# Individual Learning Program

# MICROPROCESSORS

## *Unit 8*

## INTERFACING — PART 2

EE-3401

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977
Heath Company
*All Rights Reserved*
Printed in the United States of America

# CONTENTS

Introduction .................................................. 8-3

Unit Objectives .................................................. 8-4

Unit Activity Guide .................................................. 8-5

Interfacing with Switching .................................................. 8-6

The Peripheral Interface Adapter (PIA) .................................................. 8-20

Using the PIA .................................................. 8-33

Interfacing Experiments .................................................. 8-42

Unit Examination .................................................. 8-43

Examination Answers .................................................. 8-45

# *UNIT 8*

# INTERFACING — PART 2

## INTRODUCTION

In this unit, you will continue your study of interfacing the microprocessor with other circuits. The emphasis will be on interfacing with switches and displays, which are the most common input and output devices.

You will also be introduced to a special type of support IC called the peripheral interface adapter. As you will see, this device can greatly simplify many interfacing problems.

As you complete this unit, you will perform several interfacing experiments. As with Unit 7, a basic knowledge of electronics in general and digital techniques in particular is required to gain full benefit from these experiments.

# UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Draw a diagram showing how mechanical switches can be connected to an MPU.

2. Explain how the MPU can eliminate the effects of contact bounce.

3. Explain the operation of a program that detects contact closure of switches, provides for debouncing, and decodes a simple keyboard.

4. Draw a simplified block diagram of a PIA and explain the purpose of the output, control, and data direction register.

5. Write a simple program that will configure the PIA in any desired input-output combination.

6. Explain how the PIA can be used to drive displays and encode keyboards.

# UNIT ACTIVITY GUIDE

- ☐ Play Cassette Tape Section "Designing with Microprocessors."
- ☐ Read Section on Interfacing with Switches.
- ☐ Complete Self-Test Review Questions 1 through 11.
- ☐ Read Section on the Peripheral Interface Adapter (PIA).
- ☐ Complete Self-Test Review Questions 12 through 20.
- ☐ Read Section on Using the PIA.
- ☐ Complete Self-Test Review Questions 21 through 28.
- ☐ Perform Interfacing Experiments 5 through 9.
- ☐ Play Cassette Tape Section "Comparing Microprocessors."
- ☐ Complete Unit Examination.
- ☐ Check Examination Answers.
- ☐ Complete Final Examination (optional).
- ☐ Mail Final Examination in the envelope provided (optional).

# INTERFACING WITH SWITCHES

The most popular input device used with the microprocessor is the switch. The operator of microprocessor-based equipment usually communicates with the MPU by using keyboard switches. However, in completely automated systems, the equipment being controlled often communicates with the MPU by limit switches, pressure switches, etc. In this section, you will examine some techniques used in interfacing with switches.

## Interfacing Requirements

When interfacing with a switch, or switches, four operations are involved. First, the MPU must select or address the proper switch bank. Second, it must detect contact closure. Third, it must provide for debouncing (unless this is accomplished by external hardware). Finally, it must decode the input. The following information describes each of these operations in more detail.

**Selecting the Switch** Figure 8-1 shows a simple arrangement for connecting eight switches to the MPU. Three-state buffers are used to interface the switches with the data bus. The buffers are enabled by the output of the address decoder. This decoder can use any of the decoding schemes discussed earlier. Assume that the decoder responds to address $C003_{16}$. Until the decoder receives this address, the buffers are disabled. This effectively disconnects the switches from the data bus.

To find out if a switch is closed, the MPU must read in the data from this address. An easy way to do this is with the LDAA instruction. When the LDAA C003 instruction is executed, the address C003 goes out on the address bus. The decoder detects this address and enables the three-state buffer. Thus, for an instant, the switch bank is connected to the data bus. The data from the switch bank is loaded into accumulator A.

**Detecting Contact Closure** If none of the switches are closed, all the data lines will be high because of pull-up resistors $R_0$ through $R_7$. Thus, the data entered into accumulator A will be $FF_{16}$. To test for a switch closure, the contents of accumulator A can be compared with $FF_{16}$. That is, a CMPA#FF instruction could be used. If this is followed by a BNE instruction, the MPU will branch if a key is depressed. Otherwise, it will not. For example, suppose $S_0$ is closed. When the accumulator is loaded from address $C003_{16}$, the $D_0$ line will be low. Thus, the number loaded into the accumulator will be $FE_{16}$. The CMPA instruction clears the zero flag since no match occurs. Therefore, the BNE instruction causes the branch to occur.

**Debouncing the Switch** Most mechanical switches produce contact bounce. When the switch is closed, the contacts do not make an immediately solid electrical or mechanical connection. Instead, they "bounce" open and closed for a brief period of time. Figure 8-2 illustrates this effect.

While contact bounce may only last for a few milliseconds, this is long enough for the MPU to interpret the bounce as repeated switch closures. To overcome this bounce problem, some switches use cross-coupled NAND gates that will immediately latch in one state so that all contact bounce is ignored. However, this requires additional circuitry.

In many applications, a better approach is to let the MPU itself do the debouncing. A simple scheme is to wait about ten milliseconds and then read in the data from the switch bank again. If the same indication occurs, then the MPU can be certain that the switch is closed. The switch can be checked as many times as is necessary to ensure that contact bounce is eliminated.

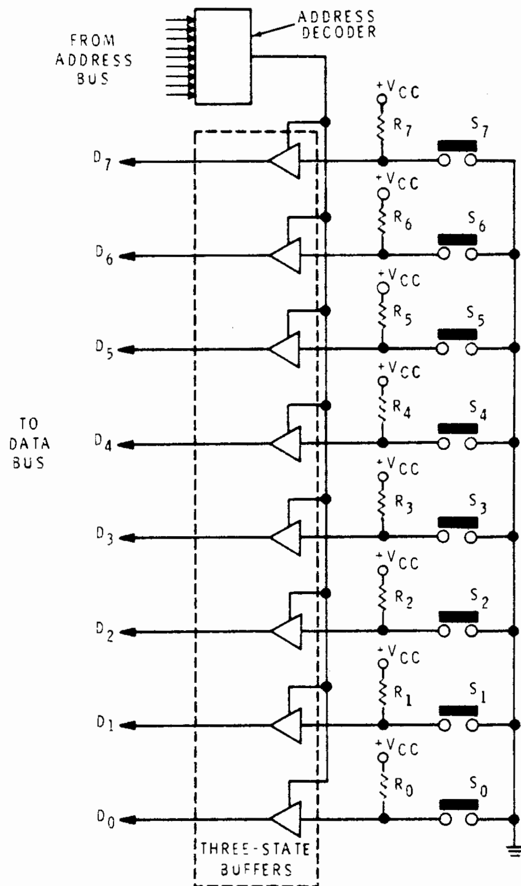


Figure 8-1
Interfacing a bank of switches to the MPU.

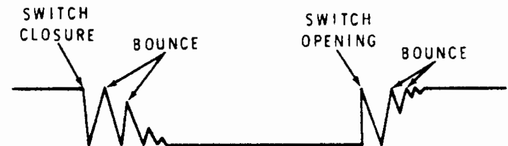


Figure 8-2
The effects of contact bounce.

**Decoding the Switches** After the MPU determines that a switch has been closed, it must decide which switch it is. In most cases the switch closure represents a number. For example, the MPU should recognize an $S_5$ closure as the number 5. This, too, is easily accomplished by the proper subroutine.

Referring to Figure 8-1, you can see that each switch corresponds to one bit of the data line. When a switch is closed, the corresponding data line goes to 0. When loaded into the accumulator, the corresponding bit is also 0. The bit that is 0 can be detected by rotating the accumulator into the carry bit until the carry bit is cleared. By counting the number of rotations, the MPU can determine which switch is depressed.

In most applications, another job of the decoding procedure is to reject multiple switch closures. If two switches are closed simultaneously, the MPU should not accept data. If a second switch is closed before the first switch is released, the MPU may reject the data or accept only the first switch closure. By using a few extra programming steps, a very simple and inexpensive keyboard can appear quite sophisticated.

## A Typical Keyboard Arrangement

The keyboard arrangement used with the ET-3400 Microprocessor Trainer is a good example of what can be done with simple switches. A simplified circuit is shown in Figure 8-3.

