	AND-IND, X			BIT Z Page	AND Z PAGE	ROL Z PAGE	
3	BMI	AND-IND, Y				AND Z PAGE, X	ROL Z PAGE, X	
4	RTI	EOR-IND, X				EOR Z PAGE	LSR Z PAGE	
5	BVC	EOR-IND, Y				EOR Z PAGE, X	LSR Z PAGE, X	
6	RTS	ADC-IND, X				ADC Z PAGE	ROR Z PAGE	
7	BVS	ADC-IND, Y				ADC Z PAGE, X		
8		STA-IND, X			STY Z Page	STA Z PAGE	STX Z PAGE	
9	BCC	STA IND, Y			STY Z Page, X	STA Z PAGE, X	STX Z PAGE, Y	
A	LDY-IMM	LDA-IND, X	LDX IMM		LDY Z Page	LDA Z PAGE	LDX Z PAGE	
B	BCS	LDA IND, Y			LDY Z Page, X	LDA Z PAGE, X	LDX Z PAGE, Y	
C	CPY-IMM	CMP-IND, X			CPY Z Page	CMP Z PAGE	DEC Z PAGE	
D	BNE	CMP IND, Y				CMP Z PAGE, X	DEC Z PAGE, X	
E	CPX-IMM	SBC-IND, X			CPX Z Page	SBC Z PAGE	INC Z PAGE	
F	BEQ	SBC IND, Y				SBC Z PAGE, X	INC Z PAGE, X	

8	9	A	B	C	D	E	F	LSD / MSD
PHP	ORA IMM	ASL A			ORA ABS	ASL ABS		0
CLC	ORA ABS, Y				ORA ABS, X	ASL ABS, X		1
PLP	AND IMM	ROL A		BIT ABS	AND ABS	ROL ABS		2
SEC	AND ABS, Y				AND ABS, X	ROL ABS, X		3
PHA	EOR IMM	LSR A		JUMP ABS	EOR ABS	LSR ABS		4
CLI	EOR ABS, Y				EOR ABS, X	LSR ABS, X		5
PLA	ADC IMM	ROR A		JMP IND	ADC ABS	ROR ABS		6
SEI	ADC ABS, Y				ADC ABS, X			7
DEY		TXA		STY ABS	STA ABS	STX ABS		8
TYA	STA ABS, Y	TXS			STA ABS, X			9
TAY	LDA IMM	TAX		LDY ABS	LDA ABS	LDX ABS		A
CLV	LDA ABS, Y	TSX		LDY ABS, X	LDA ABS, X	LDX ABS, Y		B
INY	CMP IMM	DEX		CPY ABS	CMP ABS	DEC ABS		C
CLD	CMP ABS, Y				CMP ABS, X	DEC ABS, X		D
INX	SBC IMM	NOP		CPX ABS	SBC ABS	INC ABS		E
SED	SBC ABS, Y				SBC ABS, X	INC ABS, X		F

ADDRESSING MODES

IMM — IMMEDIATE ADDRESSING — The operand is contained in the second byte of the instruction.

ABS — ABSOLUTE ADDRESSING — The second byte of the instruction contains the 8 low order bits of the effective address (EA). The third byte contains the 8 high order bits of the effective address.

Z PAGE — ZERO PAGE ADDRESSING — Second byte contains the 8 low order bits of the effective address. The 8 high order bits are zero.

A — ACCUMULATOR — One byte instruction operating on the accumulator.

Z PAGE, X - Z PAGE, Y — ZERO PAGE INDEXED — The second byte of the instruction is added to the index (carry is dropped) to form the low order byte of the EA. The high order byte of the EA is zero.

ABS, X - ABS, Y — ABSOLUTE INDEXED — The effective address is formed by adding the index to the second and third byte of the instruction.

(IND, X) — INDEXED INDIRECT — The second byte of the instruction is added to the X index, discarding the carry. The result points to a location on page zero which contains the 8 low order bits of the EA. The next byte contains the 8 high order bits.

(IND), Y — INDIRECT INDEXED — The second byte of the instruction points to a location in page zero. The contents of this memory location is added to the Y index, the result being the low order eight bits of the EA. The carry from this operation is added to the contents of the next page zero location, the result being the 8 high order bits of the EA.

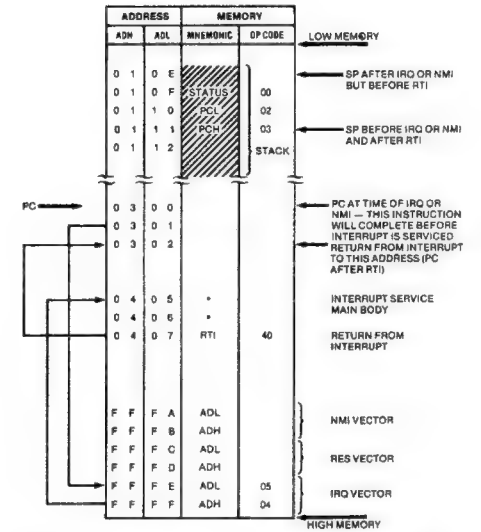


FIGURE A1-1. Response to $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ inputs and operation of the RTI and BRK instructions.

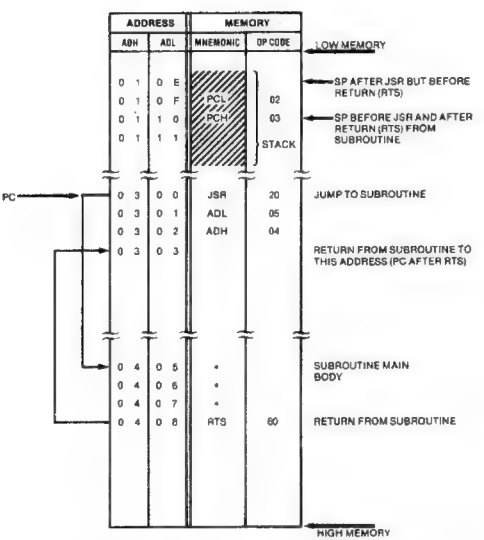


FIGURE A1-2. Operation of the JSR and RTS instructions.

Table A1-3

6502 MICROPROCESSOR INSTRUCTION SET—ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus (Negative)
BNE	Branch on Result Not Zero
BPL	Branch on Result Plus (Positive)
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode Flag
CLI	Clear Interrupt Disable Flag
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index Register X
CPY	Compare Memory and Index Register Y
DEC	Decrement Memory by One
DEX	Decrement Index Register X by One
DEY	Decrement Index Register Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index Register X by One
INY	Increment Index Register Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index Register X with Memory
LDY	Load Index Register Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode Flag
SEI	Set Interrupt Disable Flag
STA	Store Accumulator in Memory
STX	Store Index Register X in Memory
STY	Store Index Register Y in Memory
TAX	Transfer Accumulator to Index Register X
TAY	Transfer Accumulator to Index Register Y
TSX	Transfer Stack Pointer to Index Register X
TXA	Transfer Index Register X to Accumulator
TXS	Transfer Index Register X to Stack Pointer
TYA	Transfer Index Register Y to Accumulator

Table A1-4

SUMMARY OF 6502 ADDRESSING MODES

Accumulator addressing. This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

Immediate addressing. In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

Absolute addressing. In absolute addressing, the second byte of the instruction specifies the 8 low-order bits of the effective address while the third byte specifies the 8 high-order bits. Thus the absolute addressing mode allows access to the entire 64K bytes of addressable memory.

Zero page addressing. The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in a significant increase in code efficiency.

Indexed zero page addressing (X, Y indexing). This form of addressing is used in conjunction with an index register and is referred to as "Zero Page, X" or "Zero Page, Y." The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally, due to the "Zero Page" addressing nature of this mode, no carry is added to the high-order 8 bits of the effective address, so it is always in page zero.

Indexed absolute addressing (X, Y indexing). This form of addressing is used in conjunction with index register X or Y and is referred to as "Absolute, X" and "Absolute, Y." The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

Implied addressing. In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

Relative addressing. Relative addressing is used only with branch instructions and establishes a destination for the conditional branch. The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower 8 bits of the program counter when the counter is set at the next instruction. The range of the offset is -128 to +127 bytes from the next instruction.

Indexed indirect addressing. In indexed indirect addressing (referred to as (Indirect,X)), the second byte of the instruction is added to the contents of index register X, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low-order 8 bits of the effective address. The next memory location in page zero contains the high-order 8 bits of the effective address. Both memory locations specifying the high- and low-order bytes of the effective address must be in page zero.

Indirect indexed addressing. In indirect indexed addressing [referred to as (Indirect), Y] the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of index register Y, the result being the low-order 8 bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high-order 8 bits of the effective address.

Absolute indirect. The second byte of the instruction contains the low-order 8 bits of a memory address. The high-order 8 bits of that memory address is contained in the third byte of the instruction. The contents of the fully specified memory location is the low-order byte of the effective address. The next memory location contains the high-order byte of the effective address, which is loaded into the 16 bits of the program counter.

APPENDIX 2—ASCII CODE TABLE

HEX-ASCII TABLE

00	NUL		21	!		42	B		63	c	
01	SOH		22	"		43	C		64	d	
02	STX		23	#		44	D		65	e	
03	ETX		24	$		45	E		66	f	
04	EOT		25	%		46	F		67	g	
05	ENQ		26	&		47	G		68	h	
06	ACK		27	'		48	H		69	i	
07	BEL		28	(		49	I		6A	j	
08	BS		29	)		4A	J		6B	k	
09	HT		2A	*		4B	K		6C	l	
0A	LF		2B	+		4C	L		6D	m	
0B	VT		2C	,		4D	M		6E	n	
0C	FF		2D	-		4E	N		6F	o	
0D	CR		2E	.		4F	O		70	p	
0E	SO		2F	/		50	P		71	q	
0F	SI		30	0		51	Q		72	r	
10	DLE		31	1		52	R		73	s	
11	DC1	(X-ON)	32	2		53	S		74	t	
12	DC2	(TAPE)	33	3		54	T		75	u	
13	DC3	(X-OFF)	34	4		55	U		76	v	
14	DC4	~~(TAPE)~~	35	5		56	V		77	w	
15	NAK		36	6		57	W		78	x	
16	SYN		37	7		58	X		79	y	
17	ETB		38	8		59	Y		7A	z	
18	CAN		39	9		5A	Z		7B	{	
19	EM		3A	:		5B	[		7C	|	
1A	SUB		3B	;		5C	\		7D	}	
1B	ESC		3C	<		5D	]				(ALT MODE)
1C	FS		3D	=		5E	^	(↑)	7E	~	
1D	GS		3E	>		5F	—	(←)	7F	DEL	
1E	RS		3F	?		60	`				(RUB OUT)
1F	US		40	@		61	a				
20	SP		41	A		62	b				

APPENDIX 3–BRIEF DESCRIPTIONS OF 6502 FAMILY DEVICES

The following specification sheets are from the Synertek SY6500 Microprocessor Products Reference Data, Synertek Inc., Santa Clara, CA. Copyright 1979 by Synertek Inc. All rights reserved. No part of this material may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Synertek. Reproduced by permission.

SY6500 MICROPROCESSORS

The SY6500 Microprocessor Family Concept–

The SY6500 Series Microprocessors represent the first totally software compatible microprocessor family. This family of products includes a range of software compatible microprocessors which provide a selection of addressable memory range, interrupt input options, and on-chip clock oscillators and drivers. All of the microprocessors in the SY6500 group are software compatible within the group and are bus compatible with the M6800 product offering.

The family includes five 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 versions of the microprocessors are available in 1 MHz and 2 MHz ("A" suffix on product numbers) maximum operating frequencies.

Features of the SY6500 Family

- Single five volt 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
- Nonmaskable interrupt
- Use with any type or speed memory
- Bidirectional Data Bus
- Instruction decoding and control
- Addressable memory range of up to 64K bytes
- "Ready" input
- Direct memory access capability
- Bus compatible with MC6800
- Choice of external or on-board clocks
- 1MHz and 2MHz operation
- On-the-chip clock options
  * External single clock input
  * RC time base input
  * Crystal time base input
- 40 and 28 pin package versions
- Pipeline architecture

Members of the Family

Microprocessors with On-Board Clock Oscillator

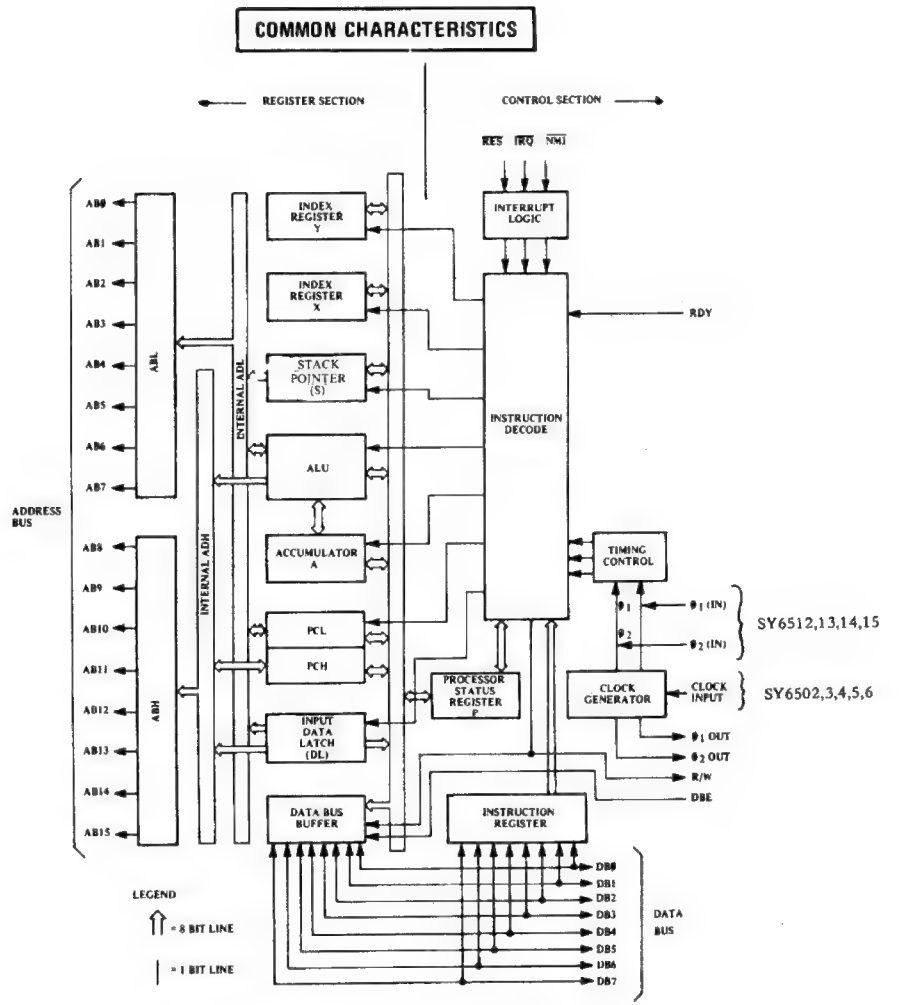
- SY6502
- SY6503
- SY6504
- SY6505
- SY6506

Microprocessors with External Two Phase Clock Input

- SY6512
- SY6513
- SY6514
- SY6515

Comments on the Data Sheet

The data sheet is constructed to review first the basic "Common Characteristics" - those features which are common to the general family of microprocessors. Subsequent to a review of the family characteristics will be sections devoted to each member of the group with specific features of each.



Note: 1. Clock Generator is not included on SY6512,13,14,15
2. Addressing Capability and control options vary with each of the SY6500 Products.

SY6500 Internal Architecture

COMMON CHARACTERISTICS

Clocks (ϕ_1, ϕ_2)

The SY651X requires a two phase non-overlapping clock that runs at the Vcc voltage level.

The SY650X clocks are supplied with an internal clock generator. The frequency of these clocks is externally controlled. Details of this feature are discussed in the SY650X portion of this data sheet.

Address Bus (A_0-A_{15}) (See sections on each micro for respective address lines on those devices.)

These outputs are TTL compatible, capable of driving one standard TTL load and 130pf.

Data Bus (D_0-D_7)

Eight pins are used for the data bus. This is a bi-directional bus, transferring data to and from the device and peripherals. The outputs are tri-state buffers capable of driving one standard TTL load and 130pf.

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 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. This condition will remain through a subsequent phase two (ϕ_2) in which the Ready signal is low. This feature allows microprocessor interfacing with low speed PROMS as well as fast (max. 2 cycle) Direct Memory Access (DMA). If Ready is low during a write cycle, it is ignored until the following read operation.

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 3KΩ register to Vcc 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 Register. This signal is sampled on the trailing edge of ϕ_1.

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.

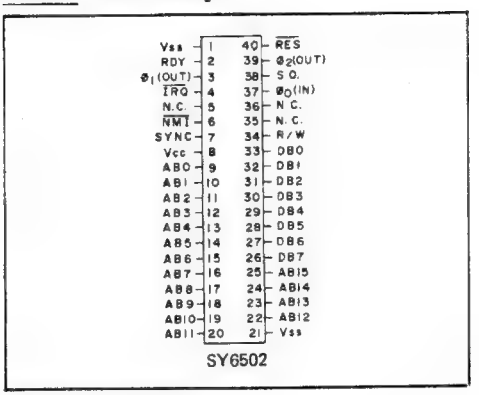
After a system initialization time of six clock cycles, the mask interrupt flag will be set and the microprocessor will load the program counter from the memory vector locations FFFC and FFFD. This is the start location for program control.

After Vcc 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.

SY6500 Signal Description

SY6502 – 40 Pin Package



SY6502

* 65K Addressable Bytes of Memory
* $\overline{\text{IRQ}}$ Interrupt * $\overline{\text{NMI}}$ Interrupt
* On-the-chip Clock
  - ✓ TTL Level Single Phase Input
  - ✓ RC Time Base Input
  - ✓ Crystal Time Base Input
* SYNC Signal
  (can be used for single instruction execution)
* RDY Signal
  (can be used for single cycle execution)
* Two Phase Output Clock for Timing of Support Chips

Features of SY6502

SY6520
SY6520A

Peripheral Interface Adapter (PIA)

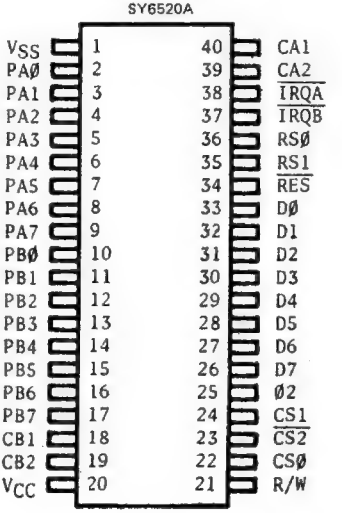
- High performance replacement for Motorola 6820 or 6821 peripheral interface adapter.
- N channel, depletion load technology, single +5V supply.
- Completely Static and TTL compatible.
- CMOS compatible peripheral control lines.
- Fully automatic "hand-shake" allows positive control of data transfers between processor and peripheral devices.

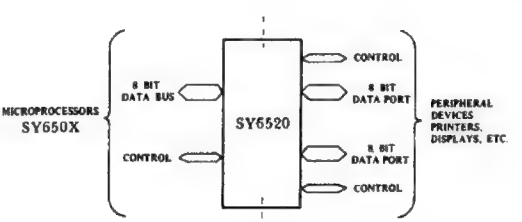
The SY6520 Peripheral Adapter is designed to solve a broad range of peripheral control problems in the implementation of microcomputer systems. This device allows a very effective trade-off between software and hardware by providing significant capability and flexibility in a low cost chip. When coupled with the power and speed of the SY6500 family of microprocessors, the SY6520 allows implementation of very complex systems at a minimum overall cost.

Control of peripheral devices is handled primarily through two 8-bit bi-directional ports. Each of these lines can be programmed to act as either an input or an output. In addition, four peripheral control/interrupt input lines are provided. These lines can be used to interrupt the processor or for "hand-shaking" data between the processor and a peripheral device.

PIN CONFIGURATION

SY6520
SY6520A





Basic SY6520 Interface Diagram

SUMMARY OF SY6520 OPERATION

CA1/CB1 CONTROL

CRA (CRB) Bit 1	CRA (CRB) Bit 0	Active Transition of Input Signal*	IRQA (IRQB) Interrupt Outputs
0	0	negative	Disable--remain high
0	1	negative	Enable--goes low when bit 7 in CRA (CRB) is set by active transition of signal on CA1 (CB1)
1	0	positive	Disable--remain high
1	1	positive	Enable--as explained above

*Note: Bit 7 of CRA (CRB) will be set to a logic 1 by an active transition of the CA1 (CB1) signal. This is independent of the state of Bit 0 in CRA (CRB).

CA2/CB2 INPUT MODES

CRA (CRB) Bit 5	CRA (CRB) Bit 4	CRA (CRB) Bit 3	Active Transition of Input Signal*	IRQA (IRQB) Interrupt Output
0	0	0	negative	Disable--remains high
0	0	1	negative	Enable--goes low when bit 6 in CRA (CRB) is set by active transition of signal on CA2 (CB2)
0	1	0	positive	Disable--remains high
0	1	1	positive	Enable--as explained above

*Note: Bit 6 of CRA (CRB) will be set to a logic 1 by an active transition of the CA2 (CB2) signal. This is independent of the stute of Bit 3 in CRA (CRB).

CA2 OUTPUT MODES

CRA Bit 5	CRA Bit 4	CRA Bit 3	Mode	Description
1	0	0	"Handshake" on Read	CA2 is set high on an active transition of the CA1 interrupt input signal and set low by a microprocessor "Read A Data" operation. This allows positive control of data transfers from the peripheral device to the microprocessor.
1	0	1	Pulse Output	CA2 goes low for one cycle after a "Read A Data" operation. This pulse can be used to signal the peripheral device that data was taken.
1	1	0	Manual Output	CA2 set low
1	1	1	Manual Output	CA2 set high

CB2 OUTPUT MODES

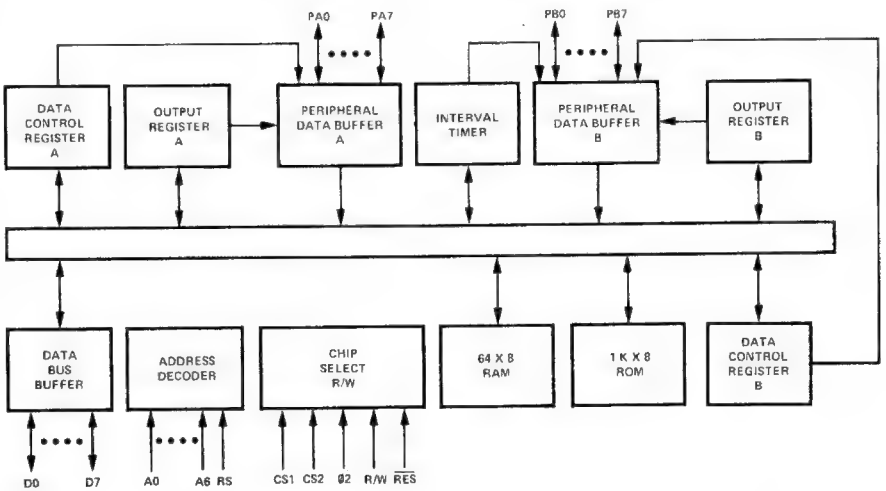
CRB Bit 5	CRB Bit 4	CRB Bit 3	Mode	Description
1	0	0	"Handshake" on Write	CB2 is set low on microprocessor "Write B Data" operation and is set high by an active transition of the CB1 interrupt input signal. This allows positive control of data transfers from the microprocessor to the peripheral device.
1	0	1	Pulse Output	CB2 goes low for one cycle after a microprocessor "Write B Data" operation. This can be used to signal the peripheral device that data is available.
1	1	0	Manual Output	CB2 set low
1	1	1	Manual Output	CB2 set high

SY6530 (MEMORY, I/O, TIMER ARRAY)

The SY6530 is designed to operate in conjunction with the SY650X microprocessor Family. It is comprised of a mask programmable 1024 x 8 ROM, a 64 x 8 static RAM, two software controlled 8 bit bi-directional data ports allowing direct interfacing between the microprocessor unit and peripheral devices, and a software programmable interval timer with interrupt, capable of timing in various intervals from 1 to 262,144 clock periods.

- 8 bit bi-directional Data Bus for direct communication with the microprocessor
- 1024 x 8 ROM
- 64 x 8 static RAM
- Two 8 bit bi-directional data ports for interface to peripherals
- Two programmable I/O Peripheral Data Direction Registers
- Programmable Interval Timer
- Programmable Interval Timer Interrupt
- TTL & CMOS compatible peripheral lines
- Peripheral pins with Direct Transistor Drive Capability
- High Impedence Three-State Data Pins
- Allows up to 7K contiguous bytes of ROM with no external decoding

FIGURE 1. SY6530 BLOCK DIAGRAM



6530 INTERFACE SIGNAL DESCRIPTION

Reset ($\overline{RES}$)

During system initialization a low ($\leq$0.4V) on the $\overline{RES}$ input will cause a zeroing of all four I/O registers. This in turn will cause all I/O buses to act as inputs thus protecting external components from possible damage and erroneous data while the system is being configured under software control. The Data Bus Buffers are put into an OFF-STATE during reset. Interrupt capability is disabled with the $\overline{RES}$ signal. The $\overline{RES}$ signal must be held low for at least one clock period when reset is required.

Input Clock

The input clock is a system Phase Two clock which can be either a low level clock ($V_{IL} < 0.4$, $V_{IH} > 2.4$ or high level clock $V_{IL} < 0.2$, $V_{IH} = V_{CC} \pm .3$).

Read/Write (R/W)

The R/W is supplied by the microprocessor array and is used to control the transfer of data to and from the microprocessor array and the SY6530. A high on the R/W pin allows the processor to read (with proper addressing) the data supplied by the SY6530. A low on the R/W pin allows a write (with proper addressing) to the SY6530.

Interrupt Request ($\overline{IRQ}$)

The $\overline{IRQ}$ pin is an interrupt pin from the interval timer. This same pin, if not used as an interrupt, can be used as a peripheral I/O pin (PB7). When used as an interrupt, the pin should be set up as an input by the data direction register. The pin will be normally high with a low indicating an interrupt from the SY6530. An external pull-up device is not required; however, if collector-OR'd with other devices, the internal pullup may be omitted with a mask option.

Data Bus (D0-D7)

The SY6530 has eight bi-directional data pins (D0-D7). These pins connect to the system's data lines and allow transfer of data to and from the microprocessor array. The output buffers remain in the off state except when a Read operation occurs.

Peripheral Data Ports

The SY6530 has 16 pins available for peripheral I/O operations. Each pin is individually software programmable to act as either an input or an output. The 16 pins are divided into 2 8-bit ports, PA0-PA7 and PB0-PB7. PB5, PB6 and PB7 also have other uses which are discussed in later sections. The pins are set up as an input by writing a "0" into the corresponding bit of the data direction register. A "1" into the data direction register will cause its corresponding bit to be an output. When in the input mode, the peripheral output buffers are in the "1" state and a pull-up device acts as less than one TTL load to the peripheral data lines. On a Read operation, the microprocessor unit reads the peripheral pin. When the peripheral device gets information from the SY6530 it receives data stored in the data register. The microproccessor will read correct information if the peripheral lines are greater than 2.0 volts for a "1" and less than 0.8 volts for a "0" as the peripheral pins are all TTL compatible. Pins PA0 and PB0 are also capable of sourcing 3 ma at 1.5V, thus making them capable of direct transistor drive.

Address Lines (A0-A9)

There are 10 address pins. In addition to these 10, there is the ROM SELECT (RS) pin. The above pins, A0-A9 and ROM SELECT, are always used as addressing pins. There are 2 additional pins which are mask programmable and can be used either individually or together as CHIP SELECTS. They are pins PB5 and PB6. When used as peripheral data pins they cannot be used as chip selects.

INTERNAL ORGANIZATION

A block diagram of the internal architecture is shown in Figure 1. The SY6530 is divided into four basic sections, RAM, ROM, I/O and TIMER. The RAM and ROM interface directly with the microprocessor through the system data bus and address lines. The I/O section consists of 2 8-bit halves. Each half contains a Data Direction Register (DDR) and an I/O Register.

ROM 1 K Byte (8 K Bits)

The 8K ROM is in a 1024 x 8 configuration. Address lines A0-A9, as well as RS are needed to address the entire ROM. With the addition of CS1 and CS2, seven SY6530's may be addressed, giving 7168 x 8 bits of contiguous ROM.

RAM - 64 Bytes (512 Bits)

A 64 x 8 static RAM is contained on the SY6530. It is addressed by A0-A5 (Byte Select), RS, A6, A7, A8, A9, and, depending on the number of chips in the system, CS1 and CS2.

Internal Peripheral Registers

There are four internal registers, two data direction registers and two peripheral I/O data registers. The two data direction registers (A side and B side) control the direction of the data into and out of the peripheral pins. A "1" written into the Data Direction Register sets up the corresponding peripheral buffer pin as an output. Therefore, anything then written into the I/O Register will appear on that corresponding peripheral pin. A "0" written into the DDR inhibits the output buffer from transmitting data to or from the I/O Register. For example, a "1" loaded into data direction A, position 3, sets up peripheral pin PA3 as an output. If a "0" had been loaded, PA3 would be configured as an input and remain in the high state. The two data I/O registers are used to latch data from the Data Bus during a Write operation until the peripheral device can read the data supplied by the microprocessor array.

During a read operation the microprocessor is not reading the I/O Registers but in fact is reading the peripheral data pins. For the peripheral data pins which are programmed as outputs the microprocessor will read the corresponding data bits of the I/O Register. The only way the I/O Register data can be changed is by a microprocessor Write operation. The I/O Register is not affected by a Read of the data on the peripheral pins.

Interval Timer

The Timer section of the SY6530 contains three basic parts: preliminary divide down register, programmable 8-bit register and interrupt logic. These are illustrated in Figure 4.

The interval timer can be programmed to count up to 256 time intervals. Each time interval can be either 1T, 8T, 64T or 1024T increments, where T is the system clock period. When a full count is reached, an interrupt flag is set to a logic "1". After the interrupt flag is set the internal clock begins counting down to a maximum of -255T. Thus, after the interrupt flag is set, a Read of the timer will tell how long since the flag was set to a maximum of 255T.

The 8 bit system Data Bus is used to transfer data to and from the Interval Timer. If a count of 52 time intervls were to be counted, the pattern 0 0 1 1 0 1 0 0 would be put on the Data Bus and written into the Interval Time register.

At the same time that data is being written into the Interval Timer, the counting intervals of 1, 8 64, 1024T are decoded from address lines A0 amd A1. During a Read or Write operation address line A3 controls the interrupt capability of PB7, i.e., $A_3 = 1$ enables $\overline{IRQ}$ on PB7, $A_3 = 0$ disables $\overline{IRQ}$ on PB7. When PB7 is used as an interrupt flag with the interval timer it should be programmed as an input. If PB7 is enabled by A3 and an interrupt occurs PB7 will go low. When the timer is read prior to the interrupt flag being set, the number of time intervals remaining will be read, i.e., 51, 50, 49, etc.

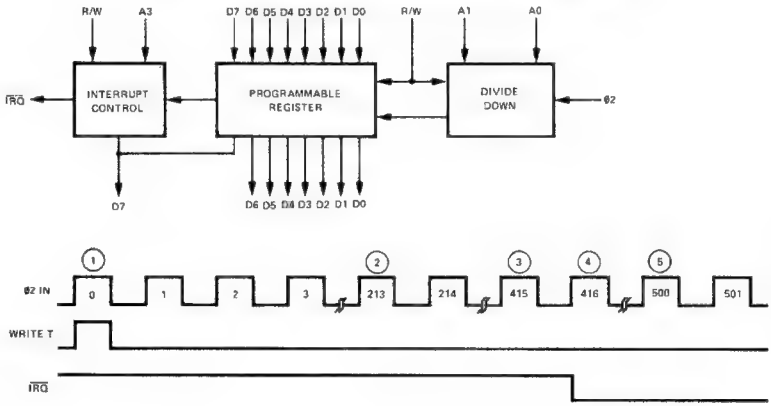
When the timer has counted down to 0 0 0 0 0 0 0 0 on the next count time an interrupt will occur and the counter will read 1 1 1 1 1 1 1 1. After interrupt, the timer register decrements at a divide by "1" rate of the system clock. If after interrupt, the timer is read and a value of 1 1 1 0 0 1 1 1 is read, the time since interrupt is 28T. The value read is in two's complement.

Value read = 1 1 1 0 0 1 0 0
Complement = 0 0 0 1 1 0 1 1
ADD 1 = 0 0 0 1 1 1 0 0 = 28.

Thus to arrive at the total elapsed time, merely do a two's complement add to the original time written into the timer. Again, assume time written as 0 0 1 1 0 1 0 0 (=52). With a divide by 8, total time to interrup is (52 x 8) +1 = 417T. Total elapsed time would be 416T + 28T = 444T, assuming the value read after interrupt was 1 1 1 0 0 1 0 0.

After interrupt, whenever the timer is written or read the interrupt is reset. However, the reading of the timer at the same time the interrupt occurs will not reset the interrupt flag.

FIGURE 4. BASIC ELEMENTS OF INTERVAL TIMER



1. Data written into Interval timer is 0 0 1 1 0 1 0 0 = 52_{10}
2. Data in Interval timer is 0 0 0 1 1 0 0 1 = 25_{10}

$$52 - \frac{213}{8} - 1 = 52\text{-}26\text{-}1 = 25$$

3. Data in Interval timer is 0 0 0 0 0 0 0 0 = 0_{10}

$$52 - \frac{415}{8} - 1 = 52\text{-}51\text{-}1 = 0$$

4. Interrupt has occured at ϕ_2 pulse #416
   Data in Interval timer = 1 1 1 1 1 1 1 1
5. Data in Interval timer is 1 0 1 0 1 1 0 0
   two's complement is 0 1 0 1 0 1 0 0 = 84_{10}
   $84 + (52 \times 8) = 500_{10}$

When reading the timer after an interrupt, A3 should be low so as to disable the $\overline{IRQ}$ pin. This is done so as to avoid future interrupts until another Write timer operation.

ADDRESSING

Addressing of the SY6530 offers many variations to the user for greater flexibility. The user may configure his system with RAM in lower memory, ROM in higher memory, and I/O registers with interval timers between the extremes. There are 10 address lines (A0-A9). In addition there is the possibility of 3 additional address lines to be used as chip-selects and to distinguish between ROM, RAM, I/O and interval timer. Two of the additional lines are CS1 and CS2. The chip-select pins can also be PB5 and PB6. Whether the pins are used as chip-selects or peripheral I/O pins is a mask option and must be specified when ordering the part. Both pins act independently of each other in that either or both pins may be designated as chip-select. The third additional address line is RS. In a 2-chip system, RS would be used to distinguish between ROM and non-ROM sections of the SY6530. With the addressing pins available, a total of 7K contiguous ROM may be addressed with no external decode.

APPENDIX 4—LABORATORY INTERFACES AND PARTS LISTS

These are the interfaces required to perform the experiments in this manual. Explanations of the functions and operations of the individual interfaces are contained in the experiments. Tables A4-1 and A4-2 contain the pin assignments for the KIM's Application Connector and Expansion Connector, respectively.

Table A4-1

PIN ASSIGNMENTS FOR THE KIM'S APPLICATION CONNECTOR

22	KB Col D
21	KB Col A
20	KB Col E
19	KB Col B
18	KB Col F
17	KB Row 0
16	PB5
15	PB7
14	PA0
13	PB4
12	PB3
11	PB2
10	PB1
9	PB0
8	PA7
7	PA6
6	PA5
5	PA4
4	PA1
3	PA2
2	PA3
1	V_{SS} (GND)

Z	KB Row 1
Y	KB Col C
X	KB Row 2
W	KB Col G
V	KB Row 3
U	TTY PRINTER
T	TTY KEYBOARD
S	TTY PRINTER RETURN (+)
R	TTY KEYBOARD RETURN (+)
P	AUDIO OUT HIGH
N	+12V
M	AUDIO OUT LOW
L	AUDIO IN
K	DECODER ENABLE
J	K7
H	K5
F	K4
E	K3
D	K2
C	K1
B	K0
A	V_{CC} (+5V)

Table A4-2

PIN ASSIGNMENTS FOR THE KIM'S EXPANSION CONNECTOR

22	V_{SS} (GND)
21	V_{CC} (+5V)
20	
19	
18	
17	SINGLE STEP OUT
16	K6
15	DB0
14	DB1
13	DB2
12	DB3
11	DB4
10	DB5
9	DB6
8	DB7
7	RESET
6	NMI
5	RO
4	IRQ
3	ϕ_1
2	RDY
1	SYNC

Z	$\overline{\text{RAM}}$/R/W
Y	ϕ_2
X	$\overline{\text{PLL}}$ TEST
W	$\overline{\text{R/W}}$
V	R/W
U	ϕ_2
T	AB15
S	AB14
R	AB13
P	AB12
N	AB11
M	AB10
L	AB9
K	AB8
J	AB7
H	AB6
F	AB5
E	AB4
D	AB3
C	AB2
B	AB1
A	AB0

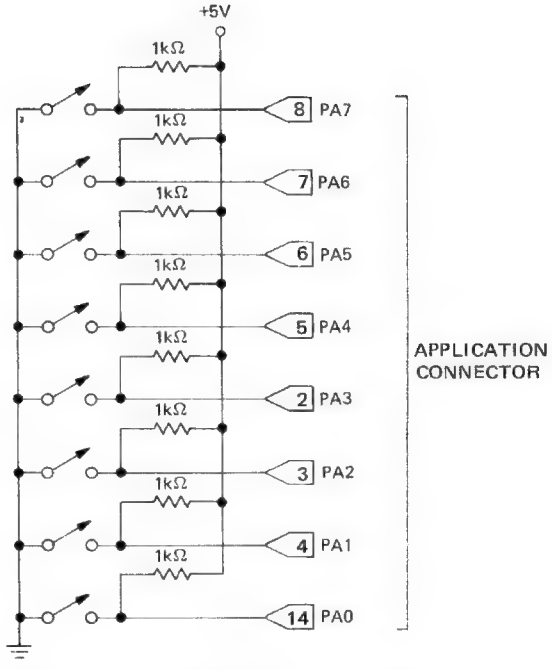


FIGURE A4-1. Attachment of switches to the Application Connector. The user 6530 device is also referred to as 6530-003 in the KIM manuals.

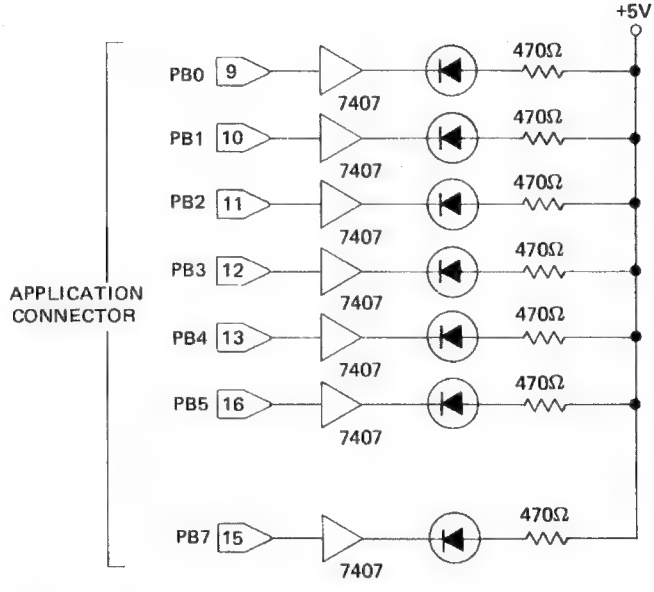


FIGURE A4-2. Attachment of LEDs to the Application Connector. (*Note:* PB6 is not available externally.)

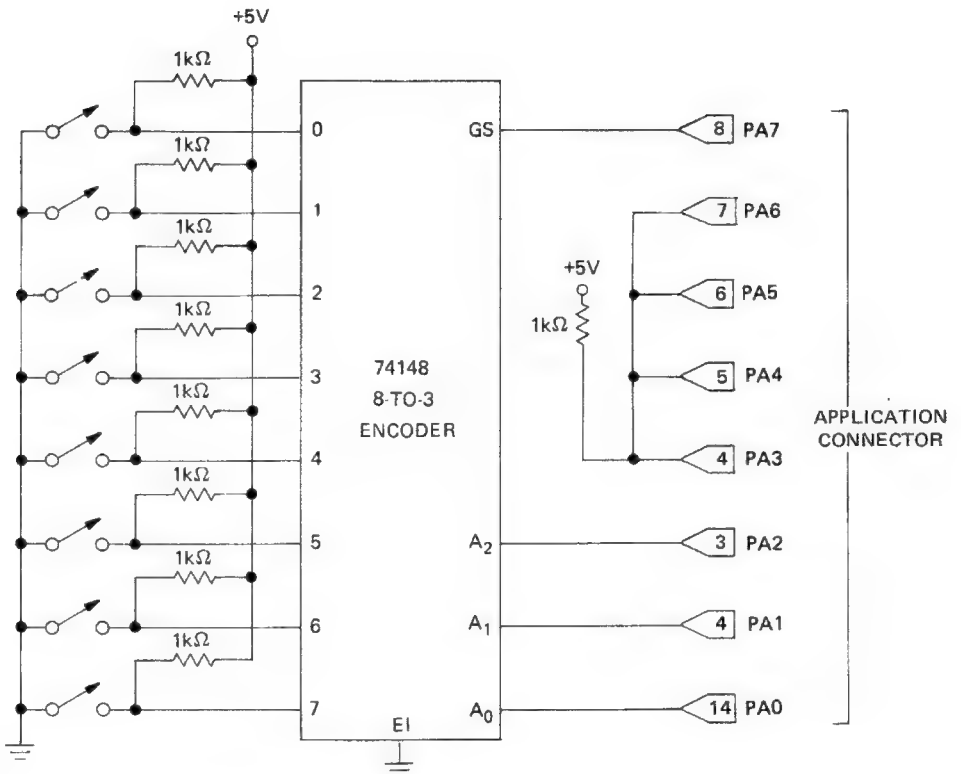
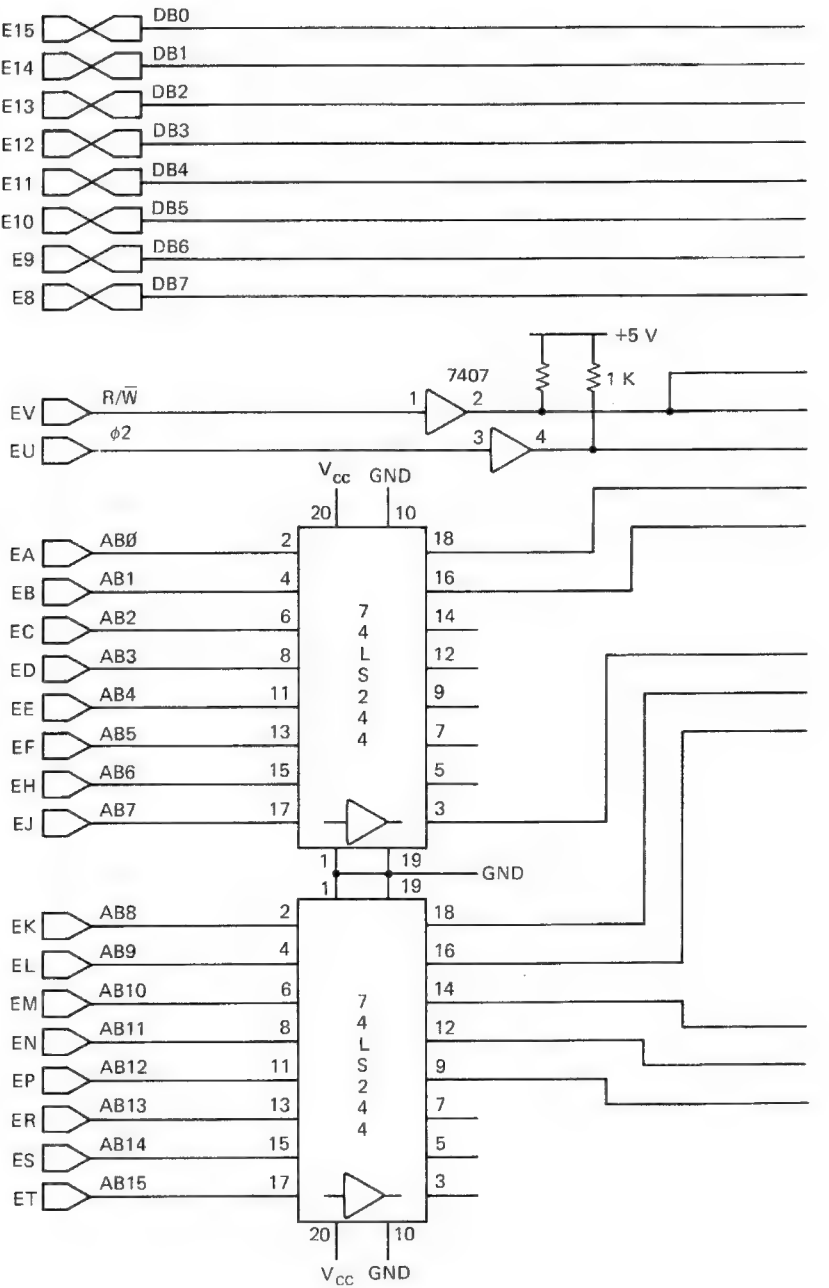


FIGURE A4-3. Attachment of switches and encoder to port A of the user 6530 device.


E15
E14
E13
E12
E11
E10
E9
E8
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
+5 V
7407
1 K
EV
R/W
EU
ϕ2
1
2
3
4
V_{cc}
GND
20
10
EA
EB
EC
ED
EE
EF
EH
EJ
ABØ
AB1
AB2
AB3
AB4
AB5
AB6
AB7
2
4
6
8
11
13
15
17
74LS244
18
16
14
12
9
7
5
3
1
19
GND
EK
EL
EM
EN
EP
ER
ES
ET
AB8
AB9
AB10
AB11
AB12
AB13
AB14
AB15

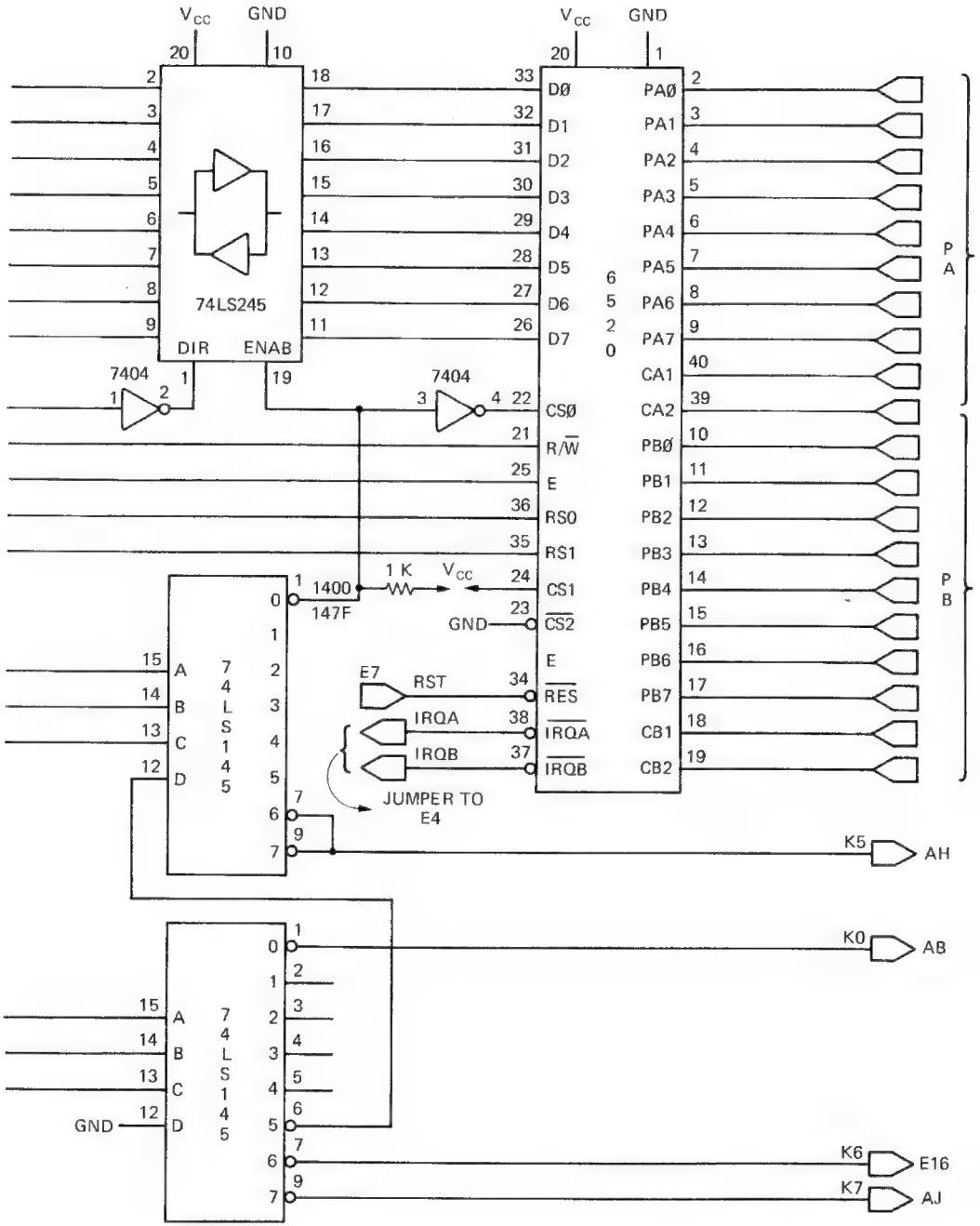


FIGURE A4-4. Attachment of a 6520 PIA to the KIM-1 microcomputer.

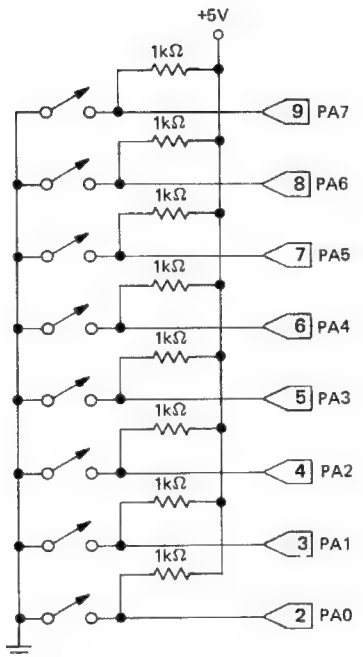


FIGURE A4-5. Attachment of switches to port A of the 6520 PIA.

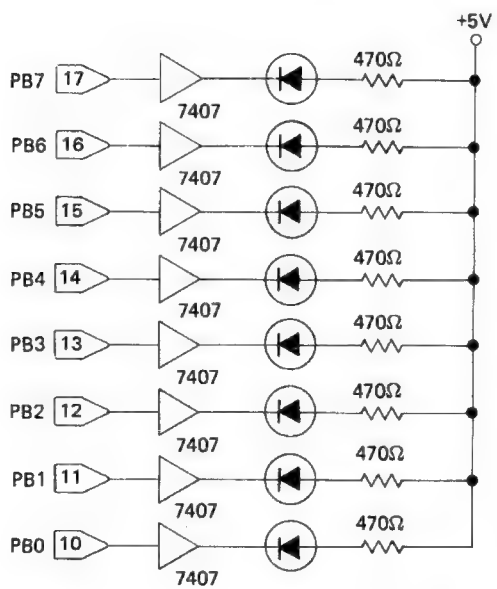


FIGURE A4.6 Attachment of LEDs to port B of the 6520 PIA.

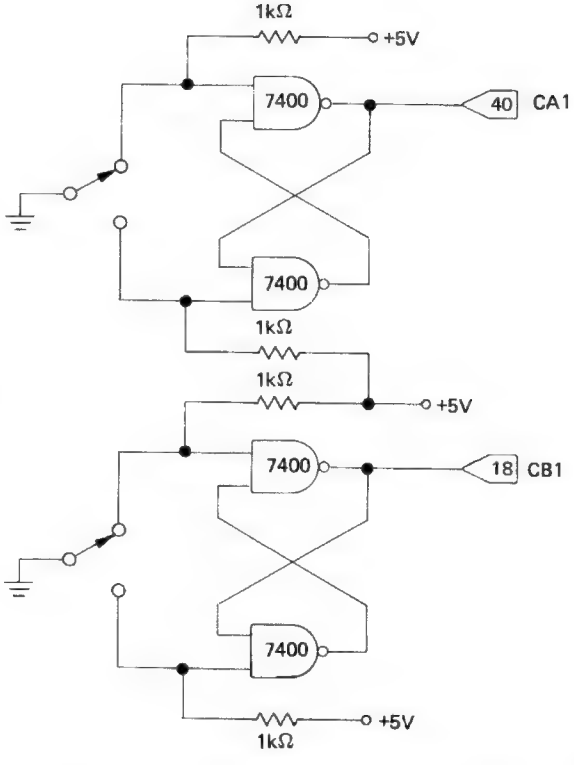


FIGURE A4-7. Attachment of switches to control lines CA1 and CB1 of the 6520 PIA.

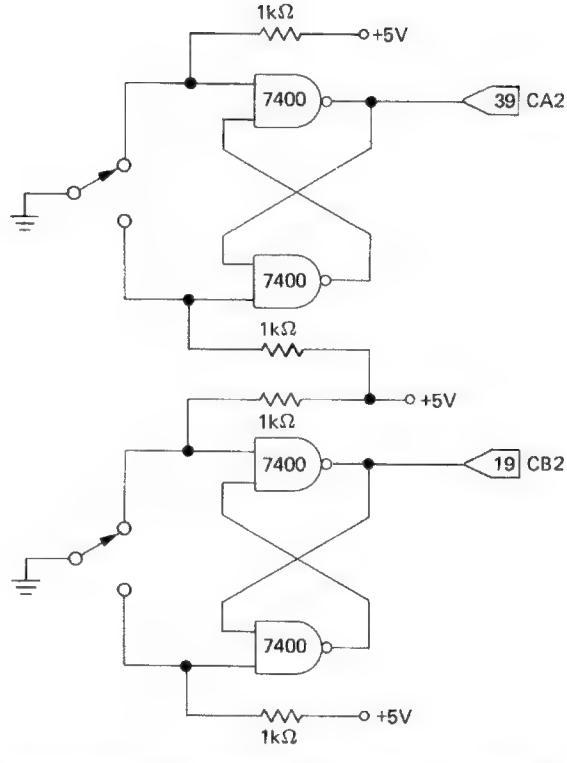


FIGURE A4-8. Attachment of switches to control lines CA2 and CB2 of the 6520 PIA.

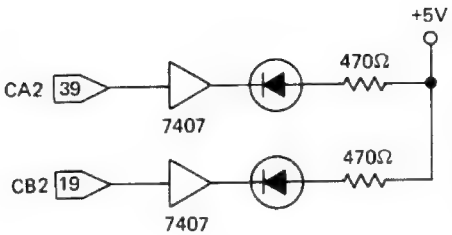


FIGURE A4-9. Attachment of LEDs to control lines CA2 and CB2 of the 6520 PIA. **Note:** Jumper wires are an easy way to select between the circuits in Figures A4-8 and A4-9. If you are not careful, using CA2 and CB2 as outputs could damage the AND gates in Figure A4-8.

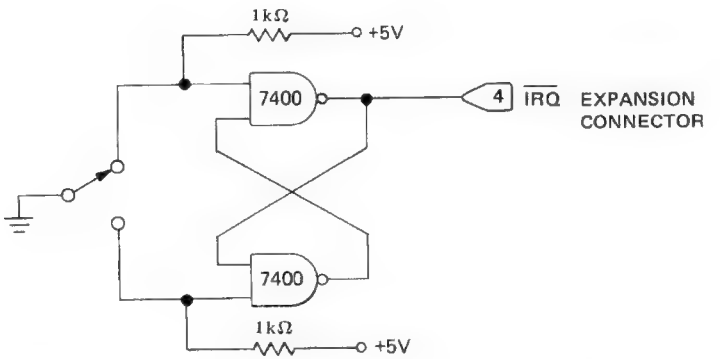


FIGURE A4-10. Attachment of a debounced switch to the processor's $\overline{\text{IRQ}}$ input via the Expansion Connector. **Note:** You should use a push-button (with three terminals) rather than a toggle switch in this circuit, so that you never start a program with the interrupt active. If you use a toggle switch, always bring $\overline{\text{IRQ}}$ high before executing your program.

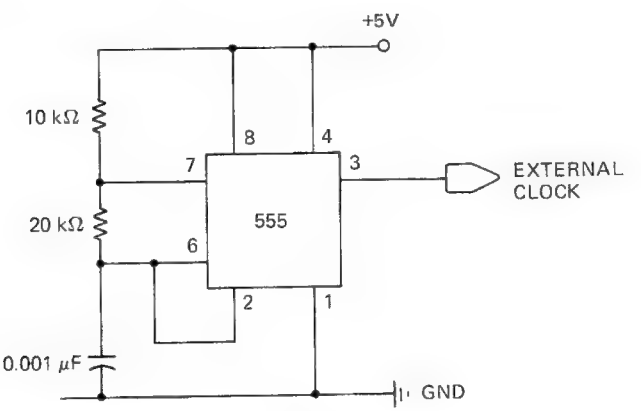


FIGURE A4-11. A simple circuit for generating a low-frequency clock from a 555 tiner chip. The frequency is approximately 83 Hz.

FIGURE A4-12. Connection of the external clock to bit 5 of user 6530 port A (address 1700 hexadecimal). **Note:** Jumper wires can be used to select either this input or the switch input shown in Figure A4-1.

FIGURE A4-13. Connection of pin PB7 of the user 6530 device (the timer interrupt request) to the microprocessor's $\overline{IRQ}$ input. **Note:** Jumper wires can be used to select either this output or the LED output shown in Figure A4-2.

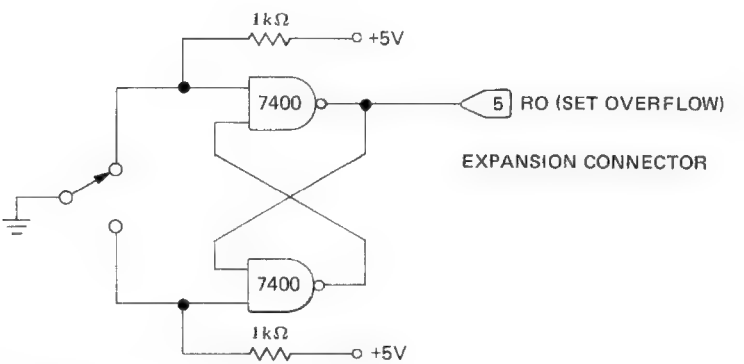


FIGURE A4-14. Connections for the 6502 Set Overflow (RO) input.

PARTS LIST FOR LABORATORY EXERCISES

ITEM	DESCRIPTION	QUANTITY	LABORATORIES	USER 6530 PORT
SPDT switch	Alco TT 11DG-WW-2T	8	2, 7, B, C, E	A
1-kΩ resistor pack	Bourns 898-1-R1K	1	2, 7, B, C, E	A
or 1-kΩ resistors		8	2, 7, B, C, E	A
LED display	Red	7	3, C, D, E	B
500-Ω resistor pack	Bourns 898-1-500	1	3, C, D, E	B
or 500-Ω resistors		7		
7407 IC	Hex Buffer/Driver	2	3, C, D, E	B
Decimal switch		1	4	A
74148 IC	Priority Encoder	1	4	A
1-kΩ resistor pack	Bourns 898-1-R1K	1	4	A
SPDT switch	Alco TT 11DG-WW-2T	1	C	
1-kΩ resistor pack	Bourns 898-1-R1K	1	C, E	
or 1-kΩ resistors		4		
7400 IC	Quad NAND	1	C, E	
10-kΩ resistor		1	D	
20-kΩ resistor		1	D	
0.001-μF capacitor		1	D	
555 timer IC		1	D	A (bit 5)
SPDT switch	Alco TT 11DG-WW-2T	1	E	
Miscellaneous:				
Vector prototyping board 3677-2		2		
44-pin connector		2		
8-pin wire-wrap sockets		1		
14-pin wire-wrap sockets		7		
16-pin wire-wrap sockets		1		

ADDITIONAL PARTS FOR OPTIONAL PIA EXPERIMENTS (LABORATORY B)

ITEM	DESCRIPTION	QUANTITY	6520 PIA PORT
6520 or 6820 IC	Parallel Interface	1	
7404 IC	Hex Inverter	1	
7407 IC	Hex Buffer/Driver	1	
74LS145 IC	Decimal Decoder/Driver	2	
74LS244 IC	Octal Buffer/Driver	2	
74LS245 IC	Octal Bus Transceiver	1	
1-kΩ resistors		3	
SPDT switch	Alco TT 11DG-WW-2T	8	A
1-kΩ resistor pack	Bourns 898-1-R1K	1	A
or 1-kΩ resistors		8	A
SPDT switch	Alco TT 11DG-WW-2T	2	A (CA1), B(CB1)
1-kΩ resistor pack	Bourns 898-1-R1K	1	A (CA1), B(CB1)
or 1-kΩ resistors		4	
7400 IC	Quad NAND	1	A (CA1), B(CB1)
LED display	Red	8	B
500-Ω resistor pack	Bourns 898-1-500	1	B
or 500-Ω resistors		8	
7407 IC	Hex Buffer/Driver	2	B
SPDT switch	Alco TT 11DG-WW-2T	2	A (CA2), B(CB2)
2-kΩ resistor pack	Bourns 898-1-R1K	1	A (CA2), B(CB2)
or 1-kΩ resistors		4	
7400 IC	Quad NAND	1	A (CA2), B(CB2)
LED display	Red	2	A (CA2), B(CB2)
500-Ω resistor		2	A (CA2), B(CB2)
7407 IC	Hex Buffer/Driver	1	A (CA2), B(CB2)
Miscellaneous:			
14-pin wire-wrap sockets		10	
16-pin wire-wrap sockets		3	
20-pin wire-wrap sockets		3	
40-pin wire-wrap sockets		1	

EXAMPLE LABORATORY SETUP

The example laboratory setup was constructed on two Vector 3677-2 prototyping boards. Figure A4-15 shows the prototyping boards and their connection to the KIM. Figures A4-1 through A4-14, A4-16, and A4-17 describe the circuitry on these boards and their connection in detail. Assembled I/O boards for the KIM are also available from several sources, including The Computerist, Inc., P.O. Box 3, S. Chelmsford, MA 01824. M. L. DeJong describes a typical experimental setup using an assembled board on pp. 60-62 of his book entitled *Programming and Interfacing the 6502, with Experiments* (Howard W. Sams, Indianapolis, IN, 1980); DeJong's setup differs from ours in that he uses port A of the 6530 device for output and port B for input. A complete laboratory interface board (MICROLAB™) is also available from Cambridge Development Laboratory, 36 Pleasant St., Watertown, MA 02172.

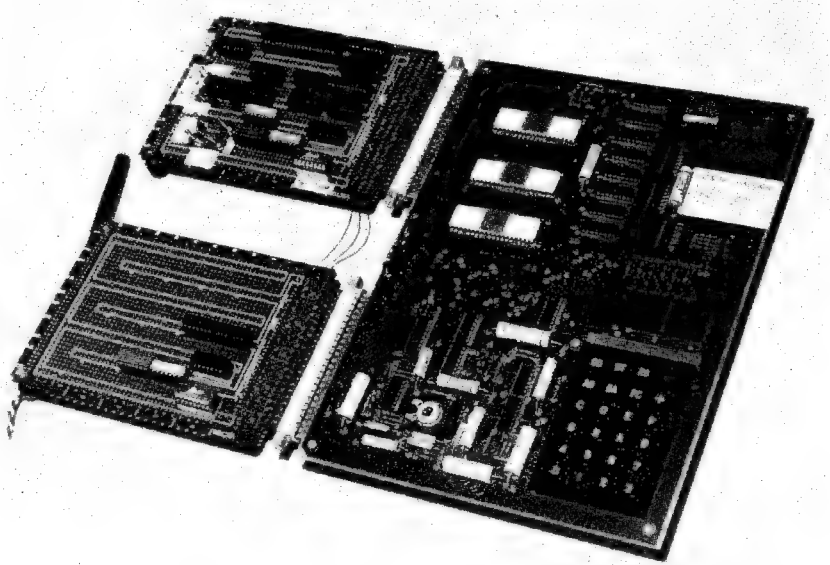


FIGURE A4-15. The KIM-1 microcomputer connected to the laboratory experiment boards. (Photo courtesy of Carter Stafford.)

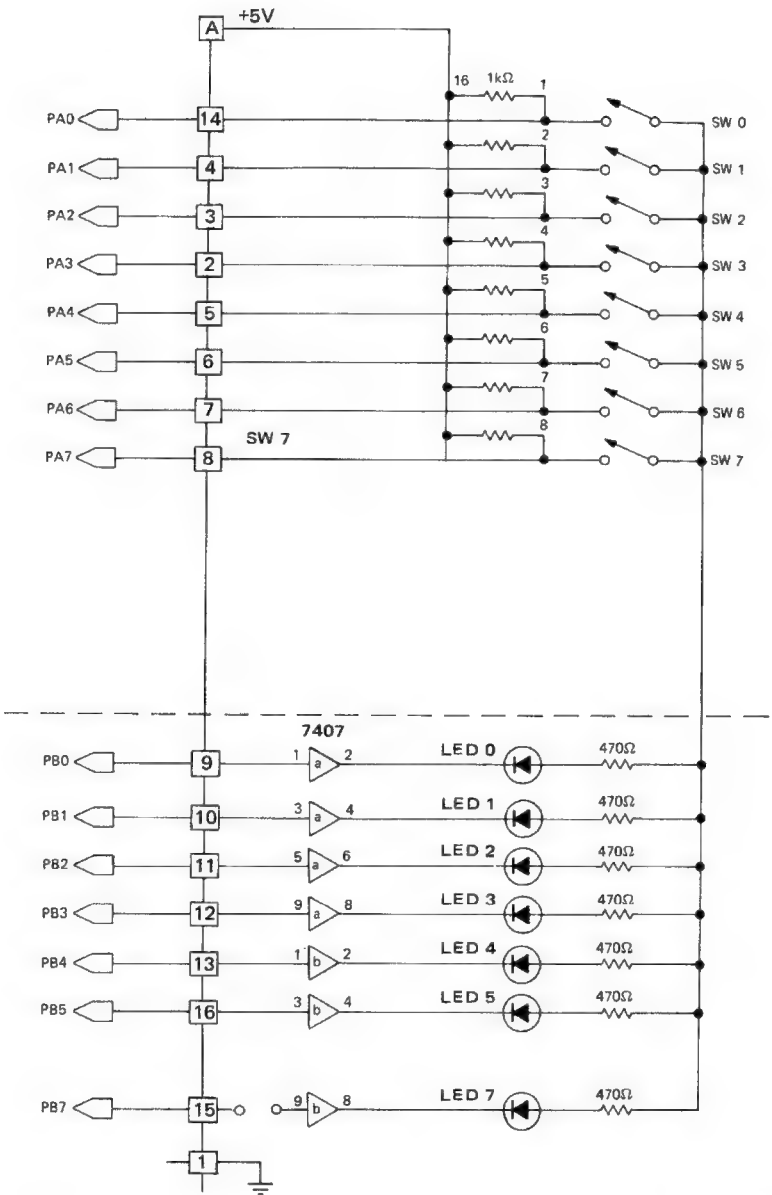


FIGURE A4-16. Connections to the Application Connector. *Note:* A break indicates the need for a jumper wire to make the connection.

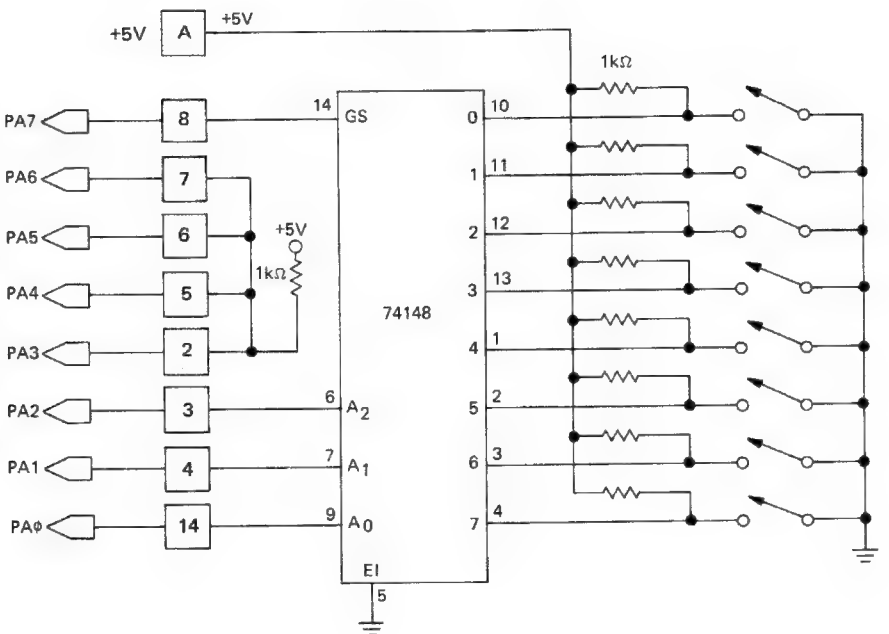


FIGURE A4-17. Encoder connection for Laboratory 4.

APPENDIX 5—SUMMARY OF THE KIM MONITOR

The following descriptions are taken from the KIM-1 Microcomputer Module User Manual (August 1976 edition) and are reprinted here with the permission of Commodore/MOS Technology Inc., Norristown, PA. The table of monitor subroutines given here as Table A5-1 also appears in Laboratory A as Table A-1; Table A5-2 appears in Laboratory A as Table A-2. Figure A5-1 describes the memory addresses available in the KIM and Figure A5-2 lists the addresses that are reserved either for monitor functions or for I/O devices.

Table A5-1

KIM MONITOR SUBROUTINES

SUBROUTINE NAME	STARTING ADDRESS	DESCRIPTION
AK	1EFE	Scans the KIM keyboard to see if any keys are depressed. Returns with (A) = 0 if none is, (A) nonzero otherwise. Requires port A of the keyboard/display 6530 device to be programmed as an input.
CHK	1F91	Adds the contents of the accumulator to the 16-bit number stored in memory locations 00F6 and 00F7. Used in calculating a checksum for paper tape records.
CONVD	1F48	1) Converts hexadecimal digit in A to a seven-segment code using KIM table. 2) Selects a KIM seven-segment display using the contents of index register X (see Table 5-2). 3) Drives the display for about 0.5 ms. 4) Increments X by 2 to prepare for driving the next display.
CRLF	1E2F	Prints a carriage return and a line feed on the teletypewriter.
DELAY	1ED4	Software delay that counts down from the 16-bit number in memory locations 17F3 (MSBs) and 17F2 (LSBs).
DEHALF	1EEB	Same as DELAY except that it starts the count at half the value in memory locations 17F3 and 17F2.
GETCH	1E5A	Reads one character from the teletypewriter and places it in A.
GETBYT	1F9D	Reads two hexadecimal characters from the teletypewriter, packs them into a single byte, and stores the byte in memory location 00F8. It also moves the previous contents of memory location 00F8 to 00F9 to allow the loading of 16-bit addresses as four hex digits.
GETKEY	1F6A	Scans the KIM keyboard and produces a result in the accumulator according to Table A5-2. Requires port A of the keyboard/display 6530 device to be programmed as an input.
HEXTA	1E4C	Prints 4 least significant bits of A as a hexadecimal character on the teletypewriter.
INCPT	1F63	Increments the 16-bit value in memory locations 00FA (LSBs) and 00FB (MSBs).

Table A5-1 (continued)

KIM MONITOR SUBROUTINES

SUBROUTINE NAME	STARTING ADDRESS	DESCRIPTION
INITS	1E88	Initializes the KIM, performing the following steps: 1) Puts keyboard in address mode. 2) Makes port A of the keyboard/display 6530 device into an input port. 3) Makes bits 0 through 5 of port B of the keyboard/display 6530 device into outputs and bit 7 into an input for use with teletypewriter. 4) Enables the teletypewriter (as opposed to the cassette tape unit) by clearing bit 5 of port B of the keyboard/display 6530 device and places the teletypewriter in its normal (logic 1) state by setting bit 0 of that same port. 5) Puts the 6502 processor in the binary mode by executing CLD.
NIT1	1E8C	Same as INITS except that it omits step 1 and thus does not affect the keyboard mode.
OPEN	1FCC	Moves the contents of memory location 00F8 to 00FA and 00F9 to 00FB.
OUTCH	1EA0	Prints the character in the accumulator on the teletypewriter.
OUTSP	1E9E	Prints a space on the teletypewriter.
PACK	1FAC	Converts the ASCII character in the accumulator into a hexadecimal digit. It then shifts that digit into the 4 least significant bit positions of the 16-bit number in memory locations 00F8 (LSBs) and 00F9 (MSBs), also shifting the previous number left 4 bits. This procedure allows the assembly of 8-bit data items or 16-bit addresses from hexadecimal digits. If the accumulator does not contain a valid digit, the routine returns with A unchanged.
PRTBYT	1E3B	Prints the contents of A as two hexadecimal digits on the teletypewriter.
PRTPNT	1E1E	Prints the contents of memory locations 00FA (LSBs) and 00FB (MSBs) as four hexadecimal digits on the teletypewriter.
PRTST	1E31	Prints on the teletypewriter a string of ASCII characters starting at 1FD5 + (X) and concluding at 1FD5. The string starting at 1FD5 consists of 6 nulls (00), line feed (0A), carriage return (0D), M, I, K, space (20), a control character (13), R, R, E, space (20), and another control character (13).
SCAND	1F19	Drives the KIM display, displaying the address in memory locations 00FA (LSBs) and 00FB (MSBs) and the contents of that address using CONVD. Exits to subroutine AK.
SCANDS	1F1F	Same as SCAND, except that it displays the contents of memory location 00F9 as the data instead of the contents of the address in 00FA and 00FB.

Table A5-2

KIM KEYBOARD CODES (FROM SUBROUTINE GETKEY)

KEY PRESSED	CODE IN ACCUMULATOR (HEX)
0	00
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	08
9	09
A	0A
B	0B
C	0C
D	0D
E	0E
F	0F
AD	10
DA	11
+	12
GO	13
PC	14
NONE	15

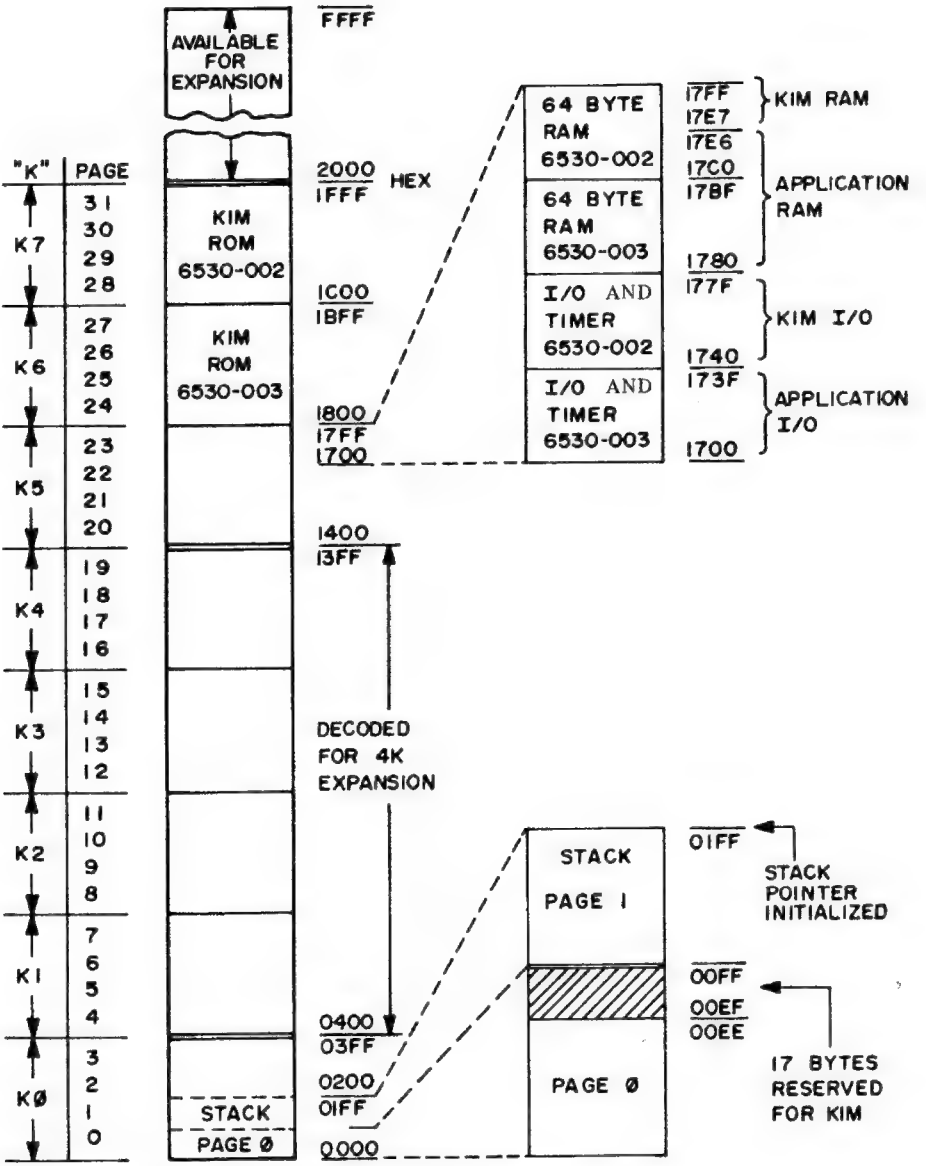


FIGURE A5-1. Memory map for the KIM-1 microcomputer.

ADDRESS	AREA	LABEL	FUNCTION
00EF	↑	PCL	Program Counter - Low Order Byte
00F0		PCH	Program Counter - High Order Byte
00F1	Machine Register Storage Buffer	P	Status Register
00F2		SP	Stack Pointer
00F3		A	Accumulator
00F4		Y	Y-Index Register
00F5	↓	X	X-Index Register
1700	↑	PAD	6530-003 A Data Register
1701	Application I/O	PADD	6530-003 A Data Direction Register
1702		PBD	6530-003 B Data Register
1703	↓	PBDD	6530-003 B Data Direction Register
1704 ↓ 170F	↑ Interval Timer ↓		6530-003 Interval Timer (See Section 1.6 of Hardware Manual)
17F5	↑	SAL	Starting Address - Low Order Byte
17F6	Audio Tape Load & Dump	SAH	Starting Address - High Order Byte
17F7		EAL	Ending Address - Low Order Byte
17F8		EAH	Ending Address - High Order Byte
17F9	↓	ID	File Identification Number
17FA	↑	NMIL	NMI Vector - Low Order Byte
17FB		NMIH	NMI Vector - High Order Byte
17FC	Interrupt Vectors	RSTL	RST Vector - Low Order Byte
17FD		RSTH	RST Vector - High Order Byte
17FE		IRQL	IRQ Vector - Low Order Byte
17FF	↓	IRQH	IRQ Vector - High Order Byte
1800	↑ Audio Tape ↓	DUMPT	Start Address - Audio Tape Dump
1873		LOADT	Start Address - Audio Tape Load
1C00	↑ STOP Key + SST ↓		Start Address for NMI using KIM "Save Machine" Routine (Load in 17FA & 17FB)
17F7	↑ Paper Tape Dump (Q)	EAL	Ending Address - Low Order Byte
17F8	↓	EAH	Ending Address - High Order Byte

FIGURE A5-2. Special memory addresses in the KIM-1 microcomputer.

INITIALIZATION PROCEDURE (FROM POWER-ON) FOR NORMAL KIM OPERATION

1) Move the SINGLE-STEP slide switch to the OFF position (the ON position is marked).
2) Set the return address for the ST (STOP) key to 1C00 hex. That is,

$$(17FA) = 00$$
$$(17FB) = 1C$$

3) Set the return address for BRK to 1C00 hex. That is,

$$(17FE) = 00$$
$$(17FF) = 1C$$

USING THE KIM-1 KEYBOARD AND DISPLAY

A brief study of your keyboard shows a total of 23 keys and one slide switch. First, let's list the purpose of each key:

[0] TO [F]	16 keys used to define the hex code of address or data.
[AD]	selects the address entry mode.
[DA]	selects the data entry mode.
[+]	increments the address by +1 but does not change the entry mode.
[PC]	recalls the address stored in the Program Counter locations (PCH, PCL) to the display.
[RS]	causes a total system reset and a return to the control of the operating program.
[GO]	causes program execution to begin, starting at the address shown in the display.
[ST]	terminates the execution of a program and causes a return to the control of the operating program.

You have seen in an earlier chapter that the six-digit display includes a four-digit display of an address (left four digits) and a two-digit display of data (right two digits).

Using only the KIM-1 keyboard and display, you may perform any of the following operations:

1) *Select an Address.* Press [AD] followed by any four of the hex entry keys. The address selected will appear on the display. If an entry error is made, just continue to enter the correct hex keys until the desired address shows on the display. Regardless of what address is selected, the data field of the display will show the data stored at that address.

2) *Modify Data.* After selecting the proper address, press [DA] followed by two hex entry keys which correctly define the data to be stored at the selected address. The data entered will appear in the data field of the display to indicate that the desired code has already been entered.
Note that it is possible for you to select an address of a ROM memory cell or even the address of a memory cell that does not exist in your system. In these cases, you will not be able to change the data display since it is clearly not possible for the system to write data to a ROM cell or a nonexistent memory location.

3) *Increment the Address.* By pressing the [+] key the address displayed is automatically increased by +1. Of course, the data stored at the new address will appear on the display. This operation is useful when a number of successive address locations must be read or modified. Note that the use of the [+] key will not change the entry mode. If you had previously pressed the [AD] key, you remain in the address entry mode and a previous depression of the [DA] means you remain in the data entry mode.

4) *Recall Program Counter.* Whenever the $\overline{\text{NMI}}$ interrupt pin of the 6502 microprocessor array is activated, the program execution in progress will halt and the internal registers of the 6502 are saved in special memory locations before the control of the system is returned to the operating program. In the KIM-1 system, the $\overline{\text{NMI}}$ interrupt may occur in response to a depression of the [ST] key (stop) or, when operating in the Single Step mode, after each program instruction is executed following the depression of the [GO] key.
The [PC] key allows you automatically to recall the value of the Program Counter at the time an interrupt occurred. You may have performed a variety of operations since the interrupt such as inspecting the contents of various machine registers stored at specific memory locations. However, when you press the [PC] key, the contents of the Program Counter at the time of the interrupt are recalled to the address field of the display. You now may continue program execution from that point by pressing the [GO] key.

5) *Execute a Program.* Select the starting address of the desired program. Now, press the [GO] key and program execution will commence starting with the address appearing on the display.

6) *Terminate a Program.* The [ST] key is provided to allow termination of program execution. As mentioned earlier, the [ST] key activates the $\overline{\text{NMI}}$ interrupt input of the 6502 microprocessor array.

Note: The [ST] key will operate correctly only if you store the correct interrupt vector at locations 17FA and 17FB. For most of your work with the KIM-1 system, you should store the address 1C00 in these locations as follows:

(17FA) = 00

(17FB) = 1C

Now, when the $\overline{\text{NMI}}$ interrupt occurs, the program will return to location 1C00 and will proceed to save all machine registers before returning control to the operating program.

You should remember to define the $\overline{\text{NMI}}$ vector each time the power to the system has been interrupted. A failure of the system to react to the [ST] key means you have forgotten to define the $\overline{\text{NMI}}$ vector.

7) *Single-Step Program Execution.* In the process of debugging a new program, you will find the single-step execution mode helpful. To operate in this mode, move the SST slide switch to the ON position (to your right). Now, depress the [GO] key for each desired execution of a program step. The display will show the address and data for the next *instruction* to be executed. Note that in the course of stepping through a program, certain addresses will appear to be skipped. A program instruction will occupy 1, 2, or 3 bytes of memory, depending upon the type of instruction. In single-instruction mode, all of the bytes involved in the execution of the instruction are accessed and the program will halt only on the first byte of each successive instruction.

Note: SST mode also makes use of the $\overline{\text{NMI}}$ interrupt of the 6502 microprocessor array. Again, the $\overline{\text{NMI}}$ vector must be defined as described in operation 6 if the SST mode is to work correctly.

This covers all of the standard operations you may perform from the KIM-1 keyboard. Using combinations of the operations described, you may wish to perform certain specialized tasks as follows:

1) *Define the $\overline{IRQ}$ Vector.* You will recall that a separate interrupt input labeled $\overline{IRQ}$ is available as an input to the 6502 microprocessor array. If you wish to use this feature, you should enter the address to which the program will jump. The $\overline{IRQ}$ vector is stored in locations 17FE and 17FF.

2) *Interrogate Machine Status.* We have mentioned that after an $\overline{NMI}$ interrupt in response to the [ST] key or during the SST mode, the contents of various machine registers are stored in specific memory locations. If you wish to inspect these locations, their addresses are:

   00EF = PCL (less significant byte of program counter)

   00F0 = PCH (more significant byte of program counter)

   00F1 = Status Register (P)

   00F2 = Stack Pointer (SP)

   00F3 = Accumulator (A)

   00F4 = Index Register Y

   00F5 = Index Register X

References

BOOKS

Artwick, B., *Microcomputer Interfacing,* Prentice-Hall, Englewood Cliffs, NJ, 1980.

Blakeslee, T. R., *Digital Design with Standard MSI and LSI, 2nd ed.,* Wiley, New York, 1979.

Camp, R. C., et al., *Microcomputer Systems Principles Featuring the 6502/KIM,* Matrix Publishers, Inc., Portland, OR, 1978.

DeJong, M. L., *Programming and Interfacing the 6502, with Experiments,* Howard W. Sams, Indianapolis, IN, 1980.

Hart, J. F., et al., *Computer Approximations,* Wiley, New York, 1968.

Hughes, J. K., and J. I. Michtom, *A Structured Approach to Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1977.

Hwang, K., *Computer Arithmetic,* Wiley, New York, 1979.

Kulisch, U. W., and W. L. Mirankar, *Computer Arithmetic in Theory and Practice,* Academic Press, New York, 1980.

Leventhal, L. A., *Introduction to Microprocessors: Software, Hardware, Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

Leventhal, L. A., *6502 Assembly Language Programming,* Osborne/McGraw-Hill, Berkeley, CA, 1979.

Lipovski, G. J., *Microcomputer Interfacing: Principles and Practices,* D. C. Heath (Lexington Books), Lexington, MA, 1980.

Luke, Y. L., *Mathematical Functions and Their Approximations,* Academic Press, New York, 1975.

McNamara, J. E., *Technical Aspects of Data Communications,* Educational Services Department, Digital Equipment Corp., Maynard, MA, 1977.

MOS Technology, Inc., *MCS6500 Microcomputer Family Hardware Manual,* Commodore/MOS Technology Inc., Norristown, PA, 1976.

MOS Technology, Inc., *MCS6500 Microcomputer Family Programming Manual,* Commodore/MOS Technology Inc., Norristown, PA, 1976.

Osborne, A., *An Introduction to Microcomputers, Vol. 1: Basic Concepts, 2nd Ed.,* Osborne/McGraw-Hill, Berkeley, CA, 1980.

Osborne, A., *An Introduction to Microcomputers, Vol. 2: Some Real Microprocessors,* Osborne/McGraw-Hill, Berkeley, CA, 1978.

Peatman, J. B., *The Design of Digital Systems,* McGraw-Hill, New York, 1972.

Peatman, J. B., *Digital Hardware Design,* McGraw-Hill, New York, 1980.

Peatman, J. B., *Microcomputer-Based Design,* McGraw-Hill, New York, 1977.

Scanlon, L. J., *6502 Software Design,* Howard W. Sams, Indianapolis, IN, 1980.

Schmid, H., *Decimal Computation,* Wiley, New York, 1974.

Tocci, R. J. and L. P. Laskowski, *Microprocessors and Microcomputers: Hardware and Software, 2nd ed.,* Prentice-Hall, Englewood Cliffs, NJ, 1982.

Weller, W. J., *Practical Microcomputer Programming: The 6502,* Northern Technology Books, Evanston, IL, 1980.

ARTICLES

Note: Two hobbyist magazines deal exclusively with 6502-based microcomputers. They are *Micro* (P.O. Box 6502, Chelmsford, MA 01824) and *Compute* (P.O. Box 5406, Greensboro, NC 27403).

Allison, D. R., "A Design Philosophy for Microcomputer Architectures," *Computer,* February 1977, pp. 35-41.

Babb, S. M., and G. L. Johnson, "Wind Instrumentation with Microprocessors," *IECI 1979 Proceedings,* pp. 121-125.

Bradshaw, P., "Two-Chip A/D Converter," *Electronic Design,* March 29, 1979, pp. 128-136.

Buzen, J. P., "I/O Subsystem Architecture," *Proceedings of the IEEE,* June 1975, pp. 871-879.

Caprihan, A., et al., "A Simple Microcomputer for Biomedical Signal Processing," *IECI 1978 Proceedings,* pp. 18-23.

Cole, G., "Time Simultaneous Events with a Microprocessor," *EDN,* November 20, 1980, pp. 99-100.

Corder, M. R., "6500 Program Automatically Sets Communications Chip Bit Rate," *Electronics,* March 13, 1980, pp. 149-151.

Cushman, R. H., "Successful Microsystems Combine User Needs and Microcomputer Technology," *EDN,* April 20, 1977, pp. 104-111.

Cushman, R. H., "2-½ Generation Microprocessors—$10 Parts that Perform Like Low-End Minis," *EDN,* September 20, 1975, pp. 36-41.

Dittman, P., et al., "Logic Analyzers Simplify System Integration Tasks," *Computer Design,* March 1981, pp. 119-126.

Donn, E. S., and M. D. Lippman, "Efficient and Effective Microcomputer Testing Requires Careful Preplanning," *EDN,* February 20, 1979, pp. 97-107.

Grappel, R., "Technique Avoids Interrupt Dangers," *EDN,* May 5, 1979, pp. 88.

Hemenway, J., "Microcomputer Operating Systems Directory," *EDN,* November 5, 1980, pp. 275-338.

Hemenway, J., and R. D. Grappel, "EDN Software Tutorial: Sorting Algorithms," *EDN,* September 20, 1980, pp. 153-157.

Hemenway, J., and E. Teja, "EDN Software Tutorial: File Structures," *EDN,* June 20, 1979, pp. 153-155.

Hemenway, J., and E. Teja, "EDN Software Tutorial: Hash Coding," *EDN,* September 20, 1979, pp. 108-110.

Landau, J. V., "State Description Techniques Applied to Industrial Machine Control," *Computer,* February 1979, pp. 32-40.

Larsen, D. G., et al., "INWAS: Interfacing with Asynchronous Serial Mode," *IEEE Transactions on Industrial Electronics and Control Instrumentation,* February 1977, pp. 2-12.

Leventhal, L. A., "Cut Your Processor's Computation Time," *Electronic Design,* August 16, 1977, pp. 82-88.

Leventhal, L. A., "Put Microprocessor Software to Work," *Electronic Design,* August 2, 1976, pp. 58-64.

Lorentzen, R., "Logic Analyzers Finish What Development Systems Start," *Electronic Design,* March 29, 1980, pp. 81-85.

Mavity, W. C., "Megabit Bubble Modules Move In on Mass Storage," *Electronics,* March 29, 1979, pp. 99-103.

McLeod, J., "Special Report: Logic Analyzers," *Electronic Design,* March 29, 1980, pp. 48-56.

Morris, G., "Make Your Next Instrument Design Emphasize User Needs and Wants," *EDN,* October 20, 1978, pp. 100-105.

Morris, G., "Task Analysis Lets You Design without Injecting Personal Quirks," *EDN,* November 5, 1978, pp. 82-88.

Ogdin, C. A., "A Floppy-Disc Interface Is More Than Just a Chip," *EDN,* August 20, 1978, pp. 115-119.

Ogdin, C. A., "Interfaces on a Chip," *Mini-Micro Systems,* November 1978, pp. 95-104.

Ogdin, C. A., "Setting Up a Microcomputer Design Laboratory," *Mini-Micro Systems,* May 1979, pp. 87-94.

Ogdin, C. A., "Some Simple Hardware Techniques Allow Fail-Safe LSI Interfacing," *EDN*, February 20, 1979, pp. 117-122.

Olivier, G., et al., "Microprocessor Controller for a Thyristor Converter with an Improved Power Factor," *IECI 1980 Proceedings*, pp. 98-106.

Peuto, B. L., and L. J. Shustek, "Current Issues in the Architecture of Microprocessors," *Computer*, February 1977, pp. 20-25.

Ripps, D. L., "Help a Real-Time Multitasking OS," *Electronic Design*, June 21, 1979, pp. 86-91.

Ripps, D. L., "A Multitasking Operating System Simplifies Real-Time Applications in Many Ways," *Electronic Design*, September 13, 1979, pp. 146-151.

Ripps, D. L., "Simplify Your Real-Time Application," *Electronic Design*, September 27, 1979, pp. 82-86.

Schreier, P. G., "Low-Cost System Requirements Multiply Interface Headaches," *EDN*, February 5, 1978, pp. 39-44.

Seim, T. A., "Numerical Interpolation for Microprocessor-Based Systems," *Computer Design*, February 1978, pp. 111-116.

Stevenson, D., "A Proposed Standard for Binary Floating-Point Arithmetic (IEEE Task P754)," *Computer*, March 1981, pp. 51-62.

Tenny, R., "Increase KIM-1 Versatility at Low Cost," *Micro*, February 1981, pp. 57-59.

Travis, T., "Patching a Program into a ROM," *Electronic Design*, September 1, 1976, pp. 98-101.

Wakerly, J. F., "Microprocessor Input/Output Architecture," *Computer*, February 1977, pp. 26-33.

Weisberg, M. J., "Designer's Guide to Testing and Troubleshooting Microprocessor-Based Products," *EDN*, March 20, 1980, pp. 177-214.

Weissberger, A. J., "Data-Link Control Chips: Bringing Order to Data Protocols," *Electronics*, June 8, 1978, pp. 104-112.

Wong, J., et al., "Software Error Checking Procedures for Data Communications Protocols," *Computer Design*, February 1979, pp. 122-125.

Zick, G. L., and T. T. Sheffer, "Remote Failure Analysis of Micro-Based Instrumentation," *Computer*, September 1977, pp. 30-35.

Note: IECI Proceedings refers to the Proceedings of the IEEE Industrial Electronics and Control Instrumentation's Annual Conference on Industrial Applications of Microprocessors, held each year since 1975. The Proceedings are available from the IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. or from the IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

Ogdin, C. A., "Some Simple Hardware [illegible]," EDN, February 20, 1979, pp. 117-[illegible].

Olivier, G., et al., "Microprocessor Controller for a [illegible] Improved Power Factor," IECI 1980 Proceedings, pp. [illegible].

Peatman, B. L., and [illegible] Shustek, "Current Issues in the [illegible] sors," Computer, February 1977, pp. 20-25.

Ripps, D. L., "[illegible] Real-Time Multitasking OS," Electronic Design, [illegible] pp. 86-[illegible].

Ripps, D. L., "A Multitasking Operating System Simplifies Real-Time [illegible] Many Ways," Electronic Design, September 13, 1979, pp. [illegible].

Ripps, D. L., "S[illegible] Your Real-Time Application," Electronic Design, [illegible] 27, 1977, pp. [illegible].

Schreier, P. G., "[illegible] System Requirements Multiply Interface [illegible]," EDN, [illegible] pp. 39-44.

Sebo, T. A., "[illegible] for Microprocessor-Based [illegible]," Computer Design, February [illegible], pp. 111-116.

Stevenson, D., "A [illegible] Standard for Binary [illegible] (IEEE Task P754)," Computer, March 1981, pp. [illegible].

Telfer, R., "Increase [illegible] Versatility at [illegible]," [illegible], February [illegible], pp. 57-59.

Travis, T., "Patching a Program Into a [illegible]," Electronic Design, September [illegible], pp. 98-[illegible].

Wakerly, J. F., "Microprocessor Input/Output Architecture," Computer, [illegible] 1977, pp. 26-33.

[illegible], M. L., "Designer's Guide to [illegible] Troubleshooting Microprocessor-Based Products," EDN, March [illegible], 1980, pp. 177-[illegible].

[illegible], A. J., "Data-Link Control Chips: Bringing Order to Data [illegible]," Electronics, June 8, 1978, pp. 104-112.

Wong, J., et al., "Software Error Checking Procedures for Data [illegible] Protocols," Computer Design, February 1979, pp. [illegible].

[illegible] G. L., and T. T. Sheffler, "Remote Failure [illegible] tion," Computer, September 1977, pp. [illegible].

[illegible] IECI Proceedings refer to the Proceedings of the IEEE Industrial Electronics and [illegible] Annual Conference on Industrial [illegible], held each year since 1975. The [illegible] available from the [illegible] Service Center, 445 [illegible], NJ 08854 or the IEEE Computer Society, [illegible] Los Vaqueros Circle, Los Alamitos, CA 90720.

Index

MICROCOMPUTER EXPERIMENTATION WITH THE MOS TECHNOLOGY KIM-1

LANCE A. LEVENTHAL

This practical, easy-to-follow, and self-contained guide to MOS Technology KIM–1 experiments was prepared for the growing population of microcomputer users representing diverse disciplines and a wide variety of applications. Its emphasis throughout is on approaches that are fundamental to the design of controllers for external systems; at the same time, it illustrates its points through examples that use nothing more complex than switches, single displays, and the on-board peripherals.

The inexpensive and widely available KIM–1 microcomputer and 6502 microprocessor were selected to provide realistic experience with popular devices for those involved in a wide range of control applications—instrumentation, communications equipment, test equipment, computer peripherals, industrial processes, signal processing, business equipment, consumer products, and more.

Author Lance Leventhal has organized his manual carefully and systematically to include an excellent overview, two major groupings of experiments, and an extensive list of references. Each chapter contains references, learning guidelines, definitions of terms, descriptions of new instructions, schematics for all interfaces, several fully-tested and documented examples (over 85 in all), and a summary of key points. No background in computer programming or digital logic is assumed. The manual includes over 270 practical problems that are closely tied to the examples; a complete fully-tested set of answers is available.

Experiments in the first group focus on writing and running simple programs, simple input and output, processing of inputs and outputs, forming and processing data arrays, designing and debugging programs, and arithmetic operations.

The second set of experiments deals with subroutines and the stack, input/output using handshakes, interrupts, timing methods, serial input/output, and microcomputer timing and control.

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

ISBN 0-13-580779-4