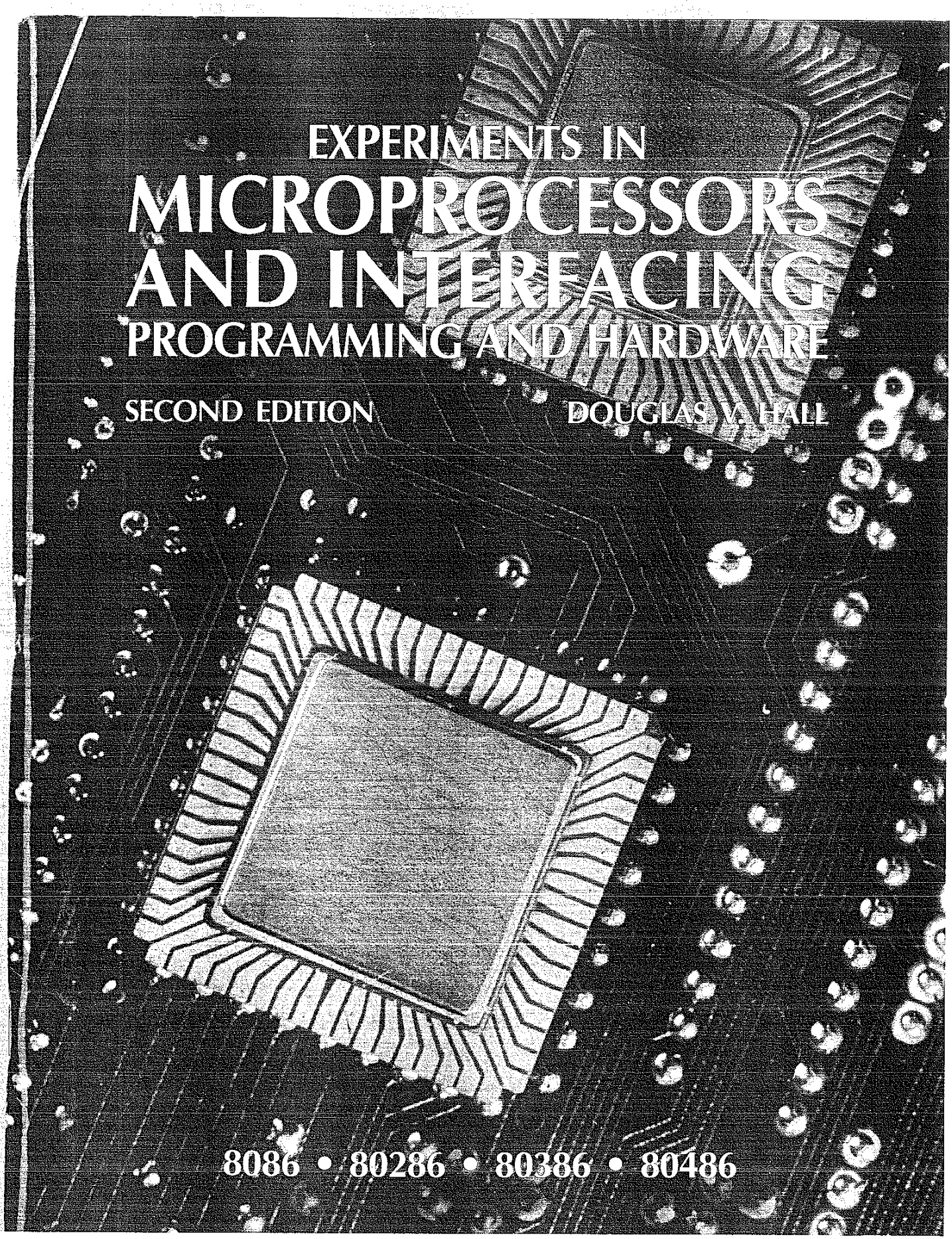

EXPERIMENTS IN
MICROPROCESSORS
AND INTERFACING
PROGRAMMING AND HARDWARE
SECOND EDITION
DOUGLAS V. HALL
8086 • 80286 • 80386 • 80486

EXPERIMENTS IN MICROPROCESSORS AND INTERFACING

PROGRAMMING AND HARDWARE

SECOND EDITION

DOUGLAS V. HALL

GLENCOE

Macmillan/McGraw-Hill

Lake Forest, Illinois Columbus, Ohio Mission Hills, California Peoria, Illinois

Send all inquiries to:
GLENCOE DIVISION
Macmillan/McGraw-Hill
936 Eastwind Drive
Westerville, OH 43081
ISBN 0-07-025743-4

Printed in the United States of America

1 2 3 4 5 6 7 8 9 SEM 99 98 97 96 95 94 93 92 91

CONTENTS

PREFACE

This book is intended to provide a broad spectrum of 8086/8088 microprocessor and microcomputer laboratory exercises. Included are exercises suitable for a wide variety of course levels, lab times, and available equipment. We did not write this book as a "fill-in-the-blank" type manual because that is not the way people develop projects in industry. For each experiment you should keep a log or diary of your thinking, algorithms, answers to any questions specifically asked in the experiment, and your final tested result. Industry guidelines indicate that about 30 to 35 percent of the development time for a project should be spent on writing documentation, so use this as a guideline in your work. This detailed documentation is critical for finding "bugs" in the project, adding improvements, and explaining the operation to a supervisor or user.

The main theme of this manual is interfacing a microcomputer to a wide variety of peripheral devices and systems. The first 29 experiments are built around the SDK-86 board available through the Academic Relations Department of Intel or directly from University Research and Development Associates (URDA) in Pittsburgh, Pennsylvania. Many of these experiments use interface modules which connect to the ports or buses of the SDK-86. The interface modules are for the most part designed with inexpensive, readily available parts. A list of the parts needed for each hardware interface module is given in the exercise where that module is first used. We recommend that they be built on protoboard using wire-wrap sockets and interfaced to port connectors on the SDK-86 with ribbon cables and the appropriate connectors. Additional circuitry, which can be wire wrapped on an SDK-86 board to give it programmable interrupt and counter capability, is given in Experiment 21 (Figure 21-1). Assembly language programs can be developed on a PC and downloaded to the SDK-86 board through an RS-232C cable. One big advantage of the SDK-86 boards is that everything is out in the open so you can easily observe signals with a scope or logic analyzer.

An alternative approach which works well for many of the experiments is to use a PC with a parallel-port expansion board. In this case the ports are connected to the various interface modules. If you decide to use this approach, make sure you get a parallel port board which has interrupt capability, because most real-world applications require at least one interrupt.

We start this book by showing you how to enter, run, and debug 8086 machine code programs on the SDK-86 using the onboard keypad. This gives you a feeling for working at the basic machine code level, but it is quite tedious for longer programs. Therefore, we next show you how to develop assembly language programs on a PC-type computer using an editor, an assembler, a linker, and a debugger. For our examples we use the Borland tools, but you can easily adapt the experiments to other tools such as those from Microsoft.

In this first group of experiments there are several which focus on looking at bus signals with a scope and/or a logic analyzer. Also included is an experiment on microcomputer troubleshooting.

The emphasis in Experiments 30–38 is on interfacing with system peripherals such as math coprocessors, CRT displays, and disk systems. We provide system-level experiments which allow you to work in C. For the C experiments we use the Borland C++ Professional Package because it works very well and it is available to educational institutions and students at a generous price. Microsoft Corp. also has an academic support program, and if you would rather work with their C programming tools, the C-based experiments here can easily be adapted to them.

Experiment 39 uses the SDK-386 board from URDA to introduce you to 80386 programming and low-level interfacing. Experiment 40 uses a 386-based PC to give you a chance to explore the advanced instructions of the 80386 and show off all you learned in the previous experiments.

We hope that the exercises shown will suggest further projects to you in areas such as electronic music synthesis, robotics, graphics, and interfacing of various devices to microcomputers. We welcome your comments concerning any problems you have with any of the exercises, suggestions you have for improving the exercises, or additional exercises you would like to see included. Address correspondence to Douglas V. Hall, 17197 S. Hattan Rd., Oregon City, OR 97045.

I wish to thank Jim Valluzzi of Portland Community College in Portland, Oregon, for carefully reviewing these experiments and contributing from his experience. I also wish to thank the many students who worked their way through these experiments and made numerous helpful suggestions.

Douglas V. Hall

EXPERIMENT 1

Introduction to the SDK-86 Microprocessor Development Board

REFERENCES

Hall: Chapter 2

Intel or URDA: SDK-86 User's Manual

EQUIPMENT AND MATERIALS

Intel or URDA SDK-86 microprocessor board

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Find the major functional blocks, such as ROM, RAM, CPU, ports, and keyboard/display interface, on the SDK-86 board.
2. Identify the RAM address range that you can load your programs into, and give the starting address of the ROM that contains the system monitor program.
3. Use the keypad monitor commands to examine and change bytes and words in memory.
4. Use the keypad monitor commands to enter and run a simple 8086 machine language program.

PROCEDURE

Overview of the SDK-86 Board

From your reading in the textbook, *Microprocessors and Interfacing: Programming and Hardware* (hereinafter referred to as Hall): Chapter 2, you should know that the main elements of a microcomputer are the CPU, ROM, RAM, and I/O ports. The IC or group of ICs which performs each of these functions is identified on the SDK-86 board by white lines and labels silk-screened directly on the printed-circuit board.

1. Carefully inspect the board to find each of the following parts. Then list the manufacturer's part numbers for the devices used as:

   a. CPU
   b. PROMs
   c. RAMs
   d. I/O ports
   e. Major device in the keypad/display interface section

2. The SDK-86 board has 2 Kbytes of RAM and the URDA board has 16 Kbytes of RAM starting at address 00000H. The monitor program, which executes when you turn on power to the board, uses addresses 00000–000FFH for its own purposes. The rest of the RAM is available for you to load your programs.

   a. What is the ending address (hexadecimal) for the RAM on your board?
   b. What is the first RAM hex address you can use?
   c. How many bytes of RAM are available for your programs?
   d. The ROMs containing the monitor program on the SDK-86 board occupy the address space FE000H–FFFFFH. How many bytes of ROM is this?

Using the SDK-86 Examine Memory, Examine Register, and GO Commands

Now that you have found your way around the board, connect a +5-V, 4–5-A power supply to the power input terminals on the board, and turn on the power.

Whenever power is applied to the SDK-86 board or the SYSTM RESET key on the keypad is pressed, the monitor program, contained in the on-board ROMs, executes automatically. Execution of this program allows you to examine the contents of registers and memory locations, enter your programs and data into RAM, run your programs, and so on, using the keys on the keypad.

A dash (-) and the message 86 1.1 should be displayed on the LEDs over the keyboard after the power is turned on or the SYSTM RESET key is pressed. The four LEDs where the -86 is displayed are referred to as the *address field displays*, because they are used to display address information, register names, and command prompts. The four LED digits where the 1.1 is displayed are referred to as the *data field displays*, because they are used to display data present in an addressed register or memory location.

1. To see the symbol displayed by the SDK-86 for hex characters 0–F, press the examine memory byte key,

   EB

   and then slowly press the following sequence of keys:

   0 1 2 3 4 5 6 7 8 9 A B 6 C D E F

   You should see the symbol for each character appear in the rightmost digit of the address field as you press the key and then move left one digit position as you press the next key. Note that the symbol for a six has a lit segment at the top, but the symbol for a B (b) does not. Watch this carefully as you enter programs and data. Also note that the symbols for the last four keys pressed remain on the displays.

2. To prepare the SDK to receive its next command, press the key labeled

   SYSTM RESET

   Whenever a dash is displayed on the leftmost LED, the SDK is ready for you to enter a command with one of the keys on the keypad. Figure 1-1 shows the functions of each of these keys. Note that many of the keys have several functions, depending upon whether you are entering a command, entering addresses and data, or examining 8086 registers. The 0 key, for example, is used to enter the EB command if a dash is showing. It is used to enter the number 0 if an address or data word is being entered. It is used to represent the AX register if registers are being examined.

Function Key	Operation
SYSTM RESET	The SYSTM RESET key allows you to terminate any present activity and to return your SDK-86 to an initialized state. When pressed, the 8086 sign-on message appears in the display and the monitor is ready for command entry.
INTR	The INTR (interrupt) key is used to generate an immediate, non-maskable type 2 interrupt (NMI). The NMI interrupt vector is initialized on power up or system reset to point to a routine within the monitor which causes all of the 8086's registers to be saved. Control is returned to the monitor for subsequent command entry.
+	The + (plus) key allows you to add two hexadecimal values. This function simplifies relative addressing by allowing you to readily calculate an address location relative to a base address.
−	The −(minus) key allows you to subtract one hexadecimal value from another.
:	The : (colon) key is used to separate an address to be entered into two parts; a segment value and an offset value.
REG	The REG (register) key allows you to use the contents of any of the 8086's registers as an address or data entry.
,	The , (comma) key is used to separate keypad entries and to increment the address field to the next consecutive memory location.
.	The . (period) key is the command terminator. When pressed, the current command is executed. Note that when using the Go command, the 8086 begins program execution at the address specified when the key is pressed.

(a)

Hexadecimal Key	Command		Register	
	Acronym	Name	Acronym	Name
0 EB/AX	EB	Examine Byte	AX	Accumulator
1 ER/BX	ER	Examine Register	BX	Base
2 GO/CX	GO	Go	CX	Count
3 ST/DX	ST	(Single) Step	DX	Data
4 IB/SP	IB	Input Byte	SP	Stack Pointer
5 OB/BP	OB	Output Byte	BP	Base Pointer
6 MV/SI	MV	Move	SI	Source Index
7 EW/DI	EW	Examine Word	DI	Destination Index
8 IW/CS	IW	Input Word	CS	Code Segment
9 OW/DS	OW	Output Word	DS	Data Segment
A /SS	none	N/A	SS	Stack Segment
B /ES	none	N/A	ES	Extra Segment
C /IP	none	N/A	IP	Instruction Pointer
D /FL	none	N/A	FL	Flag
E	none	N/A	none	N/A
F	none	N/A	none	N/A

(b)

FIGURE 1-1 SDK-86 board *(a)* Function key operation. *(b)* Hexadecimal keypad interpretation. *(Intel Corporation)*

The SDK-86 keypad monitor program allows you to execute 10 basic commands. Figure 1-2 summarizes the format for these commands. Work your way through the following sections, which show you how to use some of these commands. The sections are identified by the command name so that you can find them easily if you need to refresh your memory at some future time.

EXAMINE BYTE (EB)

The examine byte command is used to inspect and, if desired, change the byte stored in a memory location. This command allows you to easily enter program code bytes or data bytes in successive memory locations. As shown in Figure 1-2, the format for the command is

EB addr , [[data] ,] * .

The letters "EB" tell you to press the key labeled EB. The "addr" tells you to enter the address of the byte that you want to look at. Pressing the comma after entering the address causes the address and the contents of that address to be displayed. The square brackets and the * shown in the command format are not entered. Command parts contained in square brackets are optional. The inner square brackets in this command indicate that the data at that address can be changed by simply typing in the desired hex value. The "*" indicates that the part of the command contained in the outer square brackets can be repeated. In other words, by pressing the <,> key, you can step to the next address, see its contents, and change them if you want.

To see the byte in a memory location, press the key labeled EB, enter the address of the desired byte, and then press the <,> key. Addresses are entered in the segment:offset form, and so in order to see the contents of the memory location 000100H (0000:0100), you enter 0:100 as the address. Note that the SDK does not require you to enter leading zeros and does not display them.

1. To see the byte in memory at 0000:0100H, enter the following series of keys:

   EB 0 : 1 0 0 ,

   After you press the <,> key, the contents of address 0000:0100 should appear on the rightmost two digits of the data field LEDs.
2. To insert some new byte value at that address, enter the hex digits of the new value with the keypad. To enter the value C4H, press the keys

   C 4

   As you press the keys, you should see the new value appear on the data field LEDs. If you make a mistake while entering a value, just type in the two correct digits; they will replace the previously entered value.
3. To examine the next memory location, press <,>. You should now see the address 101 displayed on the address field digits, and the byte stored at that location displayed on the right two data field LEDs.
4. To change the contents of this new address, enter another value with the keypad. Press the <,> key again if you want to examine and/or change the next sequential memory address.
5. To terminate the command after you have looked at all the memory locations you want, press the <.> key. When the dash prompt (–) appears, you can enter your next command.

Command	Function/Syntax
Examine Byte	Displays/modifies memory byte locations EB <addr> ⊡ [[<data>] ⊡] *⊡
Examine Word	Displays/modifies memory word locations EW <addr> ⊡ [[<data>] ⊡] *⊡
Examine Register	Displays/modifies 8086 register contents ER <reg key> [[<data>] ⊡] *[⊡]
Input Byte	Displays data byte at input port IB <port addr> ⊡ [⊡] *⊡
Input Word	Displays data word at input port IW < port addr > ⊡ [⊡] * ⊡
Output Byte	Outputs data byte to output port OB <port addr> ⊡ <data> [⊡ <data>] *⊡
Output Word	Outputs data word to output port OW <port addr> ⊡ <data> [⊡ <data>] *⊡
Go	Transfers control from monitor to user program GO [<addr>] [⊡ <breakpoint addr>] ⊡
Move	Moves block of data within memory MV <start addr> ⊡ <end addr> ⊡ <destination addr> ⊡
Step	Executes single user program instruction ST [<start addr>] ⊡ [[<start addr>] ⊡] *⊡

FIGURE 1-2 SDK-86 board monitor command summary. (*Intel Corporation*)

EXAMINE WORD (EW)

The examine word command is used to inspect and, if desired, change a word stored in two memory locations. As shown in Figure 1-2, the format for the command is

EW addr . [[data] ,] * .

The operation of EW is identical to that of the EB command described in the preceding paragraphs, except that a data word (16 bits) will be displayed on the four data field LEDs. The right two digits of the displayed word represent the byte contained in memory at the address specified in the command. The high byte of the word, displayed on the upper two data field LEDs, represents the byte contained in the memory location after the address specified in the command.