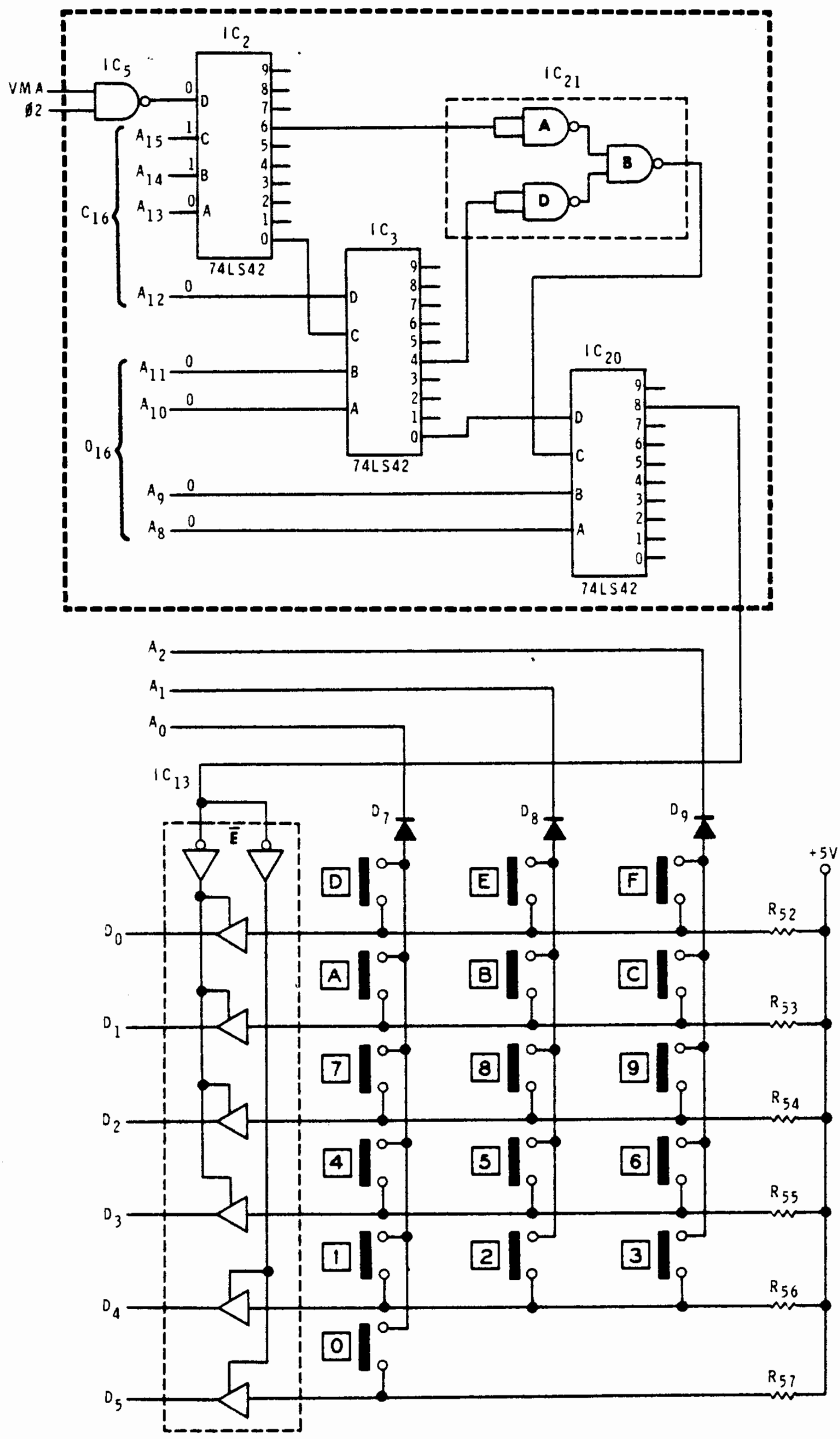


Figure 8-3
Keyboard arrangement of the ET-3400
Microprocessor Trainer.

**The Circuit** The address decoder is shown in dotted lines at the top of the Figure. For the most part it consists of three 74LS42 decoders. (The operation of this type of decoder was discussed in an earlier experiment.) The truth table for the 74LS42 is repeated in Figure 8-4.

The address decoder is also used to select the 7-segment displays. However, in this unit, we will be concerned only with keyboard decoding.

The portion of the address decoder shown in Figure 8-3 monitors address lines $A_8$ through $A_{15}$. That is, it monitors the high order address. The keyboard is selected by enabling IC13. Normally, IC13 is in its high impedance state so that the keyboard is isolated from the data bus. IC13 is enabled by applying logic 0 to the enable line ($\overline{E}$).

The address decoder enables IC13 when the high order address is $CO_{16}$. With a high order address of $CO_{16}$, address lines $A_{15}$ and $A_{14}$ are at logic 1 while $A_8$ through $A_{13}$ are at logic 0. Decoder IC2 is controlled by address lines $A_{13}$, $A_{14}$, $A_{15}$, and the VMA and $\phi 2$ clock signal. When an instruction such as LDA C003 is executed, the inputs to IC2 will be as shown in Figure 8-3. The truth table for the decoder shows that output line 6 will be logic 0 while all other output lines will be logic 1.

| DEC. NO. | BCD INPUT D | C | B | A | OUTPUT LINES 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| >9 | INVALID CODES | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 8-4
Truth table for the 74LS42 decoder.

The 1 at output 0 of IC2 is applied to IC3. Address lines $A_{10}$ through $A_{12}$ are also connected to IC3. With a high order address of CO, these lines will be at 0. With these inputs, the truth table shows that output line 4 of IC3 will be 0 while all other outputs will be 1. The 0 at output line 4 is inverted by IC21D. Simultaneously, the 0 at output 6 of IC2 is inverted by IC21A. These two signals are then NANDed together to form logic 0 at the output of IC21B. This 0 is applied to input C of IC20.

The other inputs to IC20 include a logic 1 from IC3, logic 0 from $A_9$, and logic 0 from $A_8$. The truth table shows that this will result in a logic 0 at output 8 of IC20. This logic 0 is applied to the enable input of IC13. This enables the three-state buffers and momentarily connects the keyboard to the data bus. Thus, the keyboard is momentarily connected to the data bus any time the high order address is $CO_{16}$.

The keyboard is divided into three columns. The center and right columns have five keys each while the left column has six keys. The RESET key is not shown since it is not addressed as the other keys are. The low order address determines which column of keys is selected. $A_0$ controls the left column and $A_1$ and $A_2$ control the center and right columns, respectively. A column is selected by choosing an address that will force the desired column line low, but will hold the unwanted column lines high. Address $C003_{16}$ fulfills these requirements for the right-hand column of keys. So do many other addresses, but consider this to be the address of that column. In the same way, the center column has an address of $C005_{16}$ and the left column has an address of $C006_{16}$.

**Detecting and Encoding a Key Closure** The monitor program in the ROM of the Trainer has a subroutine called ENCODE which starts at address $FDBB_{16}$. The purpose of this subroutine is to look over the keyboard, determine if a key has been depressed, and produce the proper hexadecimal value of the depressed key. The hexadecimal value is placed in accumulator A. Also, the carry flag will indicate whether or not a valid key entry has occurred. A valid entry is defined as one and only one key depressed. The C flag will be set if the entry was valid. It will be cleared for nonvalid entries or no entries at all.

The ENCODE subroutine is shown in Figure 8-5. The following explanation will refer to the instructions by the line numbers given on the left. Notice that the subroutine is written in assembly language.

The first instruction saves the original contents of accumulator B. As you will see later, the program normally comes here from another subroutine called INCH. When it does, B will hold a timing count that must not be lost.

| LINE | ASSEMBLY CODE | | | COMMENTS |
|---|---|---|---|---|
| 1 | ENCODE | PSH B | | Save contents of accumulator B. |
| 2 | | LDA B | COL1 | Load the right column into B. |
| 3 | | LDA A | COL3 | Load the left column into A. |
| 4 | | ASL A | | |
| 5 | | ASL A | | Get rid of "don't care" bits. |
| 6 | | ASL A | | |
| 7 | | ROL B | | Double precision shift left |
| 8 | | ASL A | | of accumulators A and B |
| 9 | | ROL B | | to get rid |
| 10 | | ASL A | | of "don't care" bits. |
| 11 | | ROL B | | |
| 12 | | PSH B | | Save contents of B. |
| 13 | | LDA B | COL2 | Load the center column into B. |
| 14 | | AND B | #$1F | Mask off bits 5, 6, and 7. |
| 15 | | ABA | | Merge with A. |
| 16 | | PUL B | | Restore B. |
| 17 | | COM A | | After complementing the keyboard |
| 18 | | COM B | | pattern will be in A and B (1 = key closed). |
| 19 | | STX | T0 | Save contents of index register. |
| 20 | | LDX | #HEXTAB-1 | Point index register to table of hex values. |
| 21 | | CBA | | Which accumulator contains a 1? |
| 22 | | BEQ | ENC3 | Neither or both contain 1's (invalid entry). |
| 23 | | BCC | ENC1 | A contains a 1 so go to ENC1. |
| 24 | | PSH A | | B contains a 1 so |
| 25 | | TBA | | Swap the contents |
| 26 | | PUL B | | of A and B. |
| 27 | | LDX | #HEXTAB+7 | Point the index register to upper half of hex table. |
| 28 | ENC1 | TST B | | None of these keys should be closed. |
| 29 | | BNE | ENC3 | If they are, go to ENC3. |
| 30 | ENC2 | INX | | Otherwise, have the index register |
| 31 | | ASLA | | Scan up the table until it |
| 32 | | BHI | ENC2 | finds the proper hex value. |
| 33 | | BEQ | ENC4 | If only one key is depressed, entry is valid. |
| 34 | ENC3 | CLC | | Entry is not valid so clear C. |
| 35 | ENC4 | LDA A | O,X | Load the hex value into accumulator A. |
| 36 | | LDX | T0 | Restore index register and |
| 37 | | PUL B | | accumulator B to their original values. |
| 38 | | RTS | | Return |

Figure 8-5
ENCODE subroutine.

The next two instructions load A and B from the keyboard columns. After line 3, A and B will contain the data from the two outside keyboard columns. Figure 8-6A shows which keys are associated with which bits. The indicated bit will contain 0 if its associated key is depressed. Otherwise, it will contain a 1. The X's are shown in those bits that are not affected by the keys. The first step is to eliminate these "don't care" states. In accumulator A and the carry flag, this is done by shifting to the left. After line 6, the accumulators and carry flag will contain the keyboard patterns shown in Figure 8-6B.

Next A and B are shifted left together through the carry flag. Figure 8-6C shows the contents of these registers after line 11. The contents of B are then saved by pushing them into the stack. Accumulator B is now free to be loaded with the data from the center column of the keyboard. Bits 5, 6, and 7 are masked off leaving the registers as shown in Figure 8-6D. B is added to A so that the 0's in A are replaced with the states of keys 2, 5, 8, B, and E. When B is pulled from the stack (line 16), the registers will contain the keyboard pattern as shown in Figure 8-6E. Notice that each of the $16_{10}$ keys is represented by one of the bits in the accumulators. If no keys are depressed, all bits will be 1's. If a key is depressed, its corresponding bit will be 0.

In lines 17 and 18, the two accumulators are complemented. Thus, from this point on a depressed key is represented by a 1 while an open key is represented by a 0.

Assume that the D key is depressed. In this case, the accumulators will appear as shown in Figure 8-6F after line 18. In line 19, the contents of the index register are saved in a temporary location in RAM called TO. Next the index register is loaded with one less than the starting address of a hexadecimal table.

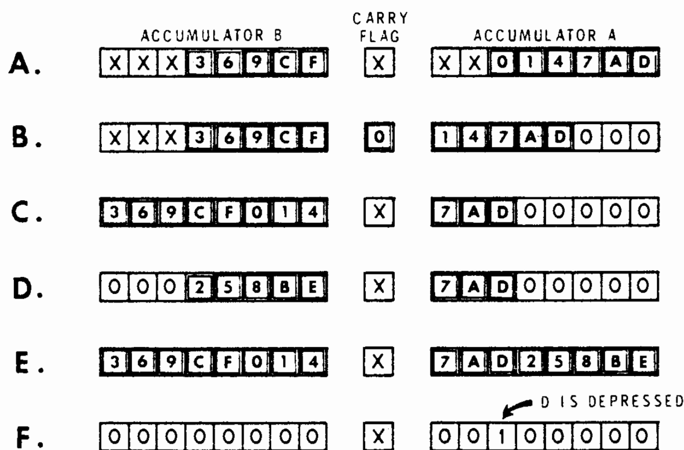


Figure 8-6
The keyboard bit pattern is placed in accumulators A and B.

The hexadecimal table is shown in Figure 8-7. Notice that the hex digits 00 through 0F are not in order in the table. However, compare the entries in the table to Figure 8-6E. The first eight entries in the table are in the same order as the keyboard pattern in accumulator A. The upper eight entries correspond to the key patterns in accumulator B. As you will see later, this is no accident.

Line 21 of the program compares the contents of accumulators A and B. With D depressed, the number in A will be larger. The result is that both the Z and C flags are cleared. The BEQ does not cause a branch because the Z flag is cleared. However, the BCC instruction does cause a branch because the C flag is cleared. Notice that the branch is to the point labelled ENC1 (line 28).

| HEX ADDRESS | HEX CONTENTS | SYMBOLIC ADDRESS |
|---|---|---|
| FFA6 | 07 | HEXTAB |
| FFA7 | 0A | |
| FFA8 | 0D | |
| FFA9 | 02 | |
| FFAA | 05 | |
| FFAB | 08 | |
| FFAC | 0B | |
| FFAD | 0E | |
| FFAE | 03 | |
| FFAF | 06 | |
| FFB0 | 09 | |
| FFB1 | 0C | |
| FFB2 | 0F | |
| FFB3 | 00 | |
| FFB4 | 01 | |
| FFB5 | 04 | |

Figure 8-7
The hexadecimal table (HEXTAB).

This part of the subroutine encodes the key closure and at the same time checks to see that no other keys are closed. The first step tests B. If the result is not zero, the MPU knows that a second key is closed and the entry is ignored by branching to ENC3. (The result of this will be shown later). Otherwise, the index register is incremented to the address of the first entry in the hex table, the contents of accumulator A is then shifted to the left, and the BHI instruction simultaneously checks both the C and Z flags. The branch is implemented if both C and Z are cleared. Both are cleared in this case so the program jumps back to ENC2 and the index register is incremented so that it points to the second entry in the table. Accumulator A is shifted left again. This places the 1 (that represents switch D being closed) into the MSB of the accumulator. C and Z are still cleared so the BHI instruction sends the program back to ENC2 again.

The index register is incremented again so that it now points to the third entry in the hex table. Notice that the third entry is $OD_{16}$. Thus, the index register is now pointing to the number that corresponds to the switch closure. Accumulator A is shifted left again so that the 1 (representing switch D) is placed in the carry flag. This sets the C flag and, consequently, the BHI instruction cannot cause a branch.

Because the BHI branch does not occur, the next instruction encountered is the BEQ instruction. If only one key is depressed, the contents of accumulator A should be zero. If it is zero, then only one key was depressed. Otherwise, a second key was depressed and the entry should be tagged not valid. If the entry is valid, the BEQ instruction causes the program to jump over the CLC instruction to ENC4 (line 35).

At ENC4, accumulator A is loaded with the third entry from the hex table. Thus, the program has fulfilled its requirements. A number corresponding to the key depressed is in accumulator A and the C flag is set indicating a valid entry. All that remains is to restore the original contents of the index register and accumulator B. Finally, the RTS instruction returns the program to the point where this subroutine was called.

If you step through this subroutine with a different key depressed, you will see that the proper hex code is always returned. If no keys are depressed or if two keys are depressed, the program will branch to ENC3. This clears the carry flag. Thus, the subroutine will end with C cleared, which indicates that the entry was not valid.

**Eliminating Contact Bounce** A second subroutine is used to eliminate contact bounce. It is called INCH for "input character." Its starting address is FDF4 and it calls the ENCODE program just discussed $20_{16}$ different times. If for $20_{16}$ consecutive times, ENCODE tells INCH that a valid entry exists, INCH is convinced and accepts the entry. Since this process requires several milliseconds, any contact bounce is eliminated.

This subroutine is shown in Figure 8-8. The subroutine is divided into two nearly identical halves. The first half (lines 1 through 6) waits for the keyboard to be cleared (no keys depressed). It does this so that it does not mistake a previous key closure for a valid entry.

If a key is depressed upon entering this subroutine, first, the contents of B are saved; then B is loaded with a delaying count of $20_{16}$. By counting this number down to zero, the program establishes a delay sufficient to "wait out" any contact bounce.

| LINE | ASSEMBLY CODE | | | COMMENTS |
|---|---|---|---|---|
| 1 | INCH | PSH B | | Save contents of accumulator B |
| 2 | INC1 | LDA B | #$20 | Load B with a delaying count of $20_{16}$. |
| 3 | INC2 | BSR | ENCODE | Branch to the ENCODE subroutine. |
| 4 | | BCS | INC1 | If carry is set go to INC1. |
| 5 | | DEC B | | Otherwise decrement the count. |
| 6 | | BNE | INC2 | If count is not zero go back to INC2. |
| 7 | INC3 | LDA B | #$20 | Load B with a delaying count of $20_{16}$. |
| 8 | INC4 | BSR | ENCODE | Branch to the ENCODE subroutine. |
| 9 | | BCC | INC3 | If carry is clear, check again. |
| 10 | | DEC B | | Otherwise, decrement the count. |
| 11 | | BNE | INC4 | If count is not zero go back to INC4 |
| 12 | | PUL B | | Restore the original contents of accumulator B. |
| 13 | | RTS | | Return. |

Figure 8-8
INCH subroutine.

The BSR instruction sends the MPU off to the ENCODE subroutine. If a key is depressed, the C flag will be set when the program returns. The BCS instruction sends the program back to INC1 if the C flag is set. The program will stay in this loop until the ENCODE subroutine returns with the carry bit clear. Of course, this will happen only after a key is released. This prevents a single entry from being mistaken for two or more entries. An entry is accepted only after the previous entry is released.

Once a key is released, ENCODE will clear the carry flag. Thus, the BCS instruction will not cause a branch. Instead B is decremented and checked for zero. If not zero, the program branches back to INC2. This loop is repeated $20_{16}$ times and gets rid of any contact bounce associated with the release of the previous key. Once the program is convinced that the previous key has been released, it proceeds to the second half of the subroutine (lines 7 through 13).

Accumulator B is loaded with a delaying count of $20_{16}$ again and the BSR instruction calls the ENCODE subroutine. Upon return from this subroutine, the BCC instruction checks the carry flag. If it is clear, no valid entry is being received and the program branches back to INC3. In practice, the MPU in the ET-3400 Trainer spends most of its time caught in this loop waiting for a key closure to occur.

When a key closure does occur, the ENCODE subroutine sets the carry bit. This allows the MPU to escape the loop. It then enters the loop composed of lines 8 through 11. It repeats this loop $20_{16}$ times to eliminate any contact bounce. When it escapes this loop, we can be confident that the key closure is absolutely valid.

This is a good example of software-hardware trade offs. These subroutines make the keyboard appear quite sophisticated. A mechanical or electromechanical keyboard having all these features would be very expensive.

You will learn more about interfacing with switches later after you learn about the peripheral interface adapter. Also, in a later experiment you will gain some practical experience interfacing switches.

## Self-Test Review

1. List four requirements that must be met when connecting mechanical switches to the MPU.

2. What type of circuit is often used between the switches and the data bus of the MPU?

3. Refer to Figure 8-1. If no switches are closed, what hexadecimal number is read from the switch bank?

4. Refer to Figure 8-1. If $S_2$ is closed, what hexadecimal number is read from the switch bank?

5. What two things can be determined by the hexadecimal number read in from the switch bank?

6. Why is debouncing required?

7. How can the MPU overcome the effects of contact bounce?

8. Refer to Figure 8-3. To what address does the center row of keys respond?

9. Refer to Figure 8-5. What is the purpose of the first 18 lines of the program?

10. What technique is used for finding the hexadecimal equivalent of the key that is depressed?

11. Refer to Figure 8-8. How does this routine overcome contact bounce?

## ANSWERS

1. The MPU must:
   1) Address the switches.
   2) Detect contact closure.
   3) Overcome contact bounce.
   4) Decode the switches.

2. A three-state buffer.

3. $FF_{16}$.

4. $FB_{16}$.

5. The hexadecimal number read from the switch bank reveals
   1) If a switch is closed.
   2) Which switch is closed.

6. When a mechanical switch is closed, its contacts often bounce open and closed several times. Unless this is taken into consideration, a single switch closure can be mistaken for multiple switch closures.

7. The MPU can overcome the effects of contact bounce by rechecking the closed switch several times.

8. $C005_{16}$.

9. The first 18 lines of the program place the switch pattern into accumulators A and B.

10. They are looked up in a table.

11. By rechecking the closed switch. It must find the switch closed for $20_{16}$ consecutive checks before the value is accepted as good.

# THE PERIPHERAL INTERFACE ADAPTER (PIA)

Most microprocessors have a family of support chips that are used to simplify the problem of interfacing with the outside world. One of the most popular of these interfacing chips is the 6820 peripheral interface adapter (PIA). The PIA was developed to support the 6800 MPU. However, it is also being used in many microprocessor based designs using other MPU's.

While a complete discussion of support chips is beyond the scope of this course, detailed data sheets on several support IC's are included in Appendix B.

In this section, you will be introduced to the PIA. You will learn enough about it to use it in the experiments that follow.

The block diagrams of two typical systems that use PIA's are shown in Figure 8-9. In Figure 8-9A a single PIA is used to drive both an input and an output device. This is possible since the PIA has two independent channels. Figure 8-9B shows a system that uses two PIA's. One controls an input device while the other controls an output device.

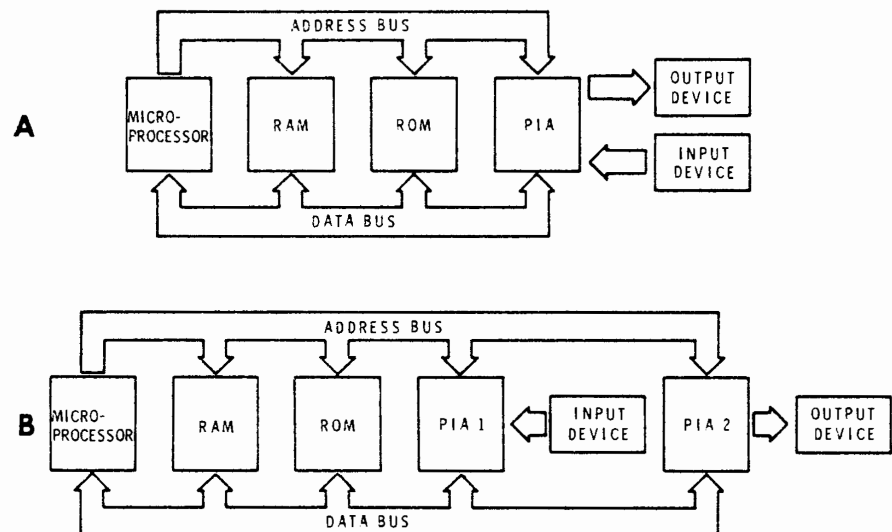


Figure 8-9
The PIA is used to interface input and output devices to the MPU.

The purpose of the PIA is to simplify the problem of interfacing the MPU to external devices. Of course, any device can be interfaced with the MPU using conventional combinational logic. However, the conventional logic approach generally requires many IC's. This defeats one of the prime advantages of the microprocessor — a simple straightforward design requiring few IC's. The advantage of the PIA is that in many cases one or two IC's can do all the interfacing.

Because the PIA can do most routine peripheral control tasks, the MPU is freed to handle more important tasks. Also, the PIA allows the MPU to treat a peripheral device as a memory location. In addition, it acts as a buffer between the high-speed MPU and the low-speed I/O device. Since the PIA has some on board address decoding, a separate address decoder is not needed in many applications.

The PIA is superior to combinational logic in another way. The PIA is extremely flexible because it is programmable. That is, its configuration can be changed from one moment to the next by the program being executed. For example, an output port can be changed to an input port in the middle of a program. Later, you will see how this is done. But first, you will learn about the internal structure of the PIA.

## I/O Diagram

The diagram of the PIA shown in Figure 8-10 shows its input and output lines. Since this is an interface device, one side connects to the MPU while the other side connects to one or more peripheral devices.

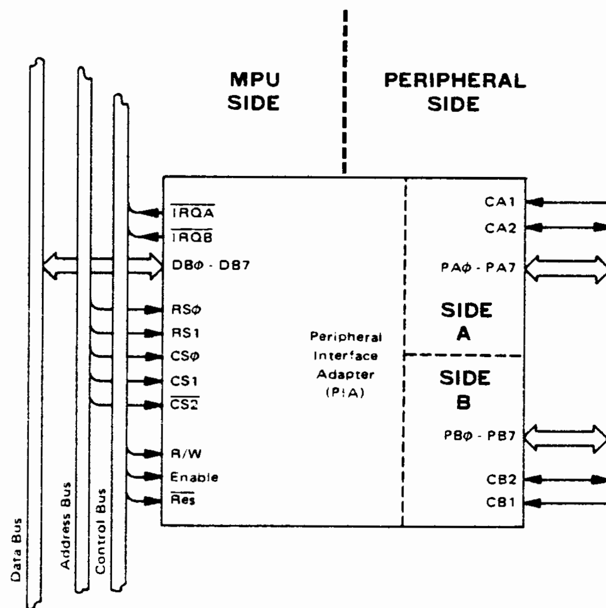


Figure 8-10
I/O diagram of the peripheral interface adapter.

On the MPU side, the PIA monitors several address, data, and control lines of the MPU. It monitors all eight data bus lines. Data is transferred to and from the PIA, a byte at a time, by the data bus. For this reason, the PIA is said to be "byte oriented." In this respect, the PIA is treated like memory. As you will see later, the PIA has four addresses that can be treated much like RAM.

The PIA can monitor five of the MPU's address lines. This is enough to partially decode the address. In many cases, no additional address decoding is necessary.

The PIA also connects to several control lines. The $R/\overline{W}$ line informs the PIA whether it is to receive data from the MPU or send data to the MPU. Once again, note the similarity between the PIA and RAM.

Another similarity is the enable line. Like RAM, the PIA is enabled by the $\phi 2$ clock (often ANDed with VMA). This provides the basic timing signal for the PIA.

The reset line of the PIA is generally connected to the master system reset. This allows the PIA registers to be reset to a known condition at the same time the MPU is reset.

The PIA has two interrupt request lines. These allow the PIA to request service from the MPU. These may connect to either the IRQ or the NMI lines of the MPU. As you saw earlier, interrupts can be used to simplify I/O operations and to save MPU time. While the interrupt capability of the PIA is not discussed in this unit, the PIA data sheet in Appendix B briefly outlines these capabilities.

The peripheral side of the MPU has two nearly identical I/O channels. $PA_0$ through $PA_7$ make up a peripheral data bus for the A side of the PIA. $CA_1$ and $CA_2$ are two control/interface lines associated with the A side. Notice that the B side has comparable data and control lines. Do not confuse the peripheral data buses on the right with the MPU data bus on the left. Both buses will be referred to frequently in the following discussion.

# PIA Registers

The A and B sides of the PIA are nearly identical. Except when noted, everything stated about one side of the PIA also applies to the other side.

Each side of the PIA has three main registers as shown in Figure 8-11. These include an output register (OR), a data direction register (DDR), and a control register (CR). The output register is used to hold a data byte that is being transferred to the peripheral data bus. It acts as a temporary storage location for data being transferred from the MPU to the peripheral device.

The data direction register sets up the individual lines in the peripheral data bus as either inputs or outputs. Each bit in the DDR controls the corresponding peripheral data line. A 1 in a specific bit of the DDR causes the corresponding peripheral data line to act as an output line. A 0 causes it to act as an input line.

To set up all eight peripheral data lines of the A side as inputs, we simply store $00_{16}$ in the DDR of the A side. By the same token, we can set up the B side as output data lines by storing $FF_{16}$ in the DDR of the B side. The various data lines can be set up in any combination. Moreover, a peripheral data line can be changed from input to output simply by changing its corresponding bit in the data direction register. Keep in mind, that this change is made by software. No hardware change is necessary.

The control register allows you to program several other characteristics of the PIA. One of these will be discussed later. The others are explained in the PIA data sheets in Appendix B.

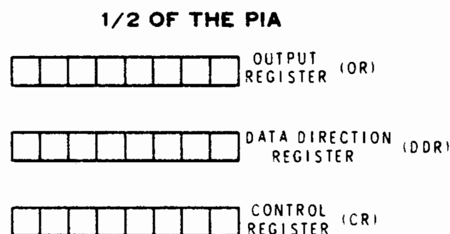


Figure 8-11
Each side of the PIA has three main registers.

## Addressing the PIA Registers

The PIA has six registers in which data can be stored and from which data can be read. The two control registers each have an address of their own. In a typical system, the control register on the A side may have an address of $4005_{16}$. The control register on the B side may have an address of $4007_{16}$. In this case, we could write data into the control register on the A side with the instruction STAA $4005_{16}$. Or, we could read from the control register on the B side with LDAA $4007_{16}$.

While the control register has an address of its own, the data direction register and the output register share a common address. Typically, the data direction and output register on the A side may share the address $4004_{16}$. Those on the B side may share the address $4006_{16}$. Thus, in a typical system, the PIA may have addresses assigned as shown in Figure 8-12.

Even though the data direction and output registers share the same address, each is still individually accessible. When address 4004 appears on the address bus, either ORA or DDRA will be selected depending on bit 2 of the control register. If bit 2 of CRA is a 1, then address $4004_{16}$ selects ORA. However, if bit 2 of CRA is 0, then address 4004 selects DDRA. In this same way, bit 2 of CRB determines which register is selected by address $4006_{16}$. A 1 selects the output register; a 0 selects the data direction register.

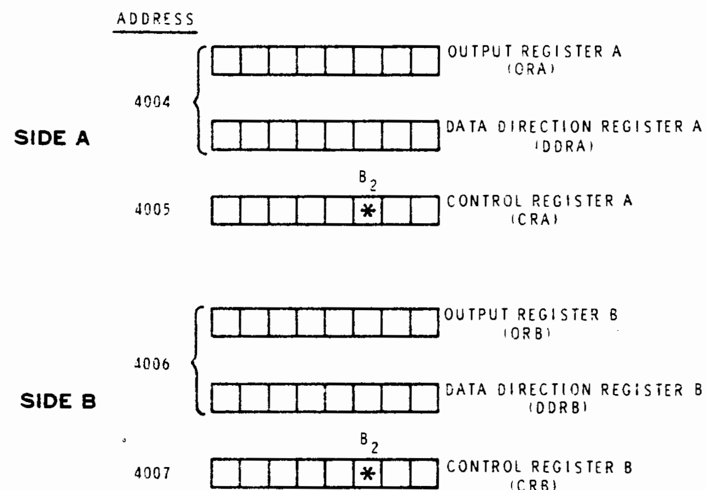


Figure 8-12
Typical address assignments of PIA registers.

# Initializing the PIA

Before the PIA can be used for input-output transfers, it must first be programmed to operate in the desired manner. For example, assume that you wish to use the A side of the PIA as an output port and the B side as an input port. Figure 8-13 shows a PIA that is configured in this manner. DDRA sets all A-side peripheral data lines as outputs and bit 2 of CRA has set the output register to respond to address 4004. DDRB sets all B-side peripheral data lines as inputs and bit 2 of CRB has set ORB to respond to address 4006. This state does not come about by accident. We must deliberately set up these conditions. This process is called initializing the PIA.

In most applications, the PIA is initialized after a system is reset. Once configured in a certain way, the PIA is normally left in this configuration. For this reason, you can assume that the initialization process starts immediately after the system is reset.

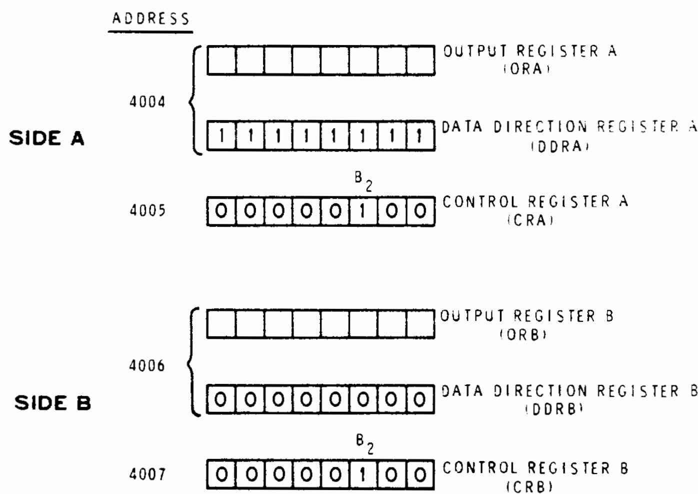


Figure 8-13
The A side is configured as an output port; the B side, as an input port.

The PIA has a reset line that is normally connected to the system reset line. When the PIAs' reset line goes low, all the registers in the PIA are reset to zero as shown in Figure 8-14. With both data direction registers reset to zero, both peripheral data buses are configured as inputs. Also, with bit 2 of the control registers reset to 0, address 4004 selects DDRA while 4006 selects DDRB. To initialize the PIA, we must change the contents of its registers from that shown in Figure 8-14 to that shown in Figure 8-13.

A program that will accomplish this is:

```
LDAA    #FF
STAA    4004
LDAA    #04
STAA    4005
STAA    4007
```

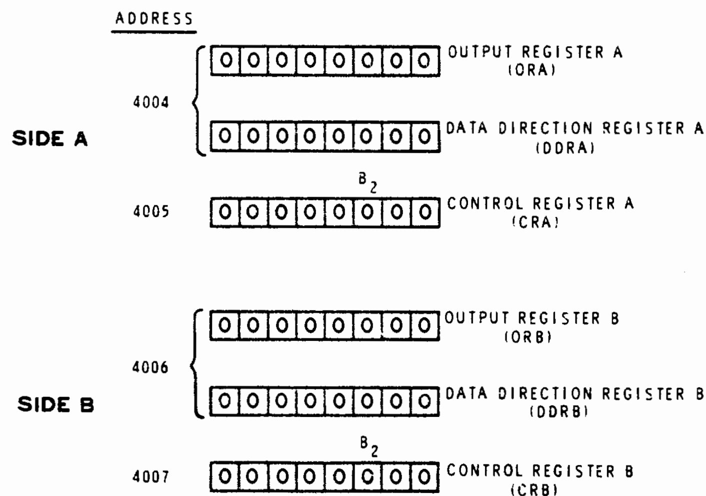


Figure 8-14
When the PIA is reset, all registers are set to zero.

The first instruction loads accumulator A with all binary 1's. The second instruction stores this at location $4004_{16}$. Since bit 2 of CRA is initially 0, FF is stored in DDRA. This configures the A-side peripheral data bus as outputs.

The third instruction loads $04_{16}$ into accumulator A. The next two instructions store $04_{16}$ at addresses $4005_{16}$ and $4007_{16}$. These addresses are the control registers. Recall that $04_{16}$ is equal to $0000\ 0100_2$. This sets bit 2 of both control registers to 1. Consequently, address $4004_{16}$ now specifies ORA while $4006_{16}$ now specifies ORB.

As you can see, the PIA is now set up as shown in Figure 8-15. Notice that we did not have to change the contents of DDRB in this case because it was initially reset to zero.

Once the PIA is configured in this manner, the MPU can transfer data to the output port using the instruction: STAA $4004_{16}$. Also, it can read data from the input port using the instruction: LDAA $4006_{16}$.

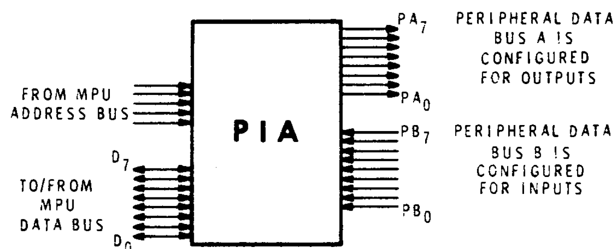


Figure 8-15
The initialization procedure configures the PIA as shown.

# Addressing the PIA

Figure 8-16 shows how the PIA fits into a microprocessor system. Now consider how the PIA is addressed.

The PIA has three chip select lines (CS0, CS1, and $\overline{CS2}$). These three lines are used to select the PIA. In order for this particular PIA to be selected, CS0 and CS1 must be high while $\overline{CS2}$ must be low.

Notice that these three lines are connected to three of the address lines (A2, A14, and A15). In this application, the A14 line is ANDed with the VMA line. This ensures that the PIA is selected only if the address is valid. In the following discussion, assume that all addresses are valid.

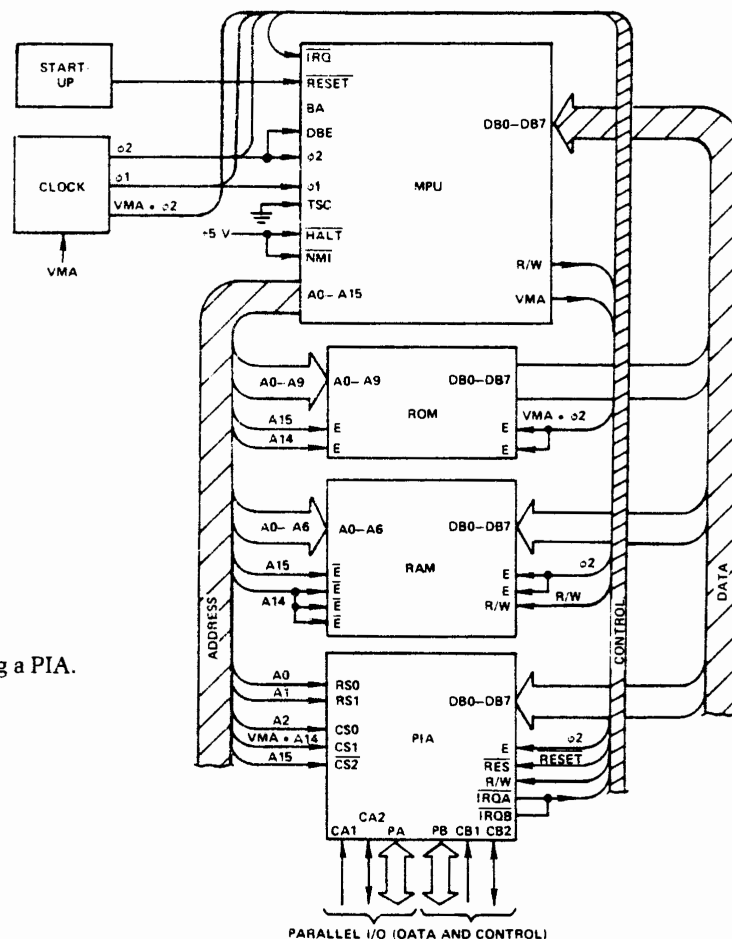


Figure 8-16
A microprocessor system using a PIA.

The PIA will be selected by any address in which A2 and A14 are high and A15 is low. Hexadecimal addresses like 4004, 5004, 6004, 7004, etc. will select the PIA because the above conditions are met. Actually, there are thousands of addresses that will select the PIA. Even so, in many systems this is no problem. In the application shown, any address below $3FFF_{16}$ will select the RAM. Many addresses between $4004_{16}$ and $7FFF_{16}$ select the PIA. Finally addresses above $C000_{16}$ select the ROM. Neither the ROM, the RAM, nor the PIA is fully decoded. Even so, their addresses are unique enough that each can be selected without additional decoding.

For programming purposes, we must assign the PIA four consecutive addresses. We will assume that these addresses are $4004_{16}$ through $4007_{16}$. Notice that we could have just as easily selected addresses $6004_{16}$ through $6007_{16}$.

In addition to the chip select lines, the PIA has two register select lines (RS0 and RS1) that also connect to the address bus. RS0 connects to A0 while RS1 connects to A1. The RS1 line determines which side of the PIA is selected. When RS1 is at logic 0, side A is selected. When RS1 is at logic 1, side B is selected.

The RS0 line selects the register on the affected side. When RS0 is 1, the control register is selected. When RS0 is 0, the data direction register or the output register is selected depending on the state of bit 2 of the control register.

| HEX ADDRESS | BINARY EQUIVALENT OF LAST HEX DIGIT | | | | REGISTER SELECTED |
|---|---|---|---|---|---|
| | | CS0 | RS1 | RS0 | |
| 4004 | 0 | 1 | 0 | 0 | DDRA or ORA* |
| 4005 | 0 | 1 | 0 | 1 | CRA |
| 4006 | 0 | 1 | 1 | 0 | DDRB or ORB* |
| 4007 | 0 | 1 | 1 | 1 | CRB |

*Determined by bit 2 of CR
0 = DDR
1 = ORA

Figure 8-17
The relationship between the address and the register selected.

The chart shown in Figure 8-17 shows why the various registers are selected for the addresses shown. For example, when the address is $4007_{16}$, address lines 1 and 0 are both high. Thus, both RS0 and RS1 are at logic 1. The 1 at RS1 selects the B side of the PIA. The 1 at RS0 selects the control register. Thus, the address $4007_{16}$ selects the control register on the B side. Figure 8-18 illustrates the same thing in another way. It shows how the data path between the MPU and the PIA register is determined.

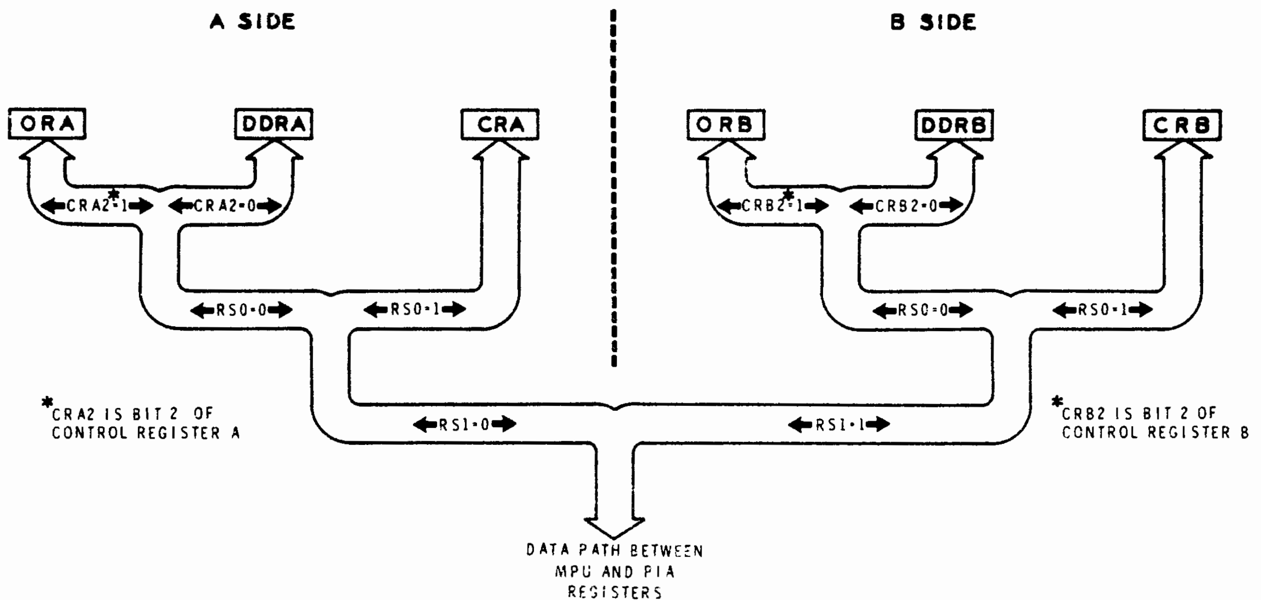


Figure 8-18
How the data path between the MPU and the PIA register is determined.

## Self-Test Review

12. What is the peripheral interface adapter (PIA)?

13. How is the PIA superior to combinational logic?

14. What is the internal structure of the PIA?

15. How is the PIA reset?

16. What are the contents of the PIA registers immediately after being reset?

17. How does the MPU decide which side of the PIA is selected?

18. Once the MPU has selected one side of the PIA, how does it determine which of the three registers is connected to the data bus?

19. Which pin of the PIA is normally connected to the A1 address line of the MPU?

20. Refer to Figure 8-16. Write a short routine that will initialize the PIA immediately after reset. Set up PA0 through PA3 and PB0 through PB3 as inputs. Set up PA4 through PA7 and PB4 through PB7 as outputs.

## ANSWERS

12. The PIA is a 40-pin IC that is used to simplify the transfer of data between a microprocessor and the outside world.

13. In many cases, one or two PIAs can handle all interfacing requirements. Also, the PIA is extremely flexible since it can be programmed to perform in several different configurations.

14. The PIA has two nearly identical sides, each of which contains three registers: the output register, the data direction register, and the control register. The MPU can transfer data to or from either of these registers by way of the data bus.

15. The PIA is reset by pulling its $\overline{\text{reset}}$ line low.

16. When reset, the contents of all PIA registers are reset to $00_{16}$.

17. The MPU selects the A side of the PIA by switching the PIA's RS1 line to 0 (low). The B side is selected by switching the RS1 line to 1.

18. If the RS0 line of the PIA is 1, the control register of the affected side is selected. If RS0 is 0, then the register selection is determined by bit 2 of the affected control register.

19. RS1 of the PIA is normally connected to address line A1 of the MPU.

20. A typical routine is:

```
LDAA    #F0
STAA    4004
STAA    4006
LDAA    #04
STAA    4005
STAA    4007
```

# USING THE PIA

Now that you are familiar with the PIA, you are ready to examine some of the ways that the PIA can be used. In this section, you will see how the PIA can be used to handle displays and keyboards. You will start by examining how a PIA can be used to drive 7-segment displays.

## Driving 7-Segment Displays

Figure 8-19 shows how a single PIA can be used to multiplex up to eight 7-segment displays. The PIA is configured to respond to addresses $4004_{16}$ through $4007_{16}$. The A side of the PIA is used to supply the 7-segment code to the displays. Inverters are used to supply the current required by the displays. The B side is used to determine which display is selected. Here discrete transistors are used to provide the required current.

The displays are common cathode types. To light segment "a" of display 1, $Q_1$ must conduct through pin "a." Thus, a logic 1 must be applied to the base of $Q_1$ and to pin "a" of display 1.

Notice that both sides of the PIA must serve as outputs. Thus, during the initialization procedure, both data direction registers are set to $FF_{16}$. Then bit 2 of both control registers are set to 1's, so that data would be routed to the output registers.

All displays are blanked by storing $FF_{16}$ in output register A. This sets PA0 through PA7 to 1's. The 1's are then inverted to 0's. Thus, no segments of any display can light.

To display a specific eight-character message, the displays must be turned on one at a time in sequence. This is controlled by the B side of the PIA. At the same time, the 7-segment character codes must be loaded into the A side of the PIA one at a time.

HEATHKIT CONTINUING EDUCATION

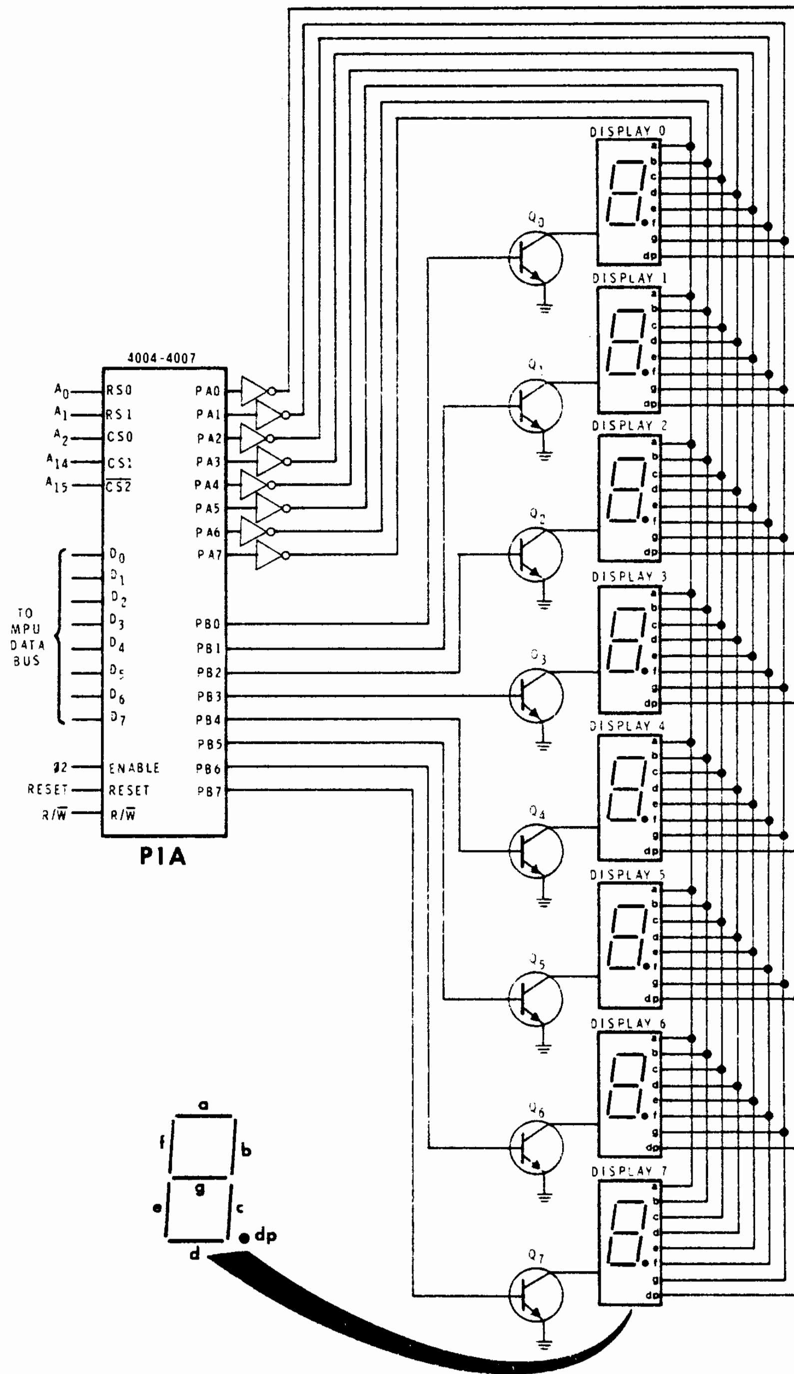


Figure 8-19
Using the PIA to multiplex displays.

A subroutine for multiplexing the displays is shown in Figure 8-20. It assumes that the eight 7-segment codes are already in RAM at consecutive addresses, SGCODE through SGCODE +7.

The first instruction loads the index register with one less than the address of the 7-segment code for the first character. Next, accumulator B is cleared and the carry flag is set to 1. Accumulator A is set to $FF_{16}$ by first clearing to $00_{16}$ and then complementing. The $FF_{16}$ in accumulator A is stored in the output register of the A side of the PIA. This sets PA0 through PA7 to 1's. The ones are inverted and blank all the displays. The displays are blanked by side A whenever the display pointer (side B) is being changed.

The next instructions rotate accumulator B to the left through the carry flag. Recall that accumulator B was originally cleared and that the carry flag was set. Thus, the 1 in the carry flag is rotated into bit 0 of accumulator B. The new contents of accumulator B are stored in the B side of the PIA. PB0 is set to 1 while PB1 through PB7 are reset to 0. The 1 at PB0 enables display 0. However, the display still does not light because the segment lines are still at logic 0.

| LINE | ASSEMBLY CODE | | | COMMENTS |
|---|---|---|---|---|
| 1 | DISPLAY | LDX | #SGCODE−1 | Point to first code minus one. |
| 2 | | CLRB | | Initialize the |
| 3 | | SEC | | display pointer |
| 4 | NXDIGIT | CLRA | | Set ACCA |
| 5 | | COMA | | to FF |
| 6 | | STAA | PIAORA | Turn off all displays |
| 7 | | ROLB | | Point to next display in sequence. |
| 8 | | STAB | PIAORB | Enable next display in sequence. |
| 9 | | BCC | NXTDSP | If last display has been lit, |
| 10 | | RTS | | exit. Otherwise, |
| 11 | LTDSP | INX | | point to next 7-segment code |
| 12 | | LDAA | O,X | Load code into ACCA. |
| 13 | | STAA | PIAORA | Display the code. |
| 14 | | CLRA | | Leave this |
| 15 | DELAY | INCA | | display lit for |
| 16 | | BNE | DELAY | $1536_{10}$ MPU cycles. |
| 17 | | BRA | NXDIGIT | Go back and do it again. |

Figure 8-20
Subroutine for multiplexing the displays.

The next instruction (BCC) checks the carry flag to see if it is cleared. It will be in this case because a 0 was rotated into it by the earlier ROLB instruction. Consequently, the RTS instruction is skipped over and the INX instruction is executed next.

The INX instruction increments the contents of the index register so that it now points at the 7-segment code for the first character that is to be displayed. The next instruction loads this character into accumulator A. Then the 7-segment code is stored in output register A of the PIA. The code is inverted and is applied to the segment lines of all eight displays. However, only display 0 is presently enabled. Thus, this is the only display that will be lit.

The next three instructions cause a delay of about $1536_{10}$ MPU cycles. In a typical system, this amounts to a delay of about 2 milliseconds. Finally, the BRA instruction branches the program back to the point called NXDIGIT.

At NXDIGIT, the displays are blanked again. Accumulator B is rotated to the left so that the 1 now appears at bit 1. This is stored at PIAORB, enabling display 1 and disabling all other displays. The carry flag is still cleared, so the RTS instruction is skipped. The next 7-segment code is selected and stored at PIAORA. Thus, the second code lights display 1. The display is held lit for about 2 milliseconds and the loop is repeated again.

The display loop continues until all eight displays have been lit. After the final display is lit, the program tries to repeat the loop again. However, this time, the 1 in accumulator B is rotated back into the carry flag. As a result, the BCC instruction does not cause a branch. The RTS instruction is executed and the program returns to wherever it came from.

In order to give the illusion of a constant display, this subroutine must be called several times each second. The display must be constantly refreshed by rewriting the same message over and over again.

# Decoding Keyboards

Figure 8-21 illustrates how the PIA can be used to decode a 16-switch keyboard. The chip select lines are connected so that this PIA responds to addresses $4008_{16}$ through $400B_{16}$.

One switch is connected to each of the peripheral data lines of the PIA. When a switch is open, its corresponding peripheral data line is pulled up to logic 1 by the pull-up resistor. When a switch is closed, the corresponding peripheral data line falls to logic 0. In this application, both sides of the PIA act as input ports. Since they are automatically set up as inputs during reset, there is little to be done during initialization. Of course, bits 2 of the control registers must be set to 1 so that the input data from the keyboard can be read from addresses $4008_{16}$ and $400A_{16}$.

The problems associated with decoding the keyboard are the same as those discussed earlier. Because this keyboard does not use interrupts, the MPU must scan the keyboard at regular intervals. Typically, it would read from addresses $4008_{16}$ and $400A_{16}$ several times each second.

The MPU detects a switch closure by comparing the input data with $FF_{16}$. If the input data is anything other than $FF_{16}$, one (or more) of the switches is closed.

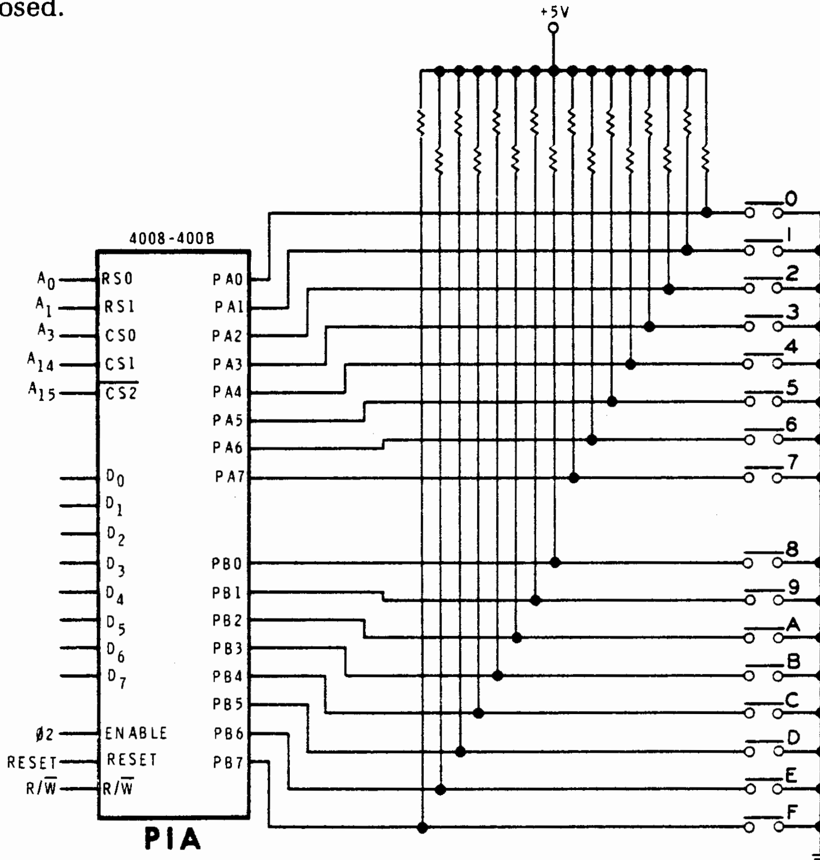


Figure 8-21
Using the PIA to monitor a keyboard.

The MPU overcomes the switch bounce problem in the same way discussed earlier. Also, the problems of rejecting multiple switch closures and producing an equivalent binary code can be accomplished using the same techniques discussed previously in this unit.

## Decoding a Switch Matrix

The method shown in Figure 8-21 is a very straightforward technique of decoding switches. Using one PIA, this technique can handle up to 16 switches. There are other techniques that use the PIA to greater advantage. An example is shown in Figure 8-22. Here again, the PIA is handling 16 switches. However, this time only one side of the PIA is used.

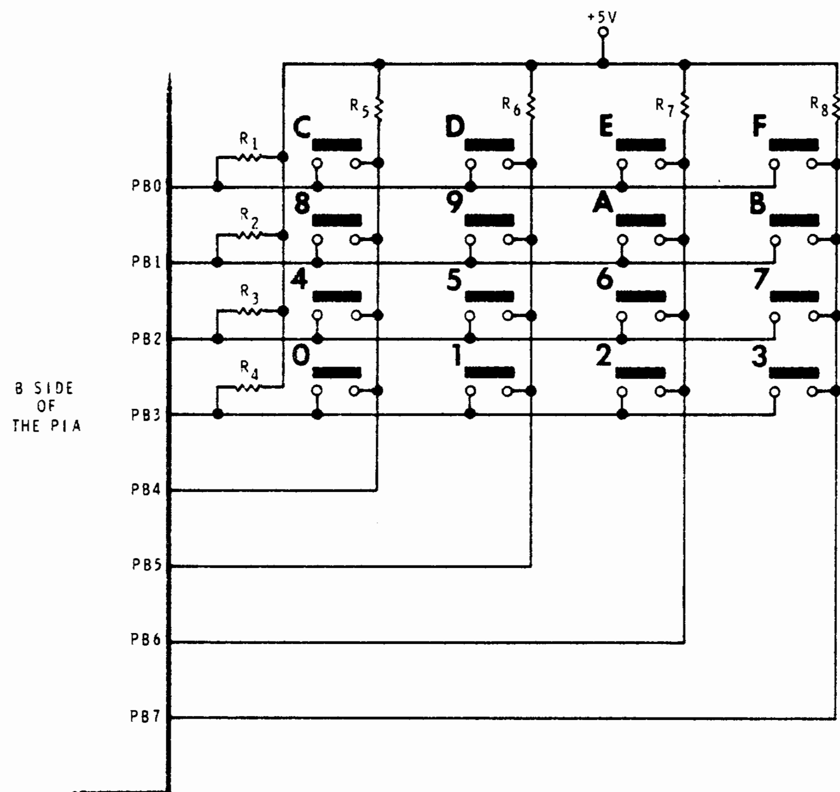


Figure 8-22
One side of the PIA can handle up to 16 switches.

The 16 switches are arranged in a 4 by 4 matrix. The B side of the PIA is used to interface with the switches. Lines PB0 through PB3 are configured as input lines while PB4 through PB7 are configured as output lines.

With no switches closed, PB0 through PB3 are pulled up to logic 1 by resistors $R_1$ through $R_4$. PB0 monitors switches C, D, E, and F. When all four switches are open, PB0 is at logic 1. Even with one of the switches closed, PB0 will still be at logic 1 if output lines PB4 through PB7 are at logic 1. And, these output lines (PB4 through PB7) are normally held at logic 1.

Periodically, the MPU scans the keyboard to see if any switch has been closed. It does this by applying logic 0 to one of the output lines and then checking for a logic 0 at one of the input lines.

A typical procedure might look like this. When the PIA is initialized, PB0 through PB3 are set up as inputs while PB4 through PB7 are set up as outputs. The MPU scans the keyboard in this manner. PB4 is reset to 0 by storing $EF_{16}$ in output register B. Next, the B side is read out. If switch 0, 4, 8, or C is closed, its corresponding PIA input line will be low. For example, if switch 8 is closed, the 0 at PB4 will pull PB1 low. By examining lines PB0 through PB3, the MPU can tell which switch (if any) is closed.

If no switch is closed in the first column, PB5 is reset to 0 by storing $DF_{16}$ in output register B. The MPU can now check to see if switch 1, 5, 9, or D is closed.

The technique just described allows the MPU to handle a large number of switches with a single PIA. Using both sides of the PIA, the MPU could handle an 8 by 8 matrix of $64_{10}$ switches. This technique will be explored in more detail in an interfacing experiment.

## Self-Test Review

21. Refer to Figure 8-19. Write a short program that will initialize the PIA in the proper configuration.

22. Refer to Figure 8-19. Which side of the PIA determines which display is selected?

23. Refer to Figure 8-19. Assume that the PIA has been properly initialized. Write a simple program segment that will display the numeral 4 on display number 7.

24. Refer to Figure 8-21. In order for this scheme to work, both sides of the PIA must be configured as ________.

25. Refer to Figure 8-21. How does the MPU tell that switch 9 is closed?

26. How can the B side of the PIA be set up as shown in Figure 8-22?

27. Refer to Figure 8-22. How does the MPU tell that switch 7 is closed?

28. If both sides of the PIA are used, how many switches can be handled using the matrix technique?

## ANSWERS

21. Assuming that the PIA was initially reset, a typical initialization routine would be:

```
LDAA    #FF
STAA    4004
STAA    4006
LDAA    #04
STAA    4005
STAA    4007
```

22. The B side.

23. Keep in mind that $Q_7$ must conduct and that pins b, c, g, and f must be high. Thus, the following instructions could be used:

```
LDAA    #80
STAA    4006
LDAA    #99
STAA    4004
```

24. inputs.

25. The MPU reads in data from both sides of the PIA and compares this data with several different bit patterns. If switch 9 is closed, the bit pattern from the B side will be $FD_{16}$.

26. The output register of the B side is configured in this manner during the initialization process by storing $OF_{16}$ in the B side data direction register.

27. The MPU periodically resets line PB7 to 0. If PB2 is subsequently found to be 0, the MPU knows that switch 7 is closed.

28. Up to $64_{10}$.

# INTERFACING EXPERIMENTS

The interfacing experiments are included in Unit 10 of this course. Go to Unit 10 and perform experiments 5 through 9. Some of these experiments are quite involved and you should not attempt more than one experiment per sitting.

# UNIT EXAMINATION

1. In some microcomputer systems, input data from a keyboard is read repeatedly before it is accepted.
   A. This speeds up the MPU.
   B. This prevents two keys from being read at once.
   C. This overcomes contact bounce.
   D. All the above.

2. Refer to the program shown in Figure 8-5. Assume that key 6 is depressed. What hex number will be in accumulators A and B after the COMB instruction at line 18 is executed?
   A. ACCB = $40_{16}$, ACCA = $00_{16}$.
   B. ACCB = $60_{16}$, ACCA = $00_{16}$.
   C. ACCB = $06_{16}$, ACCA = $00_{16}$.
   D. ACCB = $00_{16}$, ACCA = $06_{16}$.

3. The purpose of the PIA is to:
   A. Simplify the problem of interfacing the MPU to the ROM.
   B. Simplify the problem of interfacing the MPU to the RAM.
   C. Simplify the problem of interfacing the MPU to input/output devices.
   D. Act as a universal input/output device.

4. The advantage of the PIA over conventional combinational logic circuits is:
   A. Generally, fewer IC's are required.
   B. The PIA is more flexible since its characteristics can be changed by the program.
   C. In many cases, the PIA requires no separate address decoder.
   D. All the above.

5. The A side of the PIA is selected when:
   A. RS0 = 0.
   B. RS0 = 1.
   C. RS1 = 0.
   D. RS1 = 1.

6. To select output register A of the PIA:
   A. RS0 must be 0 and bit 2 of control register A must be 0.
   B. RS0 must be 1 and bit 2 of control register A must be 0.
   C. RS0 must be 0 and bit 2 of control register A must be 1.
   D. RS0 must be 1 and bit 2 of control register A must be 1.

7. When the PIA is reset:
   A. Both sides are configured as outputs.
   B. Both sides are configured as inputs.
   C. The A side is configured as outputs while the B side is configured as inputs.
   D. The B side is configured as outputs while the A side is configured as inputs.

8. Refer to Figure 8-19. Which of the following sequences will cause display 4 to display the number 1.

| A. | | B. | | C. | | D. | |
|---|---|---|---|---|---|---|---|
| LDAA | #01 | LDAA | #9F | LDAA | #60 | LDAA | #10 |
| STAA | 4004 | STAA | 4004 | STAA | 4004 | STAA | 4004 |
| LDAA | #04 | LDAA | #10 | LDAA | #10 | LDAA | #04 |
| STAA | 4006 | STAA | 4006 | STAA | 4006 | STAA | 4006 |

9. Refer to Figure 8-19. All displays can be blanked by storing:
   A. FF at address 4004.
   B. 00 at address 4006.
   C. FF at address 4004 and 00 at address 4006.
   D. Any of the above.

10. Refer to Figure 8-22. An indication that switch A is closed is:
   A. PB1 goes low when PB6 goes low.
   B. PB0 through PB4 are all high.
   C. PB1 goes low when PB6 goes high.
   D. PB0 goes low when PB6 goes low.