1. Examine the word contained in addresses 0000:0100 and 0000:0101 by pressing the keys

   EW 0 : 1 0 0 ,

   You should see the byte from 0000:0100H displayed on the right two data field LEDs and the

byte from address 0000:0101 displayed on the left two LEDs.

2. Change the displayed word by entering the desired word. Press the <,> key to step to the next word in memory.
3. Write down the value you see for this word. Then press the <.> key to get the dash prompt back, so that you can enter another command.
4. Use the EB command to verify that the low byte of the displayed word came from the low address (0000:0100) and the high byte of the word came from the higher address (0000:0101). This relationship is very important to remember.

EXAMINE REGISTER (ER)

To display the contents of an 8086 register, press the <ER> key and then the key that contains the letter initials for that register.

1. Examine the contents of the AX register by pressing the keys

   ER AX

   Notice that the <AX> key is also the <0> key. The address field of the display will show the letter A to indicate the register that is being displayed, and the data field will show the word contained in the AX register.
2. Examine the contents of the next 8086 register by pressing the <,> key. Each time you press the <,> key, the SDK will display the next register in the sequence

   AX, BX, CX, DX, SP, BP, SI,
   DI, CS, DS, SS, ES, IP, FL

   FL is the flag register. In each case the address field will show one or two mnemonic letters to indicate which register is being displayed.
3. To change the word in any register, enter the four hex digits for the word you want in the register. Then press the <,> key to step to the next register, or press the <.> key to get the dash prompt back so that you can enter your next command.

NOTE: The ER command operation is not circular, so if you want to look at a register earlier in the sequence, press the <.> key to terminate the command. Then press the <ER> key to start the command again and then press the key that has the name of the register you want to see.

GO

The SDK-86 GO command allows you to run programs which you have previously loaded into RAM. Remember, you can load programs into RAM starting at address 0000:0100H. The format for the command is

GO [addr] [, breakpoint addr]

The first address specified in the command is the program address at which you want execution to start. The breakpoint address specified in the command is the program address at which you want execution to stop. Whenever you run a program on a microcomputer, you must in some way prevent the processor from fetching and attempting to execute random bytes stored in memory after your program. One way to solve this problem is to put a jump instruction as the last instruction in the program to make the program execute over and over again in an endless loop.

For programs that you don't want to execute over and over, you can put a breakpoint after the last program instruction. When the 8086 reaches the breakpoint, it will save the values that were in the 8086 registers after the last program instruction was executed and return control to the SDK-86 monitor program. You can then use the EB or EW command to look at any program results in memory. You can use the ER command to look at any program results in registers.

Using the SDK-86 Commands to Load and Execute a Program

Figure 1-3 shows a simple program that you can enter into the RAM on your SDK and run for practice. When run, this program causes the SDK to display a 4-digit hexadecimal count sequence on the data field LEDs. The program keeps counting from 0000H to FFFFH. Don't worry if you haven't the foggiest notion how the program works at this point. You can learn to drive a car without knowing how the camshaft opens the valves.

To load and run the program in Figure 1-3, follow these instructions.

1. Look at the first column from the left in Figure 1-3 to find the addresses for the code you will be loading into RAM. The addresses shown in the program listing are *relative*. This means that they show where the bytes for each instruction are relative to the starting address at which you load the program into RAM. You want to load the program starting at address 0000:0100H, and so you add 100H to each of the addresses in Figure 1-3 to find the actual address of each byte. The EB command, described previously, is used to enter the sequence of data and instruction bytes shown in Figure 1-3 into the SDK-86 RAM starting at address 0000:0100H. To see the current contents of address 0000:0100 on the data field LEDs, press the keys

   EB 0 : 1 0 0 ,

   You want to enter the first instruction code byte of the program into this location.
2. The second column to the left in the program listing contains the codes that you will be loading into RAM. The reason there are different numbers of code bytes on different lines is that the code byte(s) required for an instruction are put on the line with the instruction. Look at the

```
IBM Personal Computer MACRO Assembler  Version 2.00  Page 1-1
                                page, 132
                                ;8086 Count Program
                                ;
                                ;       This program displays an incrementing hexadecimal count on
                                ;       the data field display of the SDK-86 board.
                                ;
0000                            CODE_HERE       SEGMENT  PUBLIC
                                        ASSUME CS:CODE_HERE, DS:DATA_HERE
                                ;initialize data segment
0000  B8 1300                           MOV  AX, DATA_HERE
0003  8E D8                             MOV  DS, AX
0005  BF 0000                           MOV  DI, 0000              ; zero counter
0008  BB 0E00                           MOV  BX, OFFSET SEVEN_SEG ; pointer to seven-segment codes
000B  BA EAFF                   DISPLAY:MOV  DX, OFFEAH            ; point at 8279 control address
000E  B0 90                             MOV  AL, 90H               ; load control word for data field
0010  EE                                OUT  DX, AL                ; send control word to 8279
0011  BA E8FF                           MOV  DX, OFFE8H            ; point at 8279 display RAM
0014  8B CF                             MOV  CX, DI
0016  8A C1                             MOV  AL, CL                ; get low byte to be displayed
0018  24 0F                             AND  AL, 0FH               ; mask upper nibble
001A  D7                                XLATB                      ; translate lower nibble to 7-seg code
001B  EE                                OUT  DX, AL                ; send to 8279 display RAM
001C  8A C1                             MOV  AL, CL                ; get low byte again
001E  B1 04                             MOV  CL, 04                ; load rotate count
0020  D2 C0                             ROL  AL, CL                ; Move upper nibble into low position
0022  24 0F                             AND  AL, 0FH               ; Mask upper nibble
0024  D7                                XLATB                      ; translate 2nd nibble to 7-seg code
0025  EE                                OUT  DX, AL                ; send to 8279 display RAM
0026  8A C5                             MOV  AL, CH                ; Get high byte to translate
0028  24 0F                             AND  AL, 0FH               ; Mask upper nibble
002A  D7                                XLATB                      ; Translate to 7-seg code
002B  EE                                OUT  DX, AL                ; send to 8279 display RAM
002C  8A C5                             MOV  AL, CH                ; get high byte to fix upper nibble
002E  D2 C0                             ROL  AL, CL                ; move upper nibble into low position
0030  24 0F                             AND  AL, 0FH               ; mask upper nibble
0032  D7                                XLATB                      ; translate to 7-seg code
0033  EE                                OUT  DX, AL                ; 7-seg code to 8279 display RAM
0034  BE FFFF                           MOV  SI, OFFFFH            ; load counter to pause
0037  4E                        PAUSE:  DEC  SI
0038  75 FD                             JNZ  PAUSE
003A  47                                INC  DI                    ; increment counter
003B  EB CE                             JMP  DISPLAY
003D                            CODE_HERE ENDS

0000                            DATA_HERE SEGMENT       WORD    PUBLIC
0000  3F 06 5B 4F 66 6D 7D 07           SEVEN_SEG       DB      3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H
                                                        ;        0   1   2   3   4   5   6   7
0008  7F 6F 77 7C 39 5E 79 71                           DB      7FH, 6FH, 77H, 7CH, 39H, 5EH, 79H, 71H
                                                        ;        8   9   A   b   C   d   E   F
0010                            DATA_HERE ENDS
                                        END
```

FIGURE 1-3 8086 counting program.

top of that column to find the B8H code byte and the other codes that will be put after it. To put the B8 code byte into RAM and to see address 0101 on the address field and the contents of that address on the data field, press the keys

B 8 ,

3. The next code byte (to the right of B8H) is 13H. To enter this with the keypad, press the keys

1 3 ,

The <,> key steps to the next RAM address.

4. The next code byte for the program is 00H, so enter this value and press the <,> key to step to the next address.
5. The next code byte, found on the next line in the program printout, is 8EH. Enter this value and press the <,> key to step to the next RAM address.
6. Enter each code byte in column 2, followed by a <,>.
7. When you reach the end of the code bytes in column 2, insert the data bytes, starting with 3FH in the series of addresses starting with 0000:013E. Note that address 0000:013D is skipped.
8. After you have entered all the program bytes, press the <.> key to return control to the monitor program.
9. The next step is to check if the program was entered correctly. Press the keys

EB 0:100,

to see the byte in the first address. If this byte is not correct, enter the correct value and press the <,> key to get to the next address. If the byte is correct, just press the <,> key to step to the next address. A common error is to miss a byte; consequently, the bytes after the missed byte end up at wrong addresses. If you find an error, you can just insert the correct value(s), and proceed.

10. When the program seems correct in RAM, run the program from the starting address 0000:0100H. Press the keys

GO 0 : 1 0 0 .

11. If the program does not operate correctly, press the RESET key and check the codes in RAM again. If the program still does not work correctly, get help.

If time permits we suggest that you now go on and do Experiment 2, which shows you how to use breakpoints to debug a program that doesn't work and how to modify the operation of the program. Experiment 2 uses the same program as this experiment, so if you leave the power on and do Experiment 2 now, you won't have to key the program in again.

EXPERIMENT 2

Debugging Programs with the SDK-86 Keypad Monitor

REFERENCES

Hall: Chapter 3

URDA or Intel: SDK-86 User's Manual

EQUIPMENT AND MATERIALS

SDK-86 microprocessor board

INTRODUCTION

Hall: Chapter 3 discusses the importance of writing microcomputer programs in a structured manner so that they are more likely to work correctly. Sometimes, however, in spite of careful planning, writing, and entering, an error creeps into a program. Most systems have programs or a series of commands that help you to find the problem(s) in a program. Finding and correcting the problems in a program is commonly referred to as *debugging* the program. The two debugging techniques we want to teach in this exercise are *breakpoints* and *single stepping*. It is important to use these techniques as part of a systematic overall debugging procedure rather than randomly experimenting. Here is a debugging sequence that we have found effective.

1. Check your algorithm to make sure it really specifies what you want it to do.
2. Check that the instructions you have chosen do actually implement the steps of the algorithm.
3. Check that the instructions are correctly coded, if you are hand coding, and that they are correctly entered into RAM.
4. If these checks don't turn up any problems, choose a logical checkpoint in the program and predict what the contents of key registers and memory locations should be when the program has executed to that point. Record your predictions.
5. In some way set a breakpoint at that point in the program and run the program. The program will run to the breakpoint and stop.
6. Use the examine memory and examine register commands to see if memory and register contents agree with your predictions. The prediction part of this is very important, because it doesn't do you any good to look at memory and register contents if you don't know what the contents are supposed to be.
7. If the results are correct at that point, you can predict the memory and register contents for a later point in the program and move the breakpoint to that point. You can then run the program to the new breakpoint and recheck key memory locations and registers to see if they are correct at that point. The breakpoint technique allows you to localize the problem area in a program.
8. Another debugging technique available in some systems is the single step. This function executes one instruction and stops. You can then examine memory and register contents to see if they agree with what you predicted they should be at that point. If the results are correct, you can use the single-step function to execute the next instruction and recheck key memory and register contents. This function allows you to step your way through a program one instruction at a time, checking the contents of key memory and register locations after each instruction. After you have debugged a few programs, you will get a feeling for which technique is best suited to debugging a particular program.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Use the SDK-86's keypad monitor commands to run a program to a breakpoint and then examine register and memory contents to see if they agree with predictions.
2. Use the SDK-86 keypad single-step command to step through a program.
3. Modify the operation of a program by changing a value in the program.

PROCEDURE

The Example Program

1. If it is not already loaded, use the SDK-86 keypad monitor, EB command to load the program shown in Figure 1-3 into RAM starting at address 0000:0100H, as described in Experiment 1.
2. Check that all the instruction code bytes and data bytes are inserted correctly.
3. Use the SDK-86 GO command with no specified breakpoint to run the program as you did in the last section of Experiment 1. If the program does not run correctly, press RESET to get back to the monitor program so that you can enter commands again. Use the EB command to make sure that the codes are all in memory correctly. Remember that the top segment on the LEDs is lit for a 6 but is not lit for a b. When you get the program working correctly, go on to the next step here.

Using a Breakpoint

As a first example of the use of a breakpoint, you might want to see if the first six instructions of the program are initializing the specified registers in the way you want.

You insert a breakpoint as part of the GO command. As shown in Figure 1-2, the format of the GO command is

GO [start addr] [, breakpoint addr] .

Remember, the brackets shown in the command format are not entered; they simply indicate the parts of the command that are optional. For our case here, the starting address is 0000:0100 (CS = 0000, IP = 0100; you could also use 0010:0000—that is, CS = 0010 and IP = 0000). The instruction at a specified breakpoint address does not get executed. Therefore, the breakpoint address you use is the address of the instruction after the last instruction you want executed. (A breakpoint must always be put at the address of the first byte of an instruction!)

1. Predict what should be in the affected registers after these instructions execute. Because you are new at this business, here's what should be in them.

Register	Predicted Contents	Observed Contents
AL	90H	_____
BX	013CH	_____
DX	FFEAH	_____
DI	0000H	_____
DS	0000H	_____

2. For the example here, you want to execute the first six instructions, so your breakpoint address will be 0000:0110, the address of the seventh instruction in the program. Enter the command

GO 0 : 1 0 0 , 0 : 1 1 0 .

When execution reaches the breakpoint address you have specified, the SDK will display a dash in the address field and the letters "br" in the data field. If the last instruction in your program is INT 3 (code CCH), the "br" message will not appear. The dash indicates that the SDK is ready for you to enter your next command.

3. Use the ER command to determine the 8086 register contents. Record these contents so that you can compare them with those predicted.

NOTE: If the SDK "locks up" and does not reach the breakpoint, you can get back to the monitor by pressing the RESET key. In this case you may have to reenter the instruction byte at the breakpoint address, because the monitor does the breakpoint operation by temporarily changing this instruction to CCH.

Using the Single-Step Command

Another technique that is sometimes useful for debugging a program is single stepping. When the single-step command on the SDK-86 is used, the monitor program will execute one program instruction and then stop. You can then use the EB, EW, and ER commands to see if the program data is correct. If the data is correct, you can then execute the next instruction and check again. If the data is not correct, you can try to determine why it isn't.

As shown in Figure 1-2, the format of the SDK single-step command is

ST start addr , [[start addr] ,] * .

What all of this means is that if you press the keypad key labeled ST (the <3> key) followed by the address of the instruction from which you want to start stepping and then press the <,> key, the SDK will execute the instruction at that address. The address of the next sequential instruction will be displayed on the address field of the display. Each time you press the <,> key, the SDK will execute one instruction and step to the next. Pressing the <.> key terminates the single-step command and returns execution to the monitor program.

1. Execute the first instruction in our example program by pressing the keys

ST 0 : 1 0 0 ,

2. The offset of the next instruction will be displayed on the address field LEDs, and the first byte of the instruction at that address will be displayed on the data field LEDs.

   Use the program listing in Figure 1-3 to see if it does this correctly for your program.
3. Now press the <,> key until you have stepped through the first six instructions, then press the <.> key to return to the monitor.
4. Use the ER command of the SDK to see if the register contents agree with those predicted.
5. There is a problem if you try to use the single-step

command to step all the way through the program to the INC DI instruction at offset 003AH. Can you see what the problem is? If you can't, try stepping through the program to the INC DI instruction to determine the problem. In the next section we show you one way around this problem.

Modifying the Operation of a Program

Take a look at the instructions starting at offset 0034H in the program. The first instruction here, MOV SI,0FFFFH, loads the immediate number FFFFH into the SI register. The next instruction, PAUSE:DEC SI, decrements the number in the SI register by 1. If the result of this decrement operation does not leave 0 in the SI register, the JNZ PAUSE instruction will send execution back to the PAUSE:DEC SI instruction. This type of operation is called a *program loop* and will repeat until the number in SI is counted down to 0.

1. Use this fact to help you predict the number that should be in SI when execution reaches the INC DI instruction at offset 003AH.
2. Enter a GO command, which will run the program to a breakpoint at the INC DI instruction's address.
3. Use the SDK-86 ER command to check your prediction.
4. Changing the MOV SI,0FFFFH instruction so that it loads a value other than FFFFH into the SI register has an interesting effect on the operation of the program. What SDK-86 command can you use to change the word that the instruction loads?
5. At what address in memory is the FFFFH part of the MOV SI, 0FFFFH instruction located?
6. Use the proper SDK-86 command to change the FFFFH in the MOV SI,0FFFFH instruction to 40FFH.
7. Use the GO command to run the program with no breakpoints in order to see the effect this change has on the operation of the program.
8. In your report describe the effect produced by the new value you put in the MOV SI instruction.
9. Experiment with different count values and observe the effects.

EXPERIMENT 3

A Simple 8086 Arithmetic Program

REFERENCES

Hall: Chapters 2 and 3

Hall: Appendices A and B

EQUIPMENT AND MATERIALS

SDK-86 microprocessor board

INTRODUCTION

For your first 8086 programming assignment, we have chosen a simple arithmetic example. This example was chosen because it can be done with just a few straightforward instructions and because it doesn't require interaction with any external hardware. In this experiment and the next you will hand code the instructions so that you develop a feel for the bit-level structure of microprocessor instructions. Except for very short programs such as these, hand coding is very tedious and error-prone, so in Experiment 5 and following experiments we show you how to use computer-based tools to develop assembly language programs.

Even though this program may seem very simple, it is important that you follow the program development steps described in Hall: Chapter 3, so that these steps are automatic to you when you write more complex programs. To refresh your memory, we will summarize the steps here.

1. Define very carefully what you want the program to do.
2. Write an algorithm for the program.
3. Choose the instructions which will implement each part of the algorithm.
4. In some way produce the machine codes for the program.
5. Load the program into a microcomputer and run it.
6. Debug the program as described in Experiment 2.

The more time you spend on the first three steps, the less time you will spend debugging the program and the more likely it is that the program will run correctly the first time.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write the algorithm for a simple arithmetic program.
2. Choose 8086 assembly language instructions that will implement this algorithm.
3. Use 8086 instruction coding templates to produce the machine codes for these instructions.
4. Enter, run, and debug the program on an SDK-86 microcomputer development board.

PROCEDURE

Adding 2 Bytes

The programming problem in this exercise is to develop a program which adds a byte in the AH register to a byte in the AL register and copies the sum to the BL register. The contents of the carry flag after the addition should be put in the least significant bit of the BH register. The other 7 bits of the BH register should all be made zeros (masked).

NOTE: Assume that you are going to load the desired numbers in the specified registers with the SDK-86 ER command before you run the program.

1. Write the algorithm (sequence of steps) you would use to do this addition. Remember, at this point just think of the operations you want to do in each step, not the instructions you are going to use to implement the algorithm. Mentally step through the algorithm to make sure you haven't left out any required parts.
2. Use the list of 8086 instructions in Hall: Chapter 3 to help you choose the appropriate instruction(s) for each step of your algorithm. To help you with this, think about the following questions.

   a. For the basic addition operation, do you want

to use the simple ADD instruction or the ADC instruction, which adds any carry from a previous operation to the sum?

b. What instruction can you use to transfer the sum to BL?

c. What instruction can you use to rotate the carry flag contents into the least significant bit of the BH register?

d. What logical instruction can be used to mask the upper 7 bits of the BH register without changing the LSB?

Put in the INT 3 instruction as the last instruction of your program. Remember from a discussion in Experiment 1 that whenever you run a program, you must in some way make sure that execution stops at the end of your program and that the processor doesn't go on executing the random bytes that happen to be present in the memory locations after your program. You can do this by inserting a breakpoint as part of the GO command, but in case you forget the breakpoint there, the INT 3 instruction will automatically cause execution to return to the monitor program.

3. When you have written out the list of assembly language instructions you have chosen for this program, refer to Hall: Figure 3-4 to see the overall format you will be using for the program. Don't write the list of instructions directly on the coding form because you won't know how many bytes each will require until you code out the instructions.

4. Write the mnemonic, operands, and comment for the first instruction on the first line of your coding form. You will be loading the program into memory starting at address 0000:0100H, so put 0100H in the address column on the paper as the offset of the first instruction.

5. Use the coding examples in Hall: Chapter 3 and the coding templates at the end of Hall: Appendix A or Appendix B to help you determine the binary code for this first instruction. After you have checked it carefully, write the hexadecimal equivalents for the instruction codes in the DATA/CODE column of the program paper.

6. Sequentially write the address offsets, hexadecimal instruction codes, assembly language mnemonics, operands, and comments for each of the other instructions on the program coding paper.

7. Use the EB command to enter the program into RAM starting at address 0000:0100.

8. Use the EB command again to check that you have correctly entered the instruction codes.

9. Use the ER command to load 42H into AH and 35H into AL.

10. For future reference, predict the result that these numbers should produce in the BH and BL registers when your program executes.

11. Use the GO command to execute the program.

12. When "br" and/or an "–" shows on the data field LEDs, indicating that the program has run to a breakpoint, use the examine register command to see if the AH, AL, BH, and BL registers contain the predicted numbers. If they do, rejoice, and go on to step 14.

13. If the registers did not contain the predicted numbers, proceed as follows.

a. Predict what the contents of the AH and AL registers should be after the first instruction executes.

b. Use the single-step command to execute the first instruction.

c. Use the ER command to see if the actual contents of the BH and BL registers agree with your predictions.

d. If the contents agree with your predictions, predict the contents that the registers should have after the next instruction executes. Then, execute the instruction and check the actual contents. Repeat the process until you find the instruction that is not producing the result you want.

e. When you find an instruction that is not producing the results you expect, read the detailed description of the instruction in Hall: Chapter 6 to make sure you are using the correct instruction. If you are, check the instruction coding to make sure you have coded it correctly. It is usually not too difficult to find the source of a problem when you narrow it down to one or two instructions with the single-step command or a breakpoint.

14. When your program works correctly, insert 9AH in AH and 78H in AL. Predict the results these numbers should produce in BH and BL. Then run the program again and see if the results agree with your predictions. Did the carry bit get moved correctly into the least significant bit of BH? If it did, you are done with this experiment. If not, debug the program as described in step 13.

EXPERIMENT 4

Writing a Program Which Accesses Data in Memory

REFERENCES

Hall: Chapters 2 and 3

Hall: Appendix

EQUIPMENT AND MATERIALS

SDK-86 microprocessor board

INTRODUCTION

For your second programming exercise, we have again chosen an arithmetic example in order to keep the programming task simple and to make external hardware unnecessary. Experiment 3 worked with data operands contained exclusively in registers. This lab works with data operands contained in memory.

OBJECTIVES

At the conclusion of this experiment, you should be able to write an assembly language program to

1. Add a BCD byte from one memory location to a BCD byte from the next higher memory location.
2. Adjust the result to a legal BCD number.
3. Store the BCD sum in the next higher memory address and any carry from the addition in the following memory location.

PROCEDURE

Accessing Data in Memory

1. Draw a memory diagram showing the relative locations of the two BCD numbers to be added, the sum, and the carry in successive memory addresses.
2. Write the algorithm for the program to add the two BCD numbers and save the sum and carry. Remember that after a BCD addition, the sum must be adjusted to make sure that the result is a legal BCD number.
3. Choose and write the series of assembly language instructions that will implement your algorithm. To help you choose the correct instructions, answer the following questions.
   a. What instructions are required to initialize the DS register so that program instructions can access the data locations in memory?
   b. Is the ADD or the ADC instruction more reasonable to use for the addition in this program?
   c. What instruction must be used to adjust the sum to the correct BCD form?
   d. What instructions can be used to copy the carry flag into the least significant bit of a byte and leave zeros in the upper 7 bits of the byte? Use a different method from the one you used in Experiment 3.
   e. What instruction should you put at the end of your program so that the 8086 will not accidentally execute the random contents in RAM after your program?
4. At the top of your coding form, set aside 4-byte-sized memory locations for the two numbers to be added, the sum, and the carry. For an SDK-86 board, these spaces can start at offset 0100H, so write "offset 0100H" next to the first data byte. For the first run of your program, try adding the BCD numbers 46H and 38H. Enter these numbers on the coding form and put zeros in the sum and carry locations, so that you can tell if the program puts anything in these locations when it runs. The instruction codes for your program can be put in memory after the data bytes. At what offset will you put the first instruction byte for your program?
5. Write your first assembly language instruction on the coding form after the data bytes. This first instruction will probably be MOV AX, (the segment base for the data segment containing numbers to be added). The segment base address is simply a number, so this instruction can be coded simply as a MOV immediate to register instruction. Find the coding template for this instruction at the end of Hall: Appendix A or Appendix B.

   According to the template in Appendix B, the basic opcode for the MOV immediate word to

register instruction is B8H + the 3-bit code for the destination register. The 3-bit code for the AX register is 000, so the basic opcode for the instruction is simply B8H. The 16-bit number to be loaded into the AX register is coded in as bytes 2 and 3 of the instruction. The least significant byte of the word to be loaded is put in the first byte after the opcode, and the high byte of the word is put in the next byte. **An important point to remember is that the low byte goes in the lower address.**

To determine the actual 16-bit number you want to code into this instruction, ask yourself the question, What is the segment base address for the data segment I set up at the top of the coding form? Write this number in as bytes 2 and 3 after the basic opcode for the instruction on the coding form.

6. Write the remaining instructions for your program one at a time, and work out the codes for each. Use the examples in Hall: Chapter 3 to help you work out the codes for the instructions that access locations in the data segment. Remember, the direct address or displacement required in these instructions is coded in as bytes 3 and 4 of the instructions, low byte first.
7. When you have coded all your program, recheck the program to make sure that you have used the correct instructions and that you haven't left any displacements or offsets out of the instruction codes. Then enter the data and code bytes in the RAM of the SDK-86 board.
8. The correct BCD result for adding 46H and 38H is 84H with no carry. Run your program and then check the sum memory location to see if your program produced this result. Also check if the carry byte is zero, as it should be. If the results in memory are not correct, use a breakpoint or single stepping to debug the program.
9. When you get the basic program operating correctly, load the data memory locations with the BCD values 89H and 93H. Predict the BCD values that these numbers should produce, then run the program and see if your program produced the predicted sum and carry. If it did not, find the problem.
10. Modify your program so that it subtracts the second BCD number from the first BCD number and puts the correct BCD difference in the next memory location. Any borrow produced by the subtraction should be put in the next memory location after the difference. Again, to help you choose the correct instruction, answer the following questions.
    a. What instruction should you use to do the subtraction?
    b. What instruction is required to adjust the result of the subtraction to BCD form?
11. Load the program and some test values in memory, run the program, and debug it, if necessary.

By now you have probably decided that hand coding and keying in 8086 programs is a painful task. In the next experiment we start showing you how to use a computer to do these tasks so that you can develop some more interesting programs.

EXPERIMENT 5

Using a Computer to Develop Assembly Language Programs

REFERENCES

Hall: Chapter 3

IBM or Microsoft DOS manual

Manual for the text editor you are using

Borland Turbo assembler (TASM) manual or Microsoft MASM manual

EQUIPMENT AND MATERIALS

IBM PC or PS/2-compatible computer

Printer

Formatted disk to save work files

INTRODUCTION

As described in Hall: Chapter 3, assembly language programs can be developed much more easily with a computer than with the hand-coding and keying method used in Experiments 3 and 4. To develop assembly language programs on a computer, several tools (programs) are used.

The first programming tool you use is an editor program. An editor program, when run, allows you to type in the mnemonics, operands, and comments for your program. Characters will be displayed on the screen as you type them in. Since the characters you type in are initially just stored in RAM in the computer, you can easily go back and correct any mistakes you may have made. When you get your program entered into RAM in the form you want, you save it in a named file on disk. This initial version of your program is called the *source file*.

There are a great many available editors that can be used to develop the source files for assembly language programs. One of these is EDLIN, the line-oriented editor that comes with DOS. This editor is somewhat inconvenient to use for all but the very shortest programs, so we recommend a screen-oriented editor. Examples of this type of editor are the "Shareware" editor PCWRITE, Wordstar™, and the editor that is part of the Borland C++ Integrated Development Environment, which we use in later exercises. The main requirement is that the editor be able to generate pure ASCII text files.

The editor we have chosen to use here is the Borland Turbo C++ Integrated Development Environment editor. If you are using some other editor, you can just replace the editor section of this exercise with the appropriate sections from the manual for your editor.

The next program you use in developing assembly language programs is an *assembler*. An assembler translates the assembly language instructions in your source program to the equivalent binary codes, which can be executed. Two commonly available 8086 assemblers for the IBM PC are Borland's Turbo Assembler, TASM, and Microsoft's Macro Assembler, MASM. We use TASM for the discussions here. MASM source file requirements are nearly the same as those for TASM, so if you have MASM, you can easily adapt our discussion. We will point out any differences as we go along.

After you get your program to assemble correctly, you LINK the .OBJ file produced by the assembler to produce an executable (.EXE) file. In later exercises we show you how to load and run the .EXE file. To link a program you will use the LINK program that came with the assembler you are using.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Use the simple DOS commands DIR, MODE, TYPE, and PRINT.
2. Use an editor to write and modify an assembly language program.
3. Use an assembler to assemble a program and observe the assembler error messages.
4. Print a program listing using DOS commands.
5. Link a .OBJ file to produce a .EXE file which can be loaded and run.

NOTE: The computer commands or file names shown can be entered as either upper- or lower-case letters. Commands are shown either as single letters or as a group of letters surrounded

by <>. Letters contained in the <> represent a single key on the keyboard. For instance, the carriage-return key is shown as <CR>. The <Ctrl> key is always used with another key, in the same way that the shift key is used with another key. For instance, the command <Ctrl >-C, sometimes shown as ≠C, means "hold down the <Ctrl> key and then press C."

This exercise and the following exercises assume that the required programming tools are installed on a hard disk identified as C:, a path command has been inserted in the autoexec.bat file so that a path does not need to be specified as part of each command, and you are using a floppy disk in the A: drive to store your program files.

PROCEDURE

Starting (Booting) the System and Getting Started

1. Turn on the power to the computer and the power to the CRT monitor.
2. If your computer does not have a built-in clock, you may be prompted to enter the date. Enter the new date in the form shown on the screen. For example, June 30, 1991, would be entered as 6-30-91<CR>. Likewise, you may be prompted to enter the time. If so, enter the time in the format shown on the screen.
3. The machine will then respond with the Disk Operating System (DOS) command prompt C>. The prompt tells you that you are logged on the C: drive. This means that if you do not specify a drive letter in a command, the computer will assume that you mean the C: drive. When the prompt is present, the machine is ready for you to enter a DOS command. For reference, Figure 5-1 shows the functions of some useful DOS commands.

   As an example of how to use these commands, enter the command

   DIR/P <CR>

   You should now see a DIRectory listing of all the files and subdirectories on the C: drive.

   If your machine is connected to a printer, you can get a printout (hard copy) of a directory by entering the command

   DIR>PRN<CR>

4. Insert your formatted work disk into drive A: and gently close the drive door.
5. Type A:<CR> to change the logged drive to A: so that all your work files will automatically be written to your floppy.

Using Borland Turbo C++ Editor to Enter a Source Program

1. With the A> prompt showing, type

   tc <CR>

   to run the program. After a short pause the opening screen should appear. Press the

   F10

   key to put the cursor in the banner menu along the top of the screen. Each entry in this banner represents a menu of available commands. Use the arrow keys to move the cursor box to the leftmost entry in the banner and press the <CR> key. A menu of commands should appear.
2. Read through this menu to get an overview of the

DOS COMMAND	ACTION
DIR/P	List directory of disk in logged drive, pause when screen is full.
DIR/W A:	List wide directory of disk in A: drive
DIR A:>PRN	Send directory list of A: to printer
MODE LPT1:132	Set printer to 132 characters width
<F3>	Display on CRT last command typed. <Enter> to run.
<Ctrl>-S	Stop display scrolling, press any key to continue.
<Ctrl>-C	Cancel command
<Ctrl>-<Break>	Cancel command
<Ctrl>-<PrtSc>	Copy CRT screen to printer.
PRINT A:Filename.LST	Print hardcopy of Filename.LST from A:
TYPE A:Filename.ASM	List on CRT, Filename.ASM from A:
ERASE A:Filename.BAK	Erase Filename.BAK from A:

FIGURE 5-1 Useful DOS commands.

commands in this menu. For future reference, note that if the Turbo Assembler and Turbo Debugger are installed on your hard disk, you can access them directly from this menu. Use the arrow keys to move the cursor one step at a time across the banner. At each stop you should see a menu of the commands available under that heading. Don't try to remember all these commands; all you want to do at this point is edit a file.

3. When you finish looking through the menus, press the <Esc> key to get back to the edit window.
4. Type in the example program *exactly* as it appears in Figure 5-2.

NOTE: A couple of errors have been intentionally put in the program so that we can show you how to correct errors in an existing file.

If you make an error, you can move the cursor to the desired position on the screen with the arrow (cursor-move) keys, and correct the error. The following commands are useful for correcting errors.

<Del> key—Delete character over cursor
←key—Delete character to left of cursor
<CR>—Insert blank line
<Ctrl> key and <y> key—Delete line where cursor located

5. As you are typing in your source program you should save your work to disk about every 10 min so that it will not be lost if the power goes off. To do this you first press the <F2> key. When the Save Editor File window appears, type in the name you want to give the file and press the <CR> key. For this first example type in the avtemp.asm as the filename, and press the <CR> key.

NOTE: You only have to type in the filename the first time you save. After that the correct filename will already be in the window, so all

```
%PAGESIZE 66,132
;8086 PROGRAM for LAB 5                                  FILENAME: P5.ASM
;ABSTRACT       : this program averages two temperatures
;                 name  HI_TEMP and LO_TEMP and puts the
;                 result in the memory location AV_TEMP.
;REGISTERS USED: DS, CS, AX, BX
;PORTS USED     : None used
;PROCEDURES     : None used

DATA_HERE       SEGMENT

HI_TEMP         DB      92H     ; max temp storage
LO_TEMP         DB      52H     ; low temp storage
AV_TEMP         DB      ?       ; put average here

DATA_HERE       ENDS

CODE_HERE       SEGMENT
                ASSUME CS : CODE_HERE, DS : DATA_HERE

START:  MOV     AX, DATA_HERE   ; Initialize data segment
        MOV     DL, AX
        MOV     AL, HI_TEMP     ; Get first temperature
        ADD     AL, LO_TEMP     ; Add second to it
        MOV     AH, 00H         ; Clear all of AH register
        ADC     AH, 00H         ; Put carry in LSB of AH
        MOV     BL, 02H         ; Load divisor in BL register
        DIV     BL              ; Divide AX by BL
                                ; Quotient in AL, Remainder in AH
        MOV     AV_TEMP, AL     ; copy result to memory
        NOP                     ; Simulate more program
        NOP
        NOP

CODE_HERE
                END START
```

FIGURE 5-2 Program to average two temperatures.

you have to do is press the <F2> key. Also note that the banner along the bottom of the screen reminds you that the <F2> key performs the save operation.

6. When you have finished editing, save your file one last time as described before. Then return to the DOS prompt by holding down the <Alt> key and pressing the <X> key. The next step is to convert your program to machine language with the assembler. As we pointed out before, you can invoke the TASM assembler directly from the integrated environment you have been using to edit your file. However, in case you don't have this environment, we will show you how to assemble and link your program using the stand-alone assembler and linker.

Using the TASM or the MASM Assembler

1. If you have TASM you can assemble your source file simply typing at the DOS A:\> prompt:

   TASM filename,, <CR>

   In this command, filename is the name you gave your source file. You do not need to include the .ASM extension. The two commas after the filename in the command tell the assembler to automatically create an object file with the same name as the source file and an extension of .OBJ and to create a list file with the same name as the source file and an extension of .LST. If the assembler finds any errors in your source program, it will identify the errors in the .LST file.
2. If you have MASM, at the A> prompt type

   MASM<CR>

   After a short pause you should see on the screen a message which prompts you for the name of your assembly language source program file (source file). For the example here, type

   AVTEMP<CR>

   MASM assumes that the source file will have the extension .ASM. You are then prompted to enter a name you want to give the object (machine code) version of your program. The <F3> key displays the last command line entry, so all you have to do is press the <F3> key and the <CR> key. The extension .OBJ will automatically be given to this file. You will then be prompted to enter the name you want to give the assembler list file. Press the <F3> key and the <CR> key again. You will then be prompted to enter the name you want for a cross-reference (XREF) file. You don't need an XREF file, so just press <CR>.
3. The assembler will then read your source program from the work disk and produce the object and list versions of the program. When the assembler finishes, it will list errors, such as illegal instructions, that it found in your source program.

We have included two errors in the example program to give you practice in using the editor to correct errors. The list file produced by the assembler points out where the errors occur in the program. Therefore, the next step is to print out the .LST file so you can identify the errors.

Making Program Printouts

1. To get a printout, first turn on the power to your printer.
2. Because the list file is wide, you have to set up the printer to accept 132 characters/line. To do this, type in the command

   MODE LPT1:132<CR>

   The printer will remain in this mode until you turn off the power to the printer or reboot the system.
3. To make the printout, type

   PRINT AVTEMP.LST<CR>

   If the message "Name of list device [PRN]:" appears, just press <CR>, and the list file will be printed out.

Making Corrections in Your Source File

1. To correct the errors in your source file using the Turbo C++ editor, at the A> prompt type

   tc<CR>

2. When the opening screen appears, it will show the last file you worked on. If this is not the file you want to edit, press the <F3> key to get to the Load a File window. Type the name of the file you want to edit, and press the <CR> key. Note that the banner along the bottom of the screen tells you that the <F3> key executes the Open File command.
3. Use the edit commands as described previously to correct the errors in your program. When you have corrected the errors, press the <F2> key to save the corrected version and then press the <Alt> and <X> keys to return to DOS.

   Use TASM or MASM to assemble the program again. Cycle through the edit-assemble loop until the display indicates no errors, then make a printout of the .lst file to use as a reference when you run and debug the program.

Linking Your Program to Produce a .EXE File

1. To link your program with the TASM linker, at the A> prompt type:

   TLINK filename,, <CR>

   In this command, filename represents the name of the .OBJ file for your program, but you don't need to include the extension when you type in the

filename. The two commas after the filename tell the linker to produce an executable file with the same name as the .OBJ file name and a .EXE extension and to generate a map file with the same name and a .MAP extension. When the linker finishes, it will respond with the message, "Warning: No STACK segment." Don't be concerned about this message; it is not important at the moment.

2. To use the MASM LINK program, just type

LINK<CR>

You will then be prompted for the name of the object file you want LINK to act on, so type

AVTEMP<CR>

The linker assumes that this file has the extension .OBJ.

You will then be prompted for the name of the run (.EXE) file. To answer this prompt, press

<F3><CR>

This gives the run file the same name as your object file but with an .EXE extension.

Next you will be prompted for the name of the LIST (.MAP) file. In response to this prompt, press

<F3><CR>

Finally, you will be prompted for the name(s) of any library (.LIB) file(s) that you want to link in. You are not using any library files in this example, so just press

<CR>

A library file, incidentally, contains object code versions of programs or program parts that can be used in writing other programs. For example, a program module to calculate square roots might be kept in a .LIB file and linked into a new program that needs it. Library files save writing these parts out each time they are needed.

When the linker finishes, it will respond with the message, "Warning: No STACK segment." Don't be concerned about this message, it is not important at the moment.

3. List the directory of your work disk to see if the files AVTEMP.EXE and AVTEMP.MAP were produced there.

4. Print a copy of the .MAP file for your program. This map shows the length, starting address, and ending address for each of the segments in the program. It is important for you to note that the addresses shown are not *absolute* addresses, they are *relative* addresses! Absolute addresses will be assigned when the program is run. In Experiments 6 and 7 we show you how to run and debug your program on a PC-type computer, and in Experiment 8 we show you how to download the program to an SDK-86 board for execution and debugging.

EXPERIMENT 6

Using DEBUG to Execute and Debug Programs

REFERENCES

Hall: Chapter 3

EQUIPMENT AND MATERIALS

IBM PC or PS/2-compatible computer

INTRODUCTION

In the previous exercise you created the source file for an assembly language program, assembled the source file to produce a .OBJ file, and linked the .OBJ file to produce a .EXE file. In this exercise you will use the DEBUG program which comes with DOS to run and test the .EXE file.

At the conclusion of this experiment, you should be able to perform the following operations on an IBM PC-compatible computer:

1. Use the DEBUG program to load the .EXE file into memory and run it.
2. Use the DEBUG commands to examine the results produced by a program and, if necessary, debug the program.

PROCEDURE

1. Insert your work disk in drive A: and close the drive door.
2. Use the DIR command to make sure that your work disk contains the files AVTEMP.ASM, AVTEMP.OBJ, AVTEMP.EXE, and AVTEMP.MAP, which you produced in Experiment 5.
3. Use the TYPE command to list the AVTEMP.MAP file on the screen (TYPE A:AVTEMP.MAP<CR>). This list shows the length, starting address, and ending address for each of the segments in the program. It is important for you to note that the addresses shown are not *absolute* addresses, they are *relative* addresses! The segment which comes first in the program is given a starting address of 00000H. The segment which comes next in the program will be given a relative starting address on the next paragraph boundary after the end of the previous segment, as shown. For future reference, write the starting addresses and lengths of the code and data segments shown on the map listing. DEBUG assigns absolute addresses to the segments when it loads the program into memory to be run.
4. If you don't have a copy of AVTEMP.LST handy, use the MODE command to set the printer for 132 columns and print a copy as described in Lab 5.
5. To load your program into memory, type

   DEBUG A: AVTEMP.EXE<CR>

   You should then see the DEBUG hyphen prompt.
6. Look over the list of debug commands in Figure 6-1. This list is only a selection of commands to get you started. The IBM Disk Operating System Manual (Chapter 8) explains the DEBUG program in detail.
7. The first thing to do after entering DEBUG is to determine the address in the code segment register. To display the contents of all the registers, type

   R<CR>

   Record the values in the CS and DS registers for future reference. The value in the CS register is the base address for the location where DEBUG loaded the code segment of your program into memory. Note that the value in the IP register is zero; this is the offset of the first instruction of your program from the start of the code segment. The value in the DS register at this point is meaningless, but when the program is run, DS will be initialized with the segment base for DATA_HERE. Note that the first line of your program is displayed below the register displays.
8. The starting address of the program is CS:0000, and from the .MAP file you know that the program is 18H bytes long. To run the program then, just type

   G = CS:0 18<CR>

   If the program terminated correctly, you should now see all the registers and their contents

```
DEBUG COMMANDS

Dump format:    D [range] <CR>
D CS:0<CR>      Display 80 bytes from start of code segment
D DS:0 2<CR>    Display 3 bytes from start of data segment
D <CR>          Display next 80 bytes from end of last dump


Enter format:   E addr [list] <CR>
E 0<CR><CR>     Examine the first byte in the data segment
E 0<CR>20       Examine & then change first byte in DS to 20H,
                Enter space to go to next addr, <CR> to end command
E 0905:0 2<CR>  Replace current contents of address 0905:0 with 02H


Go format:      G [=addr] [addr [addr]]<CR>
G=CS:00 18<CR>  Execute code from address CS:00 until IP=0018H
G 18<CR>        Execute code from present CS:IP until IP=0018H


Register format: R [register name]<CR>
R<CR>            Display contents of all registers
R IP<CR><CR>     Display contents of the IP register only.
R AX<CR>0<CR>    Display contents of AX and set AX to zero


Trace format:   T [=addr] [value]<CR>
T=0915:0<CR>    Execute one line of program starting at addr 0915:0
T<CR>           Execute next line of program
T 2 <CR>        Execute next two lines of program


Q<CR>           Terminate DEBUG
```

FIGURE 6-1 DEBUG command summary.

displayed. The DS register contains the segment base address for your data segment, DATA_HERE. To see the values in the data segment, type

D DS:0<CR>

This will dump 80H bytes of memory on the screen, starting at the beginning of DATA_HERE. The values for HI_TEMP, LO_TEMP, and AV_TEMP should be in the first three memory locations. These three memory locations should hold 92H, 52H, and 72H. The H is not displayed; it is assumed that you know that these are hexadecimal numbers. Note that as predicted by the .MAP file, the code for your program is located in memory on the next paragraph boundary up from the data.

9. You can specify both the start and stop addresses for a memory dump. To display the first three memory locations from the beginning of the data segment, for example, type

D DS:0 2 <CR>

The number 2 refers to the offset of the last byte you want to see.

10. To change the values of the data, type

E 0<CR>

The value of the first item in the data segment is displayed. To change the value, just type in a new value and press the space bar (<SB>) to go to the next location. To leave the value the same and go on to the next data item, just press the space bar. To terminate the command, press <CR>. For instance, to change the data in the program so that HI_TEMP = 50H, LO_TEMP = 20H, and AV_TEMP = 0H, just type

E 0 <CR> 50<SB> 20<SB> 00<CR>

NOTE: To use the E command with other segments, you have to specify the segment base (the number in the segment register) followed by a colon before the offset in the command.

11. Do a memory dump of the data segment to make sure that you have changed the data correctly. Predict the new value for AV_TEMP.
12. Once you have changed the data, run the program again and see if the result was as predicted.
13. Use the E command to restore the initial values to the three temperatures.
14. Experiment 4 discussed using the single-step command on the SDK-86 board. The DEBUG trace command allows you to run programs in a similar way. Look at Figure 6-1 to find the format of the trace command. Then execute the trace command by typing

T = CS:0<CR>

You will see the registers displayed on the screen after the first instruction has executed. On the line after the register display, you should see the present CS:IP address and the code and mnemonic for the next instruction. Each time you type T<CR>, the trace command will be repeated. After the third instruction executes, the registers will be displayed again. You should now see that the low byte of AX (AL) is 92H. After the ADD AL,[0001] executes, you will notice that AL = E4H (92H + 52H). See if you can predict for each instruction how the data in the registers

will be affected by that instruction. Use the trace command to execute one instruction at a time and check your predictions.

15. Use the trace command again and execute the first seven instructions of your program by typing

T = CS:0 7<CR>

You should now see the DIV BL mnemonic. Change the contents of the AX register to 66H by typing

R AX <CR> 66<CR>

Notice that the present value of AX is displayed after the first <CR>.

NOTE: The values of the other registers can be displayed or changed using the R command. Once the value of a register is displayed, you can either terminate the command (leaving the register contents unchanged) by pressing <CR> or type in a new value for the register contents followed by <CR>.

16. Continue the trace until IP = 18, then do a memory dump to see the value of AV_TEMP produced by the new value you loaded into AX.

17. To leave the DEBUG program, type

Q<CR>

NOTE: If you try to run your program to the middle of an instruction or if your program execution somehow jumps to memory set aside for the computer's own use, the computer may "lock up" and not respond to any keys. If this happens, you should remove your work disk from A: and press the <Ctrl>-<Alt>-<Del> keys all at the same time. If this does not reset the system, you will have to remove your disk, turn off the computer, wait about 1 min, and then reboot the computer.

EXPERIMENT 7

Using TURBO DEBUGGER to Run and Debug Programs

REFERENCES

Hall: Chapter 3

Borland TURBO DEBUGGER Manual

EQUIPMENT AND MATERIALS

IBM PC or PS/2-compatible computer

Mouse (highly desirable, but not absolutely required)

Borland TURBO DEBUGGER Version 2.0 or later

INTRODUCTION

In Experiment 5 you created the source file for an assembly language program, assembled the source file to produce a .OBJ file, and linked the .OBJ file to produce a .EXE file. In this exercise you will use the Borland TURBO DEBUGGER to run and test your program.

At the conclusion of this experiment, you should be able to

1. Use the TURBO DEBUGGER program to load the .EXE file into memory and run it.
2. Single step through a program, insert watches on variables, and examine and change the contents of registers or memory locations.

PROCEDURE

1. Insert your work disk in drive A: and close the drive door.
2. Type A: <CR> to make your work disk the logged drive; then use the DIR command to make sure that your work disk contains the file AVTEMP.ASM, which you produced in Experiment 5.
3. To work correctly with TURBO DEBUGGER, the .OBJ and .EXE versions of your program must contain the necessary symbol table and other debugging information. To get these in your files, first type

   TASM /zi AVTEMP,, <CR>

4. When the DOS prompt reappears, type

   TLINK /v AVTEMP,, <CR>

5. Use the MODE command and the PRINT command as described in Experiment 5 to make a copy of the new .LST file for your program, then read the symbol table at the end of the listing to get an idea of the type of information it contains.
6. Use the TYPE command to display the AVTEMP.MAP file on the screen (TYPE A:AVTEMP.MAP<CR>). This list shows the length, starting address, and ending address for each of the segments in the program. It is important for you to note that the addresses shown are not *absolute* addresses, they are *relative* addresses! The segment which comes first in the program is given a start address 00000H. The segment which comes next in the program will be given a relative starting address on the next paragraph boundary up after the end of the previous segment, as shown. For future reference, record the starting addresses and lengths of the code and data segments shown on the map listing. TURBO DEBUGGER assigns absolute addresses to the segments before it loads the program into memory.
7. To get TURBO DEBUG to load your program, type

   TD AVTEMP.EXE <CR>

8. When the TD screen appears, you should see the source version of your program in the main window, a Watch window along the bottom of the screen, and a "banner" containing a list of the available command menus along the top of the screen. There are two ways to pop up one of these command menus and execute a command in it.

   If you have a mouse, you can move the cursor to the File entry in the banner, for example, and press the left mouse key to pop up the File menu. When the File menu appears, you can move the cursor to the desired command and press the left mouse key again to execute the

command. If you pop up a menu and decide you don't want to execute a command in that menu, just press the <Esc> key to return to the main window.

If you do not have a mouse, you can get to the banner list by pressing the <F10> key. To pop up a menu of commands you press the key which corresponds to the first letter of the command menu name. For example, to pop up the File menu you press the <F10> key to get to the banner and then press the <F> key to pop up the menu. When the menu appears, you execute a command in the menu by pressing the key for the letter that is highlighted or is in a different color in the command name. If you pop up a menu and decide you don't want to execute a command in that menu, just press the <Esc> key to return to the main window.

To see how this works, use the mouse or the key sequence procedure to pop up the File menu and execute the DOS shell command. This command temporarily terminates TD so you can execute a DOS command. To return to TD where you left off, type

EXIT <CR>

9. To get an overview of all the TD commands, press the <F10> key to get to the banner, press the F key to pop up the File menu, and then use the left and right arrow keys to move along the banner. Don't try to remember all these commands, but try to find commands to run a program, trace a program, view registers, add watches, and set breakpoints. Press the <Esc> key to get back to the main window.
10. TD has extensive on-line help to help you understand how commands work. To give you an idea of how this works, press the <F10> key to get to the banner, press the <H> key to pop up the main Help menu, and then press the <I> key to bring up the Help Index. You should see the first page of a list of commands for which help is available. Press the <PgDn> key or put the cursor over the PgDn label and click the left mouse key to see the next page.

    To get the help message for a topic in the list, you use the arrow keys to move the highlighted box over the desired topic and press the <CR> key. To see how this works, find the Breakpoints entry in the Help Index, move the highlighted box to that line, and press the <CR> key. After you finish reading the message, press the <Esc> key to return to the main window.
11. Now, suppose that you want to observe register contents and watch the value of the variable AV_TEMP as you single step through your program.

    To observe the 8086 registers, go to the View menu and press the <R> key. A windowpane containing a list of the registers should appear on the screen. If this window is in a position that covers part of your program, go to the Windows menu and press the <S> key to select the Size/Move command. Use the arrow keys to move the register pane to the upper right corner of the main window and press the <CR> key to fix it in this position.
12. To put a watch on the variable AV_TEMP, go to the Data menu and press the <W> key for Add Watch. When prompted type AV_TEMP and press the <CR> key. The name of the variable and its current value should appear in the Watch window at the bottom of the screen.
13. To single step through the program, go to the Run menu and press the <T> key to execute the Trace Into command. When this command executes, a triangular cursor should appear to the left of the second instruction of your program. This means that the first instruction of your program has been executed, and TD is ready to execute the second instruction. Each time you press the <F7> key, TD will execute another instruction and update the register display. Observe how the values of the registers change as you use the <F7> key to step through your program. Stop when the triangular cursor moves to a position left of the last NOP instruction. If you step beyond this point, the computer will probably lock up and you will have to reboot to get it going again.
14. When you have stepped all the way through the program, observe that the value of AV_TEMP has changed from its initial value of 00 to a value of 72H. To observe the value of the other variables, pop up the View menu and press the <V> key for Variables. When you are through reading the Variables window, hold down the <Alt> key and press the <F3> key to close the window. If you are using a mouse, you can close a window by moving the cursor to the small square in the upper left corner of the window and pressing the left mouse key.
15. Now, suppose that you missed something on the way through and you want to run the program again. Go to the Run menu and press the <P> key for Program reset. The triangular cursor will move up to the first instruction to tell you that the program is ready to run again. Note that the AV_TEMP is reset to its initial value of 00.
16. Now, further suppose that instead of single stepping through all the instructions, you want to run to the DIV instruction and observe the contents of the registers at that point. To set a breakpoint on the DIV instruction, press the <F6> key until a _ cursor appears in the main window. Use the arrow keys to move the cursor to the DIV line, and press the <F2> key to set the breakpoint. If you decide that you don't want the breakpoint there, press the <F2> key again to toggle the breakpoint off. To get the register pane back, press the <F6> key until it comes back.

NOTE: If you are using a mouse, you can set the breakpoint by just moving the mouse cursor to

the DIV line and clicking the left mouse key. After the breakpoint is set, press the <F6> key to get the register pane back.

NOTE: It is a good idea to set a breakpoint on the last instruction of your program so that you don't accidentally RUN off the end of your program and lock up the computer.

To run to the breakpoint, go to the Run menu and press the <R> key to Run the program. As indicated by the triangular cursor at the left of the DIV instruction, execution goes to the DIV instruction and stops. The DIV instruction does not get executed.

17. As you can see in the register pane, the AX register now contains 00E4H. Suppose that you want to put some new value in AX and then execute the rest of the program to see what effect this will have.

Use the arrow keys to move the highlighted box in the register pane to the AX line. Hold down the <Alt> key and press the <F10> key to pop up the Register Modify menu and then press the <C> key to execute the Change command. When prompted, enter a new value of 00f8 and press the <CR> key. The new value should appear in AX.

Press the <F7> key to Trace through the rest of the instructions in the program. Remember to stop when the cursor is next to the last NOP instruction.

18. The final TD feature we want to officially introduce you to in this exercise is Back Trace. This command allows you to "unexecute" an instruction. To execute this command, go to the Run menu and press the B key. You should notice the triangular cursor on the left side of your program moves up one line. You can repeat this command sequence to go backward as many lines as you want in the program. The register pane and the Watches values will be updated to show the correct values.

19. In this exercise we were able to introduce you to only a few of the major commands of TD. Skim through the menus again now to get another look at the available commands. Then when you need some feature to help debug a more complex program, you can go through the menus until you find the command, use the help messages to get the syntax for the command, if necessary, and execute the command. For a fun example of this, perform a program reset, set a breakpoint on the last instruction in the program, and see if you can find and figure out how to use the Animate command.

20. To Exit from TD, hold down the <Alt> key and press the <X> key.

EXPERIMENT 8

Downloading Programs to the SDK-86 for Execution and Debugging

REFERENCES

Hall: Chapter 3

EQUIPMENT AND MATERIALS

IBM PC-compatible computer

SDK-86 board and power supply

Printer

Work disk containing AVTEMP.ASM program developed in Experiment 5

SDKCOM1.EXE program (source files shown in Hall Figure 14-27)

INTRODUCTION

As you learned in Experiment 6 and/or Experiment 7, the object code program produced by the TASM or MASM assembler is linked to produce a .EXE file, which can be loaded and run on an IBM PC- or PS/2-type computer. However, the .EXE form of the program cannot be downloaded to a board such as an SDK-86, because it does not contain the required absolute addresses. A program called EXE2BIN is used to convert the .EXE file to a .BIN file, which has absolute addresses and can be downloaded.

A program called SDKCOM1.EXE is used to download the .BIN file to the SDK-86 through an RS-232C-type cable. The SDKCOM1 program also allows you to run and debug the program by entering commands on the IBM PC keyboard.

The SDK-86 board has two monitor programs in its ROMs. One monitor program is the keypad monitor, which we showed you how to use in Experiments 1-4. The other monitor is a serial monitor. To use the serial monitor, you connect a CRT terminal or computer to the serial port of the SDK-86. Serial monitor commands are then entered on the terminal or computer keyboard.

The SDKCOM1 program makes the IBM PC *emulate*, or act like, a CRT terminal, so you can use the PC to execute SDK serial monitor commands. The SDK serial monitor commands let you examine and change registers, examine and change memory locations, run programs, set breakpoints, execute one instruction at a time, and so on, just as DEBUG and the keypad monitor commands do. With these commands you can run and debug the downloaded program. Here's how you use these tools.

NOTE: Commands are shown either as single letters or as a group of letters surrounded by <>. Letters contained in the <> represent a single key on the keyboard. For instance, the carriage-return key is shown as <CR>.

OBJECTIVES

At the conclusion of this experiment, you should be able to perform the following operations with an IBM PC-compatible computer and an SDK-86 microprocessor development board:

1. Use the EXE2BIN program to convert the .EXE file to a .BIN file, which can be downloaded to an SDK-86 board.
2. Download the .BIN file to an SDK-86 through a serial connection.
3. Run and debug a program using the SDK-86 serial monitor debugging commands.

PROCEDURE

Getting a File Ready for Conversion to a .BIN File

1. The AVTEMP.ASM file you used in Experiments 5, 6, and 7 must be modified slightly so that it can be converted to a .BIN file for downloading to the SDK-86. Specifically you must remove the START: label next to the first instruction of the program and you must remove the START after the final END in the program. Fix in your mind the fact that if you are going to run a program on a PC, you must include these two labels. If you are going to download a program to an SDK-86, you must leave these two labels out, or the program will not convert to a .BIN file.

   Use your editor to remove these two labels from the AVTEMP. ASM file.

2. Use TASM or MASM as described in Experiment 5 to reassemble the AVTEMP.ASM file.
3. Use the TLINK or LINK program as described in Experiment 5 to produce the .EXE file for your program.
4. Use the TYPE command to list the AVTEMP.MAP file on the screen. This list shows the length, starting address, and ending address for each of the segments in the program. It is important for you to note that the addresses shown are not *absolute* addresses, they are *relative* addresses! When you are going to download a program to an SDK-86, you use the EXE2BIN program to assign the desired absolute addresses to the segments.

Using EXE2BIN to Convert a .EXE File to a .BIN File

1. Enter the command

   EXE2BIN A:AVTEMP.EXE <CR>

2. The system may respond with the message, "Fix-ups needed—base segment (hex):." In response to this message, type

   0010<CR>

   Entering this value tells the EXE2BIN program to give a segment base address of 00100H to the first segment in your program.
3. When the DOS A> prompt returns, you are ready to download the .BIN file to the SDK and run it.

Using SDKCOM1 to Download and Run a .BIN File on an SDK-86

From a discussion earlier in this experiment, remember that the SDKCOM1 program has two functions. One of its functions is to download .BIN files from an IBM PC-compatible computer to an SDK-86 board RAM through an RS-232C type cable. The second function is to make the IBM PC function as a CRT terminal so that you can use the SDK-86 serial monitor commands to run and debug the downloaded program.

1. List the directory of your system disk to make sure it has the program SDKCOM1.EXE.
2. Make sure the SDK is configured for serial operation as described in Appendix A and that an RS-232C type cable is connected from the COM1 serial port of the IBM PC to the serial port of the SDK.
3. Make sure power to the SDK-86 is turned on.
4. To run the SDKCOM1 program, type

   SDKCOM1 <CR>

   After a short pause the SDKCOM1 sign-on message should appear on the screen.
5. To establish communication with the SDK (i.e., to run the serial monitor program on the SDK), press the SDK-86 key

   <SYSTM RESET>.

   Then press the key sequence "GFE00:0."

```
SDK-86 MONITOR & SDKCOM1 COMMAND AND EXAMPLES

L A:FILENAME.BIN<CR>        Load .bin program from A: drive

Q                           Leave the SDKCOM1 program


Display Memory format:      D[W] start addr [,end addr]<CR>
D 0010:0000,40<CR>          Display contents of 40H memory locations
                            starting at address 0010:0000


Go format:                  G [start addr][,breakpoint addr]<CR>
G 0010:0,25<CR>             Execute program from address 0010:0
                            to IP=25H


Substitute format:          S[W] addr, [[new contents],]*<CR>
S 0010:0,FF<CR>             Change contents of memory at 0010:0 to
                            the value FFH.


Examine Register format:    X [reg][[new contents],]*<CR>
X<CR>                       Examine contents of all registers
X AX 00<CR>                 Examine and zero contents of AX


Single Step format:         N[start addr],[[start addr],]*<CR>
N 0010:0<CR>                Execute one line of program starting at
                            address 0010:0
N 0010:0,,                  Execute three lines of program
```

FIGURE 8-1 Command Summary for Experiment 8.

The message "SDK-86 MONITOR, V1.2" should appear on the CRT of the computer. The "." prompt character on the screen below this message indicates that the SDK-86 is ready for you to enter a command on the IBM keyboard.

6. Figure 8-1 shows a list of the commands that are now available to you. The L and G commands are carried out directly by SDKCOM1. The rest of the commands are sent to the SDK and carried out by the SDK serial monitor program. The SDK-86 requires *uppercase* letters, so press

   <Caps Lock>

   To download your .BIN file, type

   L

   You will then be prompted to enter the name of the .BIN file you want to download.

7. For the example program here, type

   A:AVTEMP.BIN <CR>

   As the .BIN program is loaded, you should see on the CRT a display of program offsets, the old contents of each memory location, and the program bytes loaded into those locations. When the .BIN program is loaded, the "." prompt will reappear to tell you that the SDK is ready for its next command. If you enter an incorrect command, the SDK will simply display the incorrect command and the # symbol and then display a new prompt.

8. First, you want to use the D (display memory) command to find out if the program code bytes were correctly loaded into memory starting at address 00100H (0010:000). To see the contents of 40H memory locations starting with this address, type

   D 0010:0,40<CR>

9. Compare the bytes shown in these locations with those shown on the assembler listing. Note that address values missing on the assembler listing have been inserted in the program by LINK and EXE2BIN. Use the relative address values you wrote down from the LINK map file to help you find the start of the code segment in the memory display on the CRT.

10. To run the program on the SDK, you use the GO command, which has the format

    G start address, breakpoint address <CR>

    The start address must always be the starting address of the code segment for your program. This may not always be the starting address where you load your program into memory. You use the link map to find the starting address of a program's code segment relative to the starting address of the entire program. For the example here, the code segment comes after the data segment. The starting address for the code is 0010:0010.

    A breakpoint address must always be the first byte of an instruction. The instruction at a specified breakpoint address will not be executed. Therefore, the breakpoint address you specify in this command should be the address of the next byte after the last code byte for your program. Use the assembler list file to determine the address for a breakpoint that will allow the entire example program to execute, then return control to the monitor. In the following commands, the items inside the square brackets, [ ], are numbers that you must supply.

    The GO command for this example then will be

    G 0010:10,[breakpoint offset]<CR>

    NOTE: When you press the G key, the monitor responds with the contents of the CS register and the contents of memory at that location. Ignore this message, and type in the rest of the G command.

    When the G command finishes, the message showing the breakpoint address will appear and a new "." prompt will be displayed to tell you that the system is ready for the next command.

11. You could use the D command to see if the program produced the correct results in memory, but to expose you to a new command, we will use the S command. As shown in Figure 8-1, this command allows you to change the contents of a memory location as well as examine it. Use the assembler listing to find the memory address where the result (AV_TEMP) of the example program is put. Then type the command

    S [address],

    The contents of AV_TEMP should be displayed followed by a dash. If for some reason you want to change the contents of this memory location you can just enter the two hex digits you want in that location and press the comma key. The new value will be entered in, and the address and contents of the next memory location will be displayed. If you do not want to change the contents of a location, just press the comma key to step to the next location. Press <CR> to terminate the command.

    To experiment with this command, use it to change the value of AV_TEMP by typing

    FF,

    You should then see the next address and its contents. You don't want to change that, so terminate the command by pressing

    <CR>

Then use the G command to run the program again and the S command to see the new value of AV_TEMP.

12. Another very useful command you should become familiar with is the X command. This command allows you to examine and, if desired, change the contents of the 8086 registers. To see the contents of all of the 8086 registers displayed, type

X<CR>

To see and, if desired, change the contents of a specific register, type

X [register abbreviation]

[register abbreviation] means a two-letter code supplied by you from the following list of registers:

AX, BX, CX, DX, SP, BP, SI,
DI, CS, DS, SS, ES, IP, or FL

The display will show the contents of the specified register. If you want to change the value, type in the value you want. Then press the comma key to examine the next register in the sequence shown before, or press <CR> to terminate the command.

If you want to see and change the values in a series of registers, you can type

X [first register]

and then use the comma key to step through the sequence of registers.

13. The final command we have space to discuss here is the single-step command, which has the basic format

N [start address] ,

When this command executes, it will cause the SDK to execute one program instruction and stop.

When you type the N key, the monitor will respond with the current contents of the IP register and the contents of memory at that address. Type in the address you want execution to start from and press the comma key. The monitor will execute the instruction at that address and then display the address of the next sequential instruction. To execute that instruction, just press the comma key. At any point you can press <CR> to terminate the command and then use the D, S, or X commands described before to check program results. To continue single stepping where you left off, just type the N command again, followed by a comma.

14. Use the S command to change the value of HI_TEMP to 20H, LO_TEMP to 40H, and AV_TEMP to 00H. Predict the result this will produce for AV_TEMP and single step through the whole program using the N command. Use the D command to see if AV_TEMP has the value you predicted.

15. To terminate the SDKCOM1 program, just press Q on the computer keyboard when the "." prompt is present.

EXPERIMENT 9

Reading Characters from a Keyboard on a Polled Basis

REFERENCES

Hall: Chapter 4

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

IBM PC-compatible computer

Parallel output, ASCII-encoded keyboard with keypressed strobe (example: Jameco JE610). If an encoded keyboard is not available, the data inputs can be simulated with eight single-pole, single-throw toggle switches. The keypressed strobe can be produced with a debounced pushbutton switch.

INTRODUCTION

The previous experiments worked only with data that was already in memory or in 8086 registers. The programming problem in this experiment is to input ASCII data from a typewriter-style keyboard connected to some ports on the SDK-86 board. Figure 9-1 shows how the keyboard is connected to the ports. Here's an overview of how it all works.

When you press a key on the keyboard, a keyboard interface IC inside the keyboard detects the keypress, debounces the keypress, and outputs an 8-bit parallel code for the pressed key. The lower 7 bits of the 8-bit code are the ASCII code for the pressed key. The MSB of the 8-bit code is a parity bit that is used to detect transmission errors in some systems. The encoder IC in the keyboard also outputs a strobe signal to tell you when a valid key code is present on the eight data outputs of the keyboard. To get valid data from the keyboard, you first read and check the strobe signal line over and over again until you find that it is asserted. This process is referred to as *polling* the strobe signal line. When you find the strobe signal asserted, you then input the ASCII code for the pressed key from the keyboard data lines.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write the algorithm for a program that inputs parallel data from a keyboard on a polled basis, masks the parity bit of each code, and stores the resultant 7-bit ASCII codes in a series of successive memory locations (an array).
2. Write the assembly language program for this polled data input program.
3. Run, test, and, if necessary, systematically debug the program on an SDK-86 board.

PROCEDURE

1. The first step in developing a program such as this is to carefully define what you want the program to do. A good way to do this is to write down a list of all the tasks the program must do. For the program here, an appropriate list might be as follows.

   a. Initialize ports if necessary.
   b. Set up a data structure (array) for storing the data.
   c. Set up a pointer to the start of array.
   d. Poll keypressed strobe until it is found asserted.
   e. Input the 8-bit code from the keyboard.
   f. Mask the parity bit.
   g. Store the ASCII code in memory array.
   h. Get the array pointer ready for next code.
   i. Poll keypressed strobe until it is found unasserted.
   j. Loop back to look for the next keypressed strobe if ten codes have not been read in.

   Carefully think your way through this list.
2. The next step in developing a program such as this is to write an algorithm that implements the tasks in your task list. To help you write the algorithm for the preceding list, think about the following questions.

   a. Why is it necessary to poll the keypressed strobe until it is unasserted (low) before looping back to look for a high again?
   b. What programming structure can you use to represent polling the keypressed strobe over and over until it is found to be asserted, as in task 1*d*, or unasserted, as in task 1*i*?
   c. How can you indicate in the algorithm that

you want to input the codes for ten pressed keys and then stop?

3. When you have completed the algorithm, determine the assembly language instructions which will implement each part of the algorithm. Here are some comments and questions that should help you do this.

   a. If you are working with an assembler, you can use the DB and DUP directives to set up the array for the key codes. (See Hall: Figure 4-15*d*.)

   b. The 8255A ports are automatically initialized for input when the power is turned on, so you don't have to initialize the ports with instructions for this program.

   c. When you read the port that the strobe line is attached to, you must assume that the upper 7 bits of the byte read in are garbage. Think about what instructions you can use to determine if the strobe signal in the LSB position is asserted (high) and loop until it is asserted.

   d. How can you modify the instructions in step *c* so that they cause the program to loop until the strobe is unasserted (low)?

   e. What instructions can you use to make the program read in 10 key codes before going on?

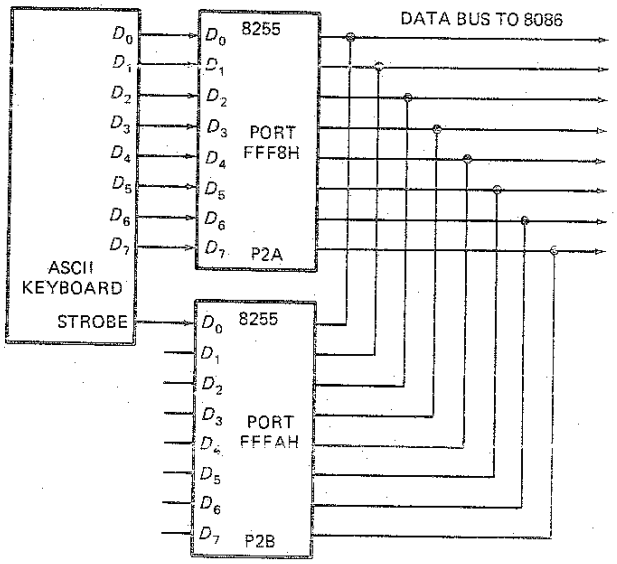


FIGURE 9-1 Microcomputer port connections for an ASCII-encoded keyboard.

4. Use your editor to create the source file for your program. Then assemble the program and use the editor to correct any errors found during assembly. When the program assembles without errors, link the .OBJ file to produce a .EXE file as described in Experiment 5.

5. Use the EXE2BIN program to produce a .BIN file and use the SDKCOM1 program to download the .BIN file to an SDK-86 board, as described in Experiment 8.

6. Use the serial monitor commands to run and, if necessary, debug the program, as also described in Experiment 8.

EXPERIMENT 10

ASCII-to-Hex Conversion

REFERENCES

Hall: Chapter 4

EQUIPMENT AND MATERIALS

IBM PC-compatible computer

Parallel output, ASCII-encoded keyboard with keypressed strobe, as used in Experiment 9

INTRODUCTION

In Experiment 9 you were given the problem of inputting characters from a keyboard on a polled basis, masking the parity bit, and then storing the character in memory. The characters input were in ASCII form.

The problem here is to write and test a program which converts the ASCII character codes for 0 through 9 and A through F to their simple 4-bit hexadecimal values. Once this program works, you will add the code to the program written for Experiment 9 so that the ASCII characters read in from the keyboard are converted to their hex values and then stored in another array in memory.

OBJECTIVES

At the conclusion of this experiment, you should be able to use a nested IF-THEN-ELSE programming structure to solve a common programming problem.

PROCEDURE

1. Look on an ASCII chart for the values of the characters 0–9 and A–F. Make a table of these values. Character codes other than these are considered illegal for this program. Indicate on your table the ranges of character codes that are illegal.
2. Write an algorithm for a program that reads a character from an array of ASCII characters, converts the character to its simple hex equivalent, and puts the hex value in an array. If the ASCII character is in one of the illegal ranges, FFH should be stored in that location.

   HINTS: You can use a nested IF-THEN-ELSE structure to determine the range of a character and take the appropriate action on that character before sending the result to the array.
3. Use the DOS copy command to make a copy of the source program you wrote for Experiment 9.
4. Modify the program to include the array for the hex values and the instructions needed to implement the algorithm you wrote for the ASCII to hex conversion.
5. Assemble your modified source program and, when it assembles correctly, link it to produce a .EXE file.
6. Run the EXE2BIN program to produce the .BIN file.
7. Turn on the power to the SDK-86 board and use the SDKCOM1 program to download the .BIN file to the SDK-86 board.
8. Use the SDK-86 serial monitor commands to run and, if necessary, debug your program.

EXPERIMENT 11

Generating Digital Waveforms on an Output Port

REFERENCES

Hall: Chapter 4 and Appendix B

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

IBM PC-compatible computer

Oscilloscope

2N3904 (or equivalent) NPN transistor

2.2-kΩ resistor

100-Ω resistor

Audio test speaker

INTRODUCTION

Once you send a signal to a port, that value remains latched on the port until a different value is sent to the port. So if you send a 1 to port pin D0 (the LSB of the port), that value remains on D0 until you send a 0.

To produce a continuous square wave on a port, you must send out a 1 or 0 in the LSB of AL to the port. Wait for a specific time. Change the LSB of AL to the other logic level and send it out to the port. Wait for the same amount of time as before. Repeat the sequence over and over. The alternating 1's and 0's produce a square wave. The length of the wait between outputs determines the frequency of the square wave. The desired wait is produced by executing a program loop which counts down a number previously loaded into a register or memory location and then exits. This loop is commonly called a *delay loop*.

For this experiment we want you to use a delay loop to produce a 7040-Hz square wave signal on pin D0 of port P1B on an SDK-86 board.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write a delay loop program section.
2. Calculate the value(s) needed to produce a desired delay.
3. Write a program to output a square-wave pulse train of a given frequency (musical tone).

PROCEDURE

1. Draw a section of the desired waveform.
2. Determine how many microseconds the waveform is high during each cycle and how many microseconds it is low during each cycle.
3. Write the algorithm for a program to output a continuous square wave on a port pin.
4. Write the assembly language program for your algorithm. Here are some notes to help you.

   a. The instructions required to initialize port P1B on the SDK-86 board as an output port are:

```
MOV DX,0FFFFH ; point at port control register
MOV AL,99H    ; control word makes P1B
              ; output port
OUT DX,AL     ; send control word
MOV DX,0FFFBH ; point DX to output port
```

   b. To produce the needed wait time for each half-cycle, use a delay loop such as the one shown here:

```
        MOV  CX,N
PAUSE: LOOP  PAUSE
```

   Later we show you how to calculate the value for *N*.

5. Find the jumper link W40/W41 on the SDK-86 board. The shorting plug should be in the W40 position for 2.45-MHz operation. Knowing that the clock frequency of the computer is 2.45 MHz, calculate its period (P)—i.e., the time for one microprocessor clock cycle.
6. Calculate the number of CPU clock cycles (C_T) required for each half cycle of your 7040 Hz square wave ($C_T = T/(2P)$).
7. The next step is to determine how many clock

cycles each instruction contributes to the required delay time. On a printout of your source program, write the number of clock cycles required for each instruction next to the instruction. Use Hall: Appendix B to get these values.

8. One half-cycle of the square wave is produced by an OUT to port instruction and the instructions up to the *next* OUT instruction. Find this sequence of instructions in your program. In this sequence identify the instructions that will execute only once each time through the sequence. Add up the number of clock cycles for these instructions to obtain the overhead clock cycles (C_O).
9. Identify the delay loop instructions which will be executed more than once each time through the sequence. Add up the number of clock cycles for these instructions to get the value for (C_L).
10. Calculate the number of times the delay loop is to be repeated (N) to give the required delay by using the following formula:

$$N = (C_T - C_O + D)/(C_L)$$

C_T = total number of clock cycles required for delay
C_O = total number of overhead clock cycles
C_L = number of clock cycles for actual delay loop
D = difference in clock cycles between loop executing or not

11. Insert the calculated value of N in your program.

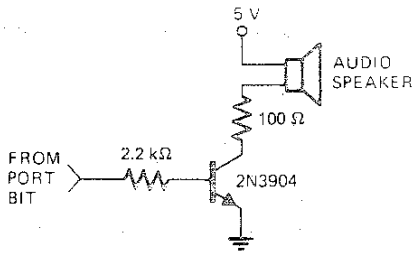


FIGURE 11-1 Transistor-buffered audio test speaker circuit.

12. Assemble your program and, when it assembles without errors, use LINK to convert the .OBJ file to a .EXE file. Then use EXE2BIN to convert the .EXE file to a .BIN file.
13. Use the SDKCOM1 program to download the .BIN file to the SDK-86 board and run it. With an oscilloscope observe the waveform on the LSB of the output port (D0, pin 16 of connector J6).
14. Determine the frequency of the observed waveform. If necessary, fine-tune the delay constant in your program to get the frequency as close to 7040 Hz as possible. Without too much difficulty you should be able to get the frequency of the signal within 100 Hz of the desired 7040 Hz and with some fine "tweaking" you can get the frequency within 50 Hz of the desired 7040 Hz.

NOTE: As a start you can use the SDK-86 monitor's substitute memory command to experiment with values of N. When you find the correct value, you can reedit, assemble, link, EXE2BIN, download, and run the program again. As you experiment with this, think about why you can't get the frequency to be exactly 7040 Hz. You might also think about whether it is more effective to add NOP instructions to the loop or to change the value of N.

15. If it is not already built, build the transistor buffer circuit shown in Figure 11-1 and connect it to an audio test speaker. Connect the circuit to the LSB of the output port. If your square wave is near 7040 Hz, you should hear a musical tone, A.
16. If you used the NOT instruction to complement the value in AL, substitute the INC AL instruction in place of the NOT instruction and run the program again. You should still hear the 7040-Hz tone on pin D0, but the other pins of the port should now have different frequencies.
17. The pin numbers for D0–D7 of the PIB port on connector J6 are as follows: D0–16, D1–12, D2–8, D3–4, D4–6, D5–10, D6–14, and D7–18. Move the input for the test-speaker buffer to bit D1 of the output port. What is the relationship between the frequency you heard on D0 and the frequency you hear now? If you cannot determine the relationship by listening to the two tones, use an oscilloscope to determine the relationship.
18. Repeat step 17 for the rest of the output port pins (D2–D7) in sequence. Think about how the program produces all these different frequencies.

EXPERIMENT 12

Working with Strings

REFERENCES

Hall: Chapters 5 and 6

EQUIPMENT AND MATERIALS

IBM PC-compatible computer

INTRODUCTION

A string is a series of the same type of data items in memory, usually ASCII or EBCDIC characters. The 8086 has string instructions that can work with strings of bytes and strings of words. We suggest that you read the discussions of the MOVS, CMPS, SCAS, and REP instructions in Hall: Chapters 5 and 6 before starting this exercise.

OBJECTIVES

At the conclusion of this experiment, you should be able to write programs which

1. Copy (move) a string from one memory location to another.
2. Find the occurrence of a character in a string.
3. Manipulate the contents of a string in memory.

PROCEDURE

1. Write an algorithm for a program that copies a string containing your name in the form CHARLIE T. TUNA from one string location in memory to a new string location which is just above the original location in memory.
2. Use your editor and the following hints to help you write the assembly language program for your algorithm.
   a. Declare a data segment named STRINGS_HERE, and use a DB directive of the form YOUR_NAME DB 'CHARLIE T. TUNA' to set up a string containing your name. The assembler will convert the ASCII characters between ' and ' to their hexadecimal equivalent.
   b. Use a DB directive of the form STRING_HERE DB_DUP(0) to set up the space to which you are going to move the string. The DUP(0) in the statement will cause zeros to be put in all the locations in the string when the program is loaded into memory.
   c. In the code section of your program you will use the MOVS instruction to transfer 1 byte from a string pointed to by SI (in the data segment) to a destination string pointed to by DI (in the extra segment). Think of the addition you must make to the ASSUME statement at the start of your program to let the assembler know the name of the destination segment.
   d. You must add instructions to initialize the ES register and the other registers used as pointers and counters for the string move instruction.
   e. If you are going to run the program on an IBM PC- or PS/2-type computer, make sure to include a START label next to the first instruction and a START label after the final END, as shown in Figure 5-2. If you are going to download the program to an SDK-86 for execution and debugging, omit these labels.
3. Assemble your program, and when it assembles with no errors, link the program to produce a .EXE file.
4. If you want to test and debug the program directly in the PC, you can use DEBUG as described in Experiment 6 or TURBO DEBUGGER as described in Experiment 7 to do so. If you want to run and debug the program on an SDK-86 board, use EXE2BIN to produce the .BIN file, then use SDKCOM1 to download, run and debug the program as described in Experiment 8. In either case you may find it instructional to single step through your program so that you can watch what happens to the values of DI, SI, and CX when the string instruction executes.
5. Now, write the algorithm for a program that looks for a carriage return character (0DH) in a string of 80 characters. If a carriage return is found, the number of characters up to (but not

including) the carriage return should be put in AL. If no carriage return is found, 50H (80 decimal) is put in AL.

6. Use your editor and the following hints to help you write the assembly language for this program.

   a. Determine the segment that is referred to by the scan string instruction (SCAS), and include the directives and instructions necessary to work with this segment.
   b. Use the DB directive to set up a test string.
   c. When the REPNZ prefix is used with the SCAS instruction, the string is scanned until a character with the value contained in AL is found or until CX is zero. If the character is found, the zero flag is set, the DI register will point to the character after the one found, and the CX register will be decremented by the number of characters that were scanned to find the carriage return. If the character is not found, the zero flag will be reset.

7. Assemble, link, run, and debug the program in the PC or on an SDK-86 board as described previously.
8. Write an algorithm for a program that moves a string containing your name in the form CHARLIE T. TUNA from a location in memory called YOUR_NAME to a new string location called LAST_FIRST. Your name will be placed in the new string location in the form TUNA, CHARLIE T. The only data you are allowed to supply is your name and the number of characters in your name.
9. Use your editor and the following hints to write the assembly language program for your algorithm.

   a. Use the SCASB instruction to find a period in your name string.
   b. Then increment the pointer to point to the first letter of your last name and use the MOVS instruction to move your last name to its new home.

      HINT: A drawing may help you keep track of where you need to start pointers, where they end up, and the required counter values.

   c. Set up the pointers again and use the MOVS instruction to move your first name and middle initial to their new home.

10. Assemble, link, run, and debug the program.
11. (OPTIONAL) Write the algorithm for a program that moves a string containing your name in the form CHARLIE T. TUNA from one string location in memory to a new string location, 4 bytes above it, in memory. Write the assembly language for this program, remembering that there is an 8086 instruction that causes DI and SI to be decremented in string operations. Assemble, link, run, and debug your program.

EXPERIMENT 13

Procedures—Nested Loops and Microcomputer Music

REFERENCES

Hall: Chapters 4 and 5 and Appendix B

EQUIPMENT AND MATERIALS

IBM PC-compatible computer

SDK-86 board and power supply

Oscilloscope

Frequency counter

Buffered speaker circuit from Experiment 11

INTRODUCTION

Experiment 11 showed how delay loops could be used in a program to output a square-wave pulse train of a given frequency. To lower the frequency, you have to make the delay-loop constant longer. Once the constant reaches FFFFH, the delay cannot be made any longer without increasing the number of instructions inside the delay loop.

To overcome this problem, nested delay loops are used. In this case a register or memory location is counted down, not just once as in Experiment 11, but *N* times.

The first part of this experiment is to produce a 2.5-Hz square-wave signal on pin D0 of port P1B (FFFBH) on an SDK-86 board. The second part of this experiment is to produce a 1760-Hz signal which pulses on for 0.2 s and off for 0.2 s continuously. The third part of this experiment, which is optional, asks you to produce a song on a speaker connected to a port.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Use a nested delay loops procedure to output a square-wave pulse train of a given frequency (musical tone).
2. Create a program to output a pulse train for a given time (musical note).
3. (Optional) Devise a program to output a series of different frequency pulses (musical song).

PROCEDURE

Nested Delay Loops

1. Study the flowchart for nested delay loops shown in Figure 13-1. The assembly language instructions for this flowchart are shown next.

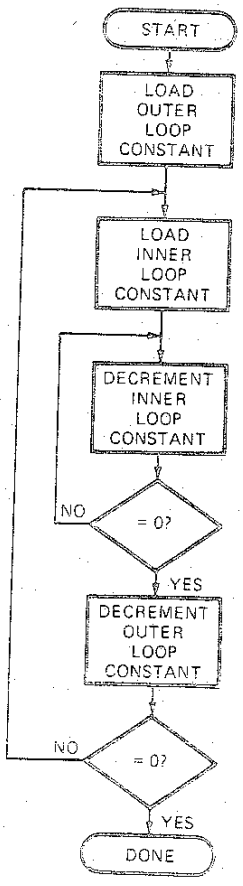


FIGURE 13-1 Flowchart of nested delay loops.

```
    MOV   BX,   N
L1: MOV   CX,   R
L2: LOOP  L2
    DEC   BX
    JNZ   L1
```

The letters N and R in the instructions represent constants for which you will later calculate values.

The instructions shown will form the heart of a delay procedure that will be called from the mainline of the program as needed to produce the delay for each half-cycle.

2. Write the algorithm for a program to produce the 2.5-Hz square wave on a port pin.
3. Write the assembly language for the program. Use the following hints to help you.
   a. Set up a stack segment so that the call instruction will have a place to hold the return address. Refer to Hall: Figure 5-8 to see how to do this. For a program that is to be downloaded to an SDK-86 board, omit the STACK directive or EXE2BIN will not convert the program.
   b. Initialize port P1B as an output port, as you did for Experiment 11.
   c. Initialize the stack segment register and the stack pointer register.
   d. Call the delay procedure.
4. Write the instructions for the delay procedure. Use Hall: Figure 5-10 to help you see the format for a procedure. Do not forget to put a RET instruction at the end of your procedure.
5. Make sure that the SDK-86 is working at 2.45 MHz (W40 shorting plug in place). Calculate the time required for one clock cycle.
6. Calculate the number of clock cycles required for each wait period (half-cycle) of the desired 2.5-Hz signal.
7. Identify the instructions which make up the program loop for each half-cycle. The overall program loop for each half-cycle includes an OUT instruction and all the instructions up to the next OUT instruction.
8. Write the number of clock cycles next to each of the instructions you identified as part of the overall loop.
9. Identify those instructions in the OUT to OUT delay loop that will be executed only once. Don't forget to include those instructions outside the body of the procedure, such as the CALL, and those instructions inside the procedure, such as the PUSHs, POPs, and RET. Add up the number of clock cycles for these instructions. This represents the overhead time (C_O).
10. The equation for the total number of clock cycles required is

$$C_T = C_O + N(C_L) - D$$

D is the difference between the JNZ jumping or falling through and, in this case, is equal to (16 − 4). C_L is calculated by adding up the clock cycles for the following instructions.

```
    MOV   CX,R:4 clock cycles
L2: LOOP  L2;(17R - 12) clock cycles
    DEC   BX;2 clock cycles
    JNZ   L1;16 clock cycles if jump taken
```

Here, $C_L = (4 + 17R - 12 + 2 + 16)$. The delay loop part of the program contains two unknowns, N and R. Choose some value of R for the inner loop constant and then calculate N, where N is the number of times the inner loop is to be repeated to give the desired delay. If the resultant value of N is not reasonable, try a different value for R and recalculate N until you get reasonable numbers for both constants.

11. Edit and assemble the program until there are no errors.
12. Use LINK and EXE2BIN, then download your program into the SDK-86.
13. Run your program and observe the waveform on the LSB of the output port with an oscilloscope. Debug the program if you do not see the desired waveform.
14. Determine the frequency of the observed waveform, using a frequency counter in period mode. If necessary, fine-tune the delay constants in your program to get the period as close to 0.4 s as possible. With little difficulty you should be able to produce a period between 0.399 and 0.401 s.

MUSICAL NOTE

A musical note is just a tone turned on and off. Suppose, for example, that you want to produce a 1-kHz tone that is on for 0.2 s and off for 0.2 s. You can produce the 0.2-s off part of the desired waveform by simply outputting a low to the port pin and waiting 0.2 s before starting the tone again. The tone on time of 0.2 s can be established by simply counting off the required number of half-cycles of the tone. Here's how this works.

The time taken for each half-cycle of the 1-kHz tone is 0.5 ms. Therefore, the desired tone on time of 0.2 s corresponds to 0.2 s/0.5 ms, or 400 half-cycles of the 1-kHz tone. In other words, all you have to do is produce a 1-kHz tone, as you did in Experiment 10, and count off 400 half-cycles of this tone. This then is essentially another example of nested loops. The inner loop produces half-cycles of the desired frequency. The outer loop counts the number of half-cycles required for a total time of 0.2 s.

A register or memory location is initially loaded with 400_{10}. After each half-cycle of the 1-kHz tone, this counter is decremented. Once the counter is zero, the program calls a procedure which turns the speaker off for 0.2 s.

1. Write the algorithm for a program to produce a note of 1760 Hz that is on for 200 ms and off for 200 ms.
2. Write the assembly language program for the on-

and-off 1760 Hz. To turn the tone on for 200 ms, count down the required number of 1760-Hz half-cycles. To turn the output off for 200 ms, you should be able to use the procedure you wrote for the first part of this experiment with little or no alteration.

3. If it is not already built, build the buffered speaker shown in Figure 11-1, and connect its input to the D0 pin of the output port.
4. Assemble, link, EXE2BIN, download, and run your program. If your program is running correctly, you should hear the result on the speaker.
5. Think about the following questions to help you understand how this program works.

   a. How can you change the time for which the note plays?
   b. How can you change the frequency of the note?
   c. Does the delay constant chosen for the frequency of the note affect the constant required for the time the note plays?

Playing a Song (Optional)

The program that outputs an on-and-off 1760-Hz square wave produces a one-note song. An improvement would be to have the program create a series of different notes.

1. Figure 13-2 is a flowchart for a song program. Write and test an assembly language program for this flowchart, using the following suggestions.

   a. Store the inner delay constant for each note in sequential memory locations. Then store the number of half-cycles for each note in another sequence of memory locations. (A group of sequential memory locations used to store data such as this is often called a *look-up table*, or just a table.)
   b. Use a register as a pointer to the offset of the frequency delay constant and another register as a pointer to the length delay constant.
   c. Initialize a memory location to the number of notes in your song. Alternatively, you could just repeat the song over and over.
   d. To play a note, load a register for the outside loop with the length constant from the look-up table in memory. Load a register for the inner loop with the frequency delay constant from the look-up table in memory. Count these values down for the duration of the note.
   e. When the note is finished, increment the memory pointers to point to the constants for the next note. (If you used DW for your data, increment the pointers by 2.)
   f. The 12 notes of the top octave and their frequencies are shown here. As you saw in Experiment 11, it is difficult to accurately produce frequencies such as 7040 Hz because of the lengths of the times required for 8086 instructions. To make your music sound better, we suggest you divide these frequencies by 4 or by 8 and calculate the number of microprocessor clock cycles accordingly.

C8	4186 Hz	F#8	5920 Hz
C#8	4434 Hz	G8	6270 Hz
D8	4699 Hz	G#8	6645 Hz
D#8	4978 Hz	A8	7040 Hz
E8	5274 Hz	A#8	7459 Hz
F8	5588 Hz	B8	7902 Hz

The length of time the note is to be played can then be calculated from these frequencies.

2. For your first song, choose just a 5- to 10-note sequence. For example, you might use the following 5-note frequencies for an interesting theme: 1760, 1976, 1681, 784, and 1175 Hz.

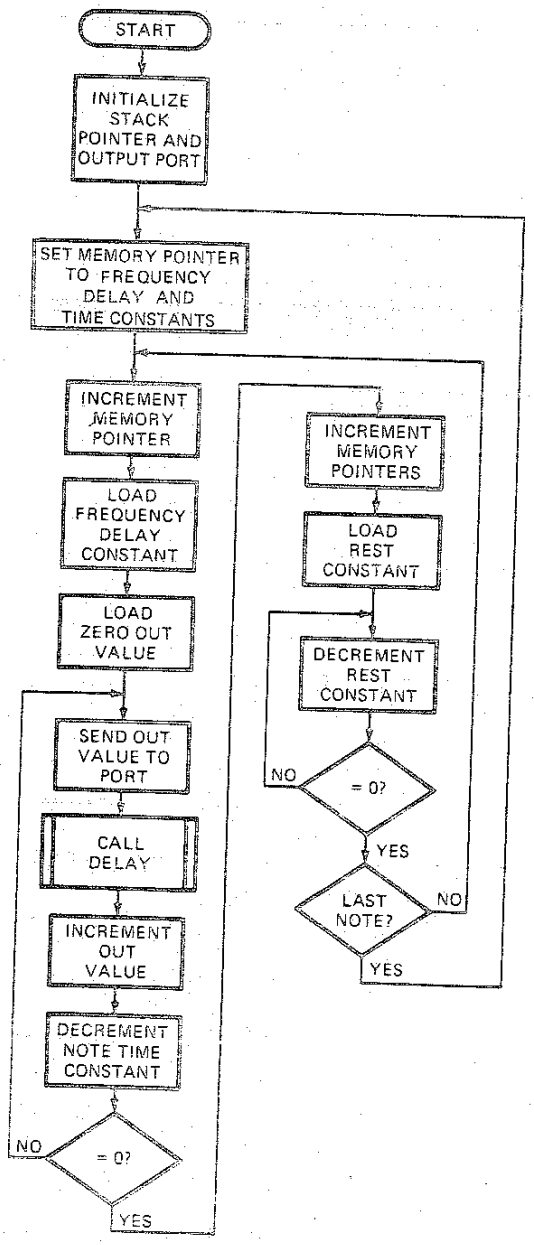


FIGURE 13-2 Flowchart for a song program.

EXPERIMENT 14

Procedures—Measuring Reaction Time with a Microcomputer

REFERENCES

Hall: Chapters 4 and 5 and Appendix B

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

IBM PC-compatible computer

Oscilloscope

7400 quad two-input NAND gate

LED

2 × 2.2-kΩ resistors

150-Ω resistor

Spring-loaded, pushbutton, single-pole, double-throw switch

INTRODUCTION

Experiment 11 used delay loops to time the output of pulses. Experiment 13 used a nested delay loop procedure to produce longer delay times. This experiment uses both types of delay loops and introduces the concept of individually testing portions of a program.

In this experiment you determine the time between two events by counting the number of times that a delay loop executes during the time between the two events. Specifically, in this experiment, you measure the time it takes to press a pushbutton switch after you see an LED light.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Use a microcomputer delay loop and counter to measure the time between two events.
2. Modify the operation of program parts so that they can be individually tested.

PROCEDURE

1. Write an algorithm for an assembly language program to

   a. Initialize the data and stack segments.
   b. Initialize port P1A as input and P1B as output.
   c. Call a procedure which gives you 2 s to get ready to press the pushbutton switch.
   d. Initialize a register as a counter.
   e. Set aside a memory location called R_TIME to store the final count, and initialize that memory location to zero with program instructions.
   f. Turn on the LED.
   g. Execute a 1-ms delay loop over and over until a high from the switch debouncer is detected on the pin of the input port. This indicates that you have pressed the pushbutton switch. Use the register you initialized to keep track of the number of times the delay loop executes before the button is pressed.
   h. When the button is pressed, move the count from the register to the reserved memory location.
   i. Turn off the LED.

2. Write the assembly language program for your algorithm. Include the following lines of code in your program for steps 1*d* onward.

```
;instructions for steps step 1d onward
        MOV   BX, 000H      ; initialize counter
        MOV   R_TIME, 0     ; zero reaction time
        MOV   DX, 0FFFBH    ; point to output port
        MOV   AL, 01        ; set LED-on bit
        OUT   DX, AL        ; light LED
        MOV   DX, 0FFF9H    ; ** point to input port
AGAIN:  MOV   CX, N         ; YOU CALCULATE N
WAIT_1MS: LOOP WAIT_1MS     ; wait for 1 ms
        INC   BX            ; **increment counter
        IN    AL, DX        ; **check if keypress
        AND   AL, 01H       ; **see if D0 high
        JZ    AGAIN         ; if not wait 1 ms
        MOV   R_TIME, BX    ; save reaction time
        MOV   DX, 0FFFBH    ; point at LED port
        MOV   AL, 00
        OUT   DX, AL        ; turn off LED
```

```
        JMP   EXIT          ; end program
;put procedure for 2-second delay here
EXIT:   NOP
        NOP
CODE_HERE ENDS
        END
```

3. Calculate an approximate value for the value *N* (just after the AGAIN label in the program) to give a delay of 1 ms. Edit and assemble this program until there are no errors, then make a hard copy of the list file.
4. To test that the delay loop is in fact 1 ms, a test program is made. This test program is a copy of the original program, which is changed slightly so that the delay loop can be tested. The changes made in the test program must not change the number of clock cycles in the delay loop. In order to test the delay loop, a continuous square wave must be output instead of inputting a signal on the input port pin (to see if the button has been pressed). Make a copy of your source file; then change the four lines of code marked with ** in step 2 in the following way.
   a. Replace MOV DX,0FFF9H with MOV DX,0FFFBH to point DX to the output port address, then replace IN AL,DX with OUT DX,AL.
   b. Replace INC BX with INC AX so that you can send the contents of the LSB (alternating 1's and 0's) to an output port.
   c. Replace AND AL,01H with AND BL,01H so that the delay part of the program executes over and over.
   d. Use the comment part of your program to note which instructions you changed.
5. Edit and assemble this test program until there are no errors.
6. Make a hard copy of the test program's list file, link the program, and convert it to the .BIN form using EXE2BIN. Download the .BIN file into the SDK-86.
7. Use an oscilloscope to observe the output signal. It should be a square wave with a 2-ms period (twice the delay loop time). Fine-tune the delay constant (*N*) in the test program until the period of the square-wave signal is as close to 2 ms as possible.
8. Once a value for *N* is obtained, insert this value in the original program.
9. If the value of *N* was changed in the original program, cycle through the edit and assemble, link, EXE2BIN, and download program development steps.
10. If they are not already built as a lab module, build and test the switch debouncer and the LED drive circuits, as shown in Figure 14-1.
11. Connect the debounced switch to the correct input port line and the LED driver to the correct output port line of the SDK-86.
12. Test the program. To get a true test of your reaction time, have someone else start the program running so that you can concentrate on pushing the button as soon as you see the LED light.
13. Rewrite your program so that it measures your reaction time for eight successive trials and calculates the average for the eight trials.
14. Test the modified program. Is the average value comparable with the individual results you obtained?
15. (Optional) Add the display program shown in Figure 20-1 to your program as a procedure so that your reaction time for each trial will be displayed on the SDK-86 LEDs.

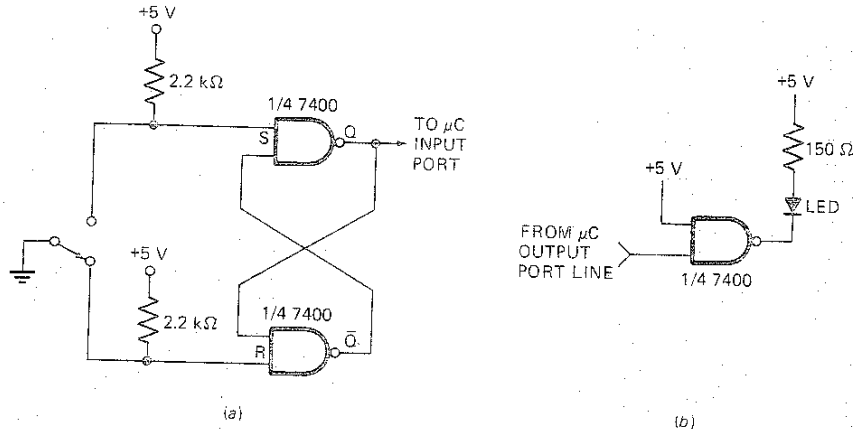


FIGURE 14-1 (*a*) NAND gate switch debouncer. (*b*) LED driver circuit.

EXPERIMENT 15

Microcomputer Circuits and Bus Signals

REFERENCES

Hall: Chapter 7

Intel: MCS-86 System Design Kit User's Guide

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

SDK-86 functional block diagram and schematics (Hall: Figures 7-7 and 7-8 or URDA schematics)

8086 timing diagram (Hall: Figure 7-1*b*)

Dual-trace oscilloscope

40-pin IC test clip (glomper clip)

16-pin glomper clip (We recommend the type of glomper clip with metal spring(s) and a metal hinge pin.)

INTRODUCTION

Previous experiments have shown you how to operate a simple microcomputer, such as an SDK-86, and how to write programs for it. The purpose of this experiment is to introduce you to some of the circuitry of the SDK-86 and the signals that appear on its buses as it executes programs. For this experiment you use just a simple dual-trace oscilloscope to observe microprocessor bus signals. The experiment was written for the original Intel SDK-86 board, but if you have the URDA, Inc. SDK-86 board, you should be able to easily adapt the instructions here to that board.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Identify the functions of ICs on a microcomputer schematic.
2. Equate a device symbol on a schematic with the physical device on the PC board.
3. Observe the address, data, and control bus signals of a microcomputer as it executes a simple memory read instruction sequence.
4. Describe how the lower 16 bits of addresses are demultiplexed from the 8086 data/address bus.

PROCEDURE

1. *a.* Study the SDK-86 functional block diagram in Hall: Figure 7-7 to determine the part numbers of the devices used for ROM, RAM, ports, address bus latches, data bus transceivers, and the clock generator.

   b. Write the part numbers in the appropriate spaces in the table below, then locate each of the parts on the SDK-86 board.

FUNCTION	PART NUMBER
CPU	_____
ROM	_____
RAM	_____
Parallel ports	_____
Address latches	_____
Data transceivers	_____
Clock generator	_____

2. *a.* Locate the clock generator IC on the SDK-86 schematic and study the circuitry surrounding it. The clock generator produces two output frequencies. The frequency of the CLK output is one-third the frequency of the attached crystal, and the PCLK signal is one-half the frequency of the CLK frequency.

   b. Calculate these two frequencies for your board.

   CLK = _____ MHz PCLK = _____ MHz

   c. Carefully clip a 16-pin glomper clip on the clock generator IC; then use an oscilloscope to see if the output frequencies agree with your predictions.

3. *a.* Inspect the circuitry to find out what determines which of these clock signals is

actually sent to the 8086 CLK input.

b. Find the W40 jumper pins in the upper right quadrant of the SDK board. Based on the way the jumper is installed, predict the clock frequency that is being sent to the 8086 on your board.

c. Gently clip a 40-pin glomper clip on the 8086, and use an oscilloscope to see if your prediction for the clock frequency applied to it is correct.

4. As explained in Hall: Chapter 7, the 8086 multiplexes the lower 16 bits of addresses out on the data bus. In other words, at some times the data bus is used for addresses and at other times it is used for data. The data bus may also be in an idle state when it is not being used by the 8086. When the 8086 is sending out an address on the data bus, it also sends out a signal called Address Latch Enable (ALE). The falling edge of this signal can be used to strobe the address in external latches. After the addresses have been latched, the data bus can then be used for data. In the next few steps of this experiment you will be looking at ALE, address, and data signals.
BE VERY CAREFUL NOT TO SHORT 8086 PINS TO EACH OTHER AS YOU MOVE THE SCOPE PROBE FROM ONE PIN TO ANOTHER.

a. To start, connect a probe from channel 1 of a dual-trace oscilloscope to the CLK signal on the 8086.

b. Connect a probe from channel 2 to the ALE signal on the 8086.

c. Set the scope to trigger on the ALE signal (channel 2).

d. Turn on the power to the SDK-86, and stabilize the display on the scope.

e. Observe that the ALE pulses do not come at regular intervals.

5. *a.* To produce predictable waveforms on the ALE, address, and data lines, enter the following simple program into SDK-86 RAM starting at address 100H. Then run the program.

```
0100   EB FE      HERE: JMP  HERE
0102   90                NOP
0103   90                 NOP
0104   04 55             ADD AL,55H
```

NOTE: The only instruction that will execute is JMP HERE. The other instructions are included so that the bytes which will be prefetched and held in the 8086 instruction byte queue are predictable.

b. As the program runs, stabilize the display of CLK and ALE on the scope. You should see a group of three ALE pulses, a space, and then another group of three ALE pulses. As you will see, each group of three ALE pulses and the space represents one execution of the JMP HERE instruction.

c. Count and record the number of clock pulses required for each execution of the JMP HERE instruction. Note that this number may be different from the number of clock pulses shown for the instruction in the data sheet because of the way the BIU and the EU interact during different instruction sequences.

6. The next step here is to determine the contents of the data/address bus as the JMP HERE instruction executes. The three ALE pulses during each execution indicate that three different addresses are output by the 8086 during the execution of this instruction.

a. To determine the three addresses, leave one scope probe on the ALE pin of the 8086 so that you can see when addresses are present on the data bus.

b. Then connect the second scope probe to the AD0 line of the 8086.

c. In the following chart, record the logic level present on this line during each of the three ALE pulses.

7. *a.* Move the scope probe from the AD0 line to the AD1 line of the 8086. Record in the table the logic level present on AD1 during each ALE pulse.

b. Repeat the process for lines AD2–AD15 from the 8086.

	AD15 AD14 AD13 AD12 AD11 AD10 AD9 AD8 AD7 AD6 AD5 AD4 AD3 AD2 AD1 AD0
1st ALE	
2nd ALE	
3rd ALE	
1st $\overline{RD}$	
2nd $\overline{RD}$	
3rd $\overline{RD}$	
Idle	

8. *a.* From the table you have just made you should then be able to read off the three addresses that are output by the 8086 during execution of the JMP HERE instruction. These three addresses are

____, ____, ____

b. Why does the 8086 output these three addresses during the instruction execution, even though the only instruction it needs is at the first address?

9. To see the result of the demultiplexing done by the addresses latches:

a. Connect one scope probe to the ALE signal on the processor and the other scope probe to the AD1 pin on the processor.

b. Sketch the observed ALE and AD1 waveforms for one execution of the JMP instruction (leave room below the waveforms so that you can later add another waveform for comparison).

c. Next study the SDK-86 schematics to find the address latch which demultiplexes the address bit from this line.

d. Identify the pin number of the device where address bit A1 is output.

e. Locate the device and the appropriate pin of the device on the SDK-86 board.

f. Move the scope probe from the AD1 pin on the 8086 to the A1 output pin on the address latch.

g. Sketch the observed A1 waveform below the AD1 waveform you drew before.

NOTE: The A1 waveform shows only address information.

10. The next thing you want to look at here is the data words that are read in from each address as the 8086 executes the JMP HERE instruction. Whenever the 8086 reads in data from memory or from a port, it will assert its $\overline{RD}$ signal. You can then use this signal as a reference for identifying read-in data words on the data bus.

a. Connect the second scope probe to the 8086 $\overline{RD}$ pin, so that you can compare this signal with the ALE signal.

b. Sketch the waveforms for one execution of JMP HERE.

c. Why are the $\overline{RD}$ pulses interleaved between the ALE pulses?

11. a. Leave one scope probe connected to the $\overline{RD}$ signal. Use the other scope probe to determine the logic level present on the 8086 AD0 line during each of the $\overline{RD}$ pulses and during the longer time interval between the last $\overline{RD}$ and the start of the next execution of the JMP instruction.

b. Record these values in the appropriate line of the table.

c. For lines AD1–AD15, observe and record in the table lines the logic levels during each $\overline{RD}$ pulse and the logic level for the interval between the last $\overline{RD}$ pulse of one execution and the start of the next execution.

12. a. Determine the word read in during each $\overline{RD}$ pulse. The hex values for these three words are

_____, _____, _____

b. Compare these words with the program values you loaded into memory. Explain what each of these three words represents.

The table you have built here is a picture of the activity on the data/address bus as the 8086 executes a simple JMP instruction. From this picture you should see how the 8086 multiplexes addresses on the data bus and how it overlaps instruction fetches and instruction execution to speed up processing.

Among other things, the next experiment shows you how an address output by the 8086 is decoded to produce a signal which enables the desired memory or I/O device.

EXPERIMENT 16

Address Decoders, I/O Operations, and WAIT States

REFERENCES

Hall: Chapter 7

Intel: MCS-86 System Designer's Kit User's Guide

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

Dual-trace oscilloscope

INTRODUCTION

The preceding experiment introduced you to the address and data bus signals produced by the 8086 as it executes a simple instruction. As explained in Hall: Chapter 7, when the 8086 sends out an address, some of the address signals go to address decoders. An address decoder converts the applied address signals and some control bus signals to a signal which enables the physical device(s) containing the desired address. In this experiment you will look at the control bus signals and the address decoder output signals produced as the 8086 executes simple instruction sequences. Also in this experiment you will see how WAIT states are inserted in 8086 machine cycles.

NOTE: This experiment was written for the Intel SDK-86 board. If you have an URDA, Inc. SDK-86 board, you can use the manual for it to help you adapt the instructions given here.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Describe how the chip select signals for the ROMs on the SDK-86 board are produced.
2. Describe how the chip select signals for the RAM devices on the SDK-86 board are produced.
3. Describe how the chip select signals for the port devices on the SDK-86 board are produced.
4. Draw a timing diagram which shows the effect of inserting one or more WAIT states during the execution of an 8086 write to port instruction.

PROCEDURE

1. Read the section "8086 and 8088 Addressing and Address Decoding" in Hall: Chapter 7. This section describes the purpose and operation of address decoders in a microcomputer system.
2. Read the discussion of the SDK-86 ROM decoder in Hall and study sheet 1 of the SDK-86 schematics in Hall: Figure 7-8.
3. From the schematic and the text discussion, determine which output on the address decoder PROM enables the ROMs which contain the keypad monitor program.
4. Locate the ROM decoder and the keypad monitor ROMs on the SDK-86 board. Then connect one probe from a dual-trace scope to the ALE pin on the 8086 and another scope probe to the address decoder output pin which sends an enable signal to the keypad monitor ROMs.
5. Turn on the power and press the RESET key so that the SDK-86 executes its keypad monitor program. Adjust the scope so that you can see stable traces of the two signals. Remember that the 8086 sends out an ALE pulse every time it sends out an address. Since the 8086 is fetching and executing instructions from the ROMs enabled by this decoder output, you should see the decoder output asserted (PROMs enabled) after nearly every ALE pulse. One case in which it is not asserted is when the 8086 addresses and reads a port to determine if you have pressed a key on the keypad.

   Draw a section of the observed waveforms to show the relationship of ALE and the decoder output signals.
6. Move the second scope probe to the address decoder output that enables the ROMs containing the serial monitor program. Think about why this pin isn't asserted.
7. Press the keys G FE00:0, to run the serial monitor program, and then use the scope to determine if pulses are present on the address decoder output pin which enables the serial monitor ROMs. Use the reference material to find the range of addresses that will cause this

output to be asserted.

8. In Hall, study the RAM section of the SDK-86 schematics in Figure 7-8, sheet 6, and the section of Chapter 7 which discusses RAM address decoding. As part of your report, include the following.

   a. Draw a RAM memory map similar to the ROM memory map shown in Hall: Figure 7-14.
   b. Identify which devices on the board contain the byte at address 00100H.
   c. Identify which devices contain the byte at address 00101.
   d. Indicate which outputs of the RAM address decoder PROM connect to the even byte devices.
   e. Indicate which decoder outputs connect to the odd byte RAMs.

9. Connect a scope probe to one of the decoder even-byte outputs and a scope probe to one of the decoder odd-byte outputs.
10. Enter the following test program in RAM with the code starting at address 00100H and data at address 00200H. Note that addresses contained in instructions are shown in high-byte, low-byte order as produced by TASM or MASM, but they have to be keyed in to the SDK-86 in low-byte, high-byte order.

```
0000                    DSEG SEGMENT
0200  2222        WORD1 DW   2222H
0202  1111        WORD2 DW   1111H
0204  0000        SUM   DW   0000H
0206                    DSEG ENDS
0100                    CSEG SEGMENT
      ASSUME CS: CSEG,          DS:DSEG
0100  B8 0000               MOV  AX,DSEG
0103  8E D8                 MOV  DS,AX
0105  8B 16 0200 AGAIN:MOV  DX,WORD1
0109  03 16       0202      ADD  DX,WORD2
010D  89 16       0204      MOV  SUM,DX
0111  EB F2       JMP       AGAIN
0113  CSEG       ENDS
                  END
```

Run the program and check that it is working by examining the SUM memory location.

   a. Run the program again and while the program is running, observe and draw a section of the decoder output waveforms you see on the scope.
   b. In your report explain why both decoder outputs are often enabled at the same time.

11. Study the port address decoder circuitry on sheet 7 of Hall: Figure 7-8 and the text section which discusses the operation of this circuitry. Then determine which output of the decoder PROM enables the parallel port device which contains addresses FFF9H, FFFBH, FFFDH, and FFFFH.
12. The short test program below inputs a byte over and over again from port FFFBH.

```
BA  FB  FF        MOV   DX, 0FFFBH
EC          AGAIN: IN   AL,    DX
EB  FD             JMP  AGAIN
```

    Enter the codes shown on the left of the program into RAM, starting at address 100H, and run the program.

13. Use a scope to determine if the port device enable signal on the output of the decoder PROM is being produced. Draw a section of the waveform you see on this decoder output.

    NOTE: Running a test program and checking if desired signals are present is an important microcomputer troubleshooting method!

14. Sometimes a memory or I/O device cannot read out or accept data fast enough to be compatible with the 8086. To solve this problem, we cause the 8086 to insert WAIT states. Read the section in Hall on how WAIT states are inserted in machine cycles and study sheet 2 of the SDK-86 schematics in Hall: Figure 7-8. Also, on the SDK-86 board, find the jumper pins which are used to specify how many WAIT states are inserted.
15. Connect one scope probe to the CLK signal pin and another scope probe to the M/$\overline{IO}$ signal pin of the 8086. Remove the WAIT state jumper from the SDK-86 WAIT state selector pins if one is present. With no jumper the 8086 will insert no WAIT states.
16. Check to see if the port test program used in step 12 is still present in memory. Sometimes touching the pins will alter memory. If the program is not there, reenter the program into RAM starting at address 100, and run it.
17. Draw a section of the CLK and M/$\overline{IO}$ waveforms this produces on the scope for one execution of the loop and use these waveforms to help you determine how many clock cycles are required for one execution of the loop.
18. Install a jumper on the 7 position of the WAIT state selector jumper pins on the SDK-86 board and run the test program again.
19. Draw a section of the CLK and M/$\overline{IO}$ waveforms this produces on the scope for one execution of the loop and use these waveforms to help you determine how many clock cycles are required for one execution of the loop now. Use the waveforms to help you determine where the extra WAIT states are inserted in the program loop.
20. Install a jumper on the 2 position of the WAIT state selector jumper pins and determine how many clock cycles the test program loop now takes to execute.
21. Install another jumper in jumper position W39 on the SDK-86 board and run the test program again.
22. Determine how many clock cycles the test program loop now takes to execute. Refer to the discussion in Hall to help you explain in your report why the loop now takes more clock cycles.

EXPERIMENT 17

Looking at Microcomputer Signals with a Logic Analyzer

REFERENCES

Hall: Chapter 7

Instruction manual for your logic analyzer

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

Logic analyzer with 16 or more input channels, clock qualifier feature, and internal clock feature

40-pin glomper clip

INTRODUCTION

The preceding two experiments showed you how to observe microcomputer bus and control signals with a standard dual-trace oscilloscope. No doubt you found that using a standard oscilloscope to observe these signals is somewhat tedious. With a logic analyzer you can take a "snapshot," which shows the waveforms or sequences of states present on many signal lines.

As explained in the references, a logic analyzer has 8 to 80 data inputs, depending on the specific model. Each time a logic analyzer receives a clock signal, it samples the logic levels on the data inputs and stores the samples in RAM. An external clock signal is used to produce a display of the sequence of states that a state machine or a microprocessor steps through. An asynchronous internal clock signal is usually used to make timing measurements, such as the time it takes for valid data to appear on the outputs of a ROM after the ROM receives a $\overline{CS}$ signal.

As long as the analyzer is receiving a clock signal, it is continuously taking samples and storing them in its memory. When the analyzer takes a new set of samples, it discards the oldest set of samples to make room for the newest set. A trigger signal is used to tell the analyzer when to stop taking samples and display the samples stored in its memory. You can use an internal "word recognizer" to generate a trigger signal when a specified binary word appears on the signal lines, or you can use an external signal to trigger the analyzer. If you want to see samples which occur after the trigger, you can tell the analyzer to *start* taking samples when a trigger occurs. If you want to see the sequence of samples that occurred before the trigger, you can tell the analyzer to *stop* when the trigger event occurs. Most analyzers also allow you to tell the analyzer to take some specified number of samples after the trigger event occurs. This mode allows you to see events that occurred on the signal lines before the trigger and events that occurred after the trigger.

A good, general approach whenever you use a logic analyzer is as follows.

1. Decide what signals on a microcomputer you need to look at, and connect the analyzer data inputs to these signal lines.
2. Decide which clock signal and which edge of that clock signal will cause the analyzer to take the samples needed to get a desired trace of the input signals. Connect the analyzer clock input to the selected signal line, and set the appropriate controls on the analyzer.
3. Decide whether you want the analyzer to trigger when a particular word is present on the input signal lines or when some external signal makes a transition. If you are using the internal word recognizer, select it and set it for the desired trigger word. If you are using an external trigger signal, connect it to the external trigger input of the analyzer, set the analyzer for external trigger, and for the transition (high to low or low to high) you want to trigger on.
4. Decide whether you want the display in timing diagram form or state table form and set the analyzer appropriately.
5. Press the START or the TRACE button on the analyzer to get the analyzer to take the specified data samples.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Connect and set up a logic analyzer as needed to see the sequence of addresses output by an 8086

as it executes a test program.

2. Connect and set up a logic analyzer as needed to see the series of code and data words read in from memory by an 8086 as it executes a test program.
3. Correctly use the clock qualifier feature of a logic analyzer to produce a trace of only data written to an output port or only the data written to memory.

PROCEDURE

NOTE: In order to have a specific example rather than just a generic description of how to use a logic analyzer, we discuss here the use of the Tektronix 1230 logic analyzer, which is commonly found in educational as well as industrial settings. One big advantage of the 1230 is that you can view many of the instruction manual pages by simply pressing the NOTES key on the front panel. If you have some other type of analyzer, you can use its manual to find the corresponding controls or menu choices. The basic 1230 can be expanded to sample as many as 64 input channels, but we have written Experiments 17, 18, and 19 so that you need only the basic 16 channels. If you have more channels available, you can extend the directions given here to include, for example, the upper 4 address lines of the 8086 and more control signals as appropriate.

1. If you are using a 1230, make sure a data pod is plugged into the A connector. *Always* make sure the power is off before plugging in a logic analyzer pod or removing a logic analyzer pod.
2. Gently put a 40-pin glomper clip on the 8086. Connect the data input leads of the A pod to 8086 AD0–AD15 [D0 bit of the A pod (black wire) to the AD0 pin of the 8086]. Connect the white wires from each section of the pod to pin 1 (ground) on the 8086.
3. Turn on the power to the analyzer and then turn on the power to the SDK-86 board.

NOTE: To avoid damage to the circuitry in the pods of an analyzer, make sure you *always* turn on the analyzer before you turn on the SDK-86 and *always* turn the SDK-86 off before you turn off the analyzer.

4. Enter the following test program in SDK-86 memory at 100H and check that it is entered correctly.

```
0100 EB FE      HERE  JMP HERE
0102 90               NOP
0103 90               NOP
0104 04 55            ADD AL,55H
```

5. Run the program. As you should see, this simple test program executes an endless loop, so that the sequence of addresses present on the address bus is easily predictable. Here's how you set up the 1230 to observe this sequence of addresses.
6. The 1230 analyzer is a *menu-driven instrument*, which means that you set up the analyzer for a desired measurement by working your way through a series of on-screen menus. To help you, the 1230 contains summaries of all the menu functions. Press the <NOTES> key to see the first page of these notes. After you have scanned through all the note pages, press the MENU key to leave the NOTES and then press the 0 key to bring up the first menu. The lighted boxes on the displayed menu contain parameters that you can change as needed for a specific application. To change the parameter in a box you use the number or letter keys specified in the menu. As a first example of how this works, press the <0> key or the <2> key to cycle through the possible settings for the Format parameter. You want the analyzer to take a set of samples each time an external clock pulse occurs, so leave the Format set for synchronous.
7. The four arrow keys in the edit box on the front panel are used to move the highlighted box to other parameters on the screen. Use the arrow keys to move the box to the Threshold parameter column and use the <0> key or <2> key to cycle through the possible settings. When you finish, leave the threshold set for TTL + 1.4 V. Use the <A> key or the <D> key to link all the probes together.
8. To get to the second menu (Channel Grouping) press the MENU key and then the 1 key. Move the highlighted box to the Radix column for the A channel and set it for HEX.

   Since you are only going to use the A pod (channel) for this measurement, you can turn off all the other channels to simplify the display. To do this, use the arrow keys to move the highlighted box to the first column of the B Channel Definitions and then press the <0> key to delete this column. Move to the next column and delete it, then repeat the process to turn off the rest of the B, C, and D Channel Definitions.
9. To get to the third menu (Trigger Spec) press the MENU key and then the <2> key. In this menu move the highlighted box to the Condition column and use the <0> key to cycle through the choices until an A is displayed in the box. The statement should then read "IF [A]*[0001] THEN [TRIG) & [FILL]." This direction tells the analyzer to trigger on the first occurrence of the condition specified by A and fill the memory with samples. You specify the condition, A, in the next menu.
10. To get to the fourth menu (Conditions), press the menu key and then the <3> key. Move the highlighted box to the first X in this menu and use the keys to enter 0100, the address you want the analyzer to trigger on so that it shows the sequence of addresses starting from the beginning of the test program.
11. To get to the fifth menu (Run Control), press the MENU key and then the <4> key. Here's a

summary of the options in this menu and the settings for this measurement.

a. The 1230 has four separate memories, each of which can be used to store a complete set of samples. As you will discover in a later experiment, you can store a trace from a working system in one memory, store a trace from a malfunctioning system in another memory, and then compare the two. A careful comparison often points to the source of a problem. For this first trace use memory 1.
b. The three choices for the Display option are State, Timing, and Disassembly. You will eventually get to use all these, but for this first exercise you want a State listing.
c. As the name implies, the Trigger Position option specifies the number of pretrigger and the number of posttrigger samples that will be displayed. Cycle through the possible settings and observe how the small cursor moves across the 0–2K window to indicate the trigger position. Set the position to 128.
d. Set the "Look for Trigger" option to "Immediately."
e. You will not be doing a comparison in this first exercise, so set the Compare option to Manual. Set compare memory 1 to memory [2]. Set Compare Memory Locations to 0000 to 2047. Set Use Channel Mask to A. Set Display Data to at least 9 s.

NOTE: This last setting will leave the trace displayed indefinitely.

12. The analyzer is now set up to make the measurement, but you still have to connect the signal that will clock the analyzer. The 8086 ALE signal indicates when a valid address is on the address bus, so it is appropriate to use this signal to clock the analyzer if you want to see a sequence of addresses. Use the 8086 timing diagram in Hall: Figure 7-1*b* to determine which edge of the ALE signal you want the analyzer to clock on so that the samples it takes represent valid addresses. Connect the 8086 ALE signal to the CLK1 input on the pod, and set the small switch in the pod for the ALE edge you determined.

NOTE: On some Tektronix logic analyzer pods you specify the desired clock edge by choosing different inputs on the pod.

13. Press the START button on the 1230 analyzer to tell it to do a trace of the data on the input lines. If the program is running and everything is connected correctly, a sequence of addresses should appear in a column along the left side of the screen. Record one sequence of these addresses and compare them with the expected addresses.

NOTE: After glomper clips have been used many times, some of the pins in the clip may get bent and therefore not make dependable contact with the IC pins. If you have carefully checked your analyzer setup values and the analyzer still refuses to do a trace, check with a scope to see if signals are reaching the top of the glomper clip where the analyzer is connected. If not, get help from your instructor.

14. Use the up and down arrow keys to scroll through the display. Observe that the trigger position, sample 128, is identified with TRIG in the left column. From this point on down the listing, you should see the address sequence 0100, 0102, 0104 repeated over and over. (A fast way to get to a desired sample is to press the ENTER key to invoke the JUMP mode, then use the numeric keypad to enter a 4-digit sample number such as 0246.
15. It is often useful to see a listing in binary instead of in hex. To change the Radix of a state display, press the E key to get to the proper mode; then press the <2> key until BIN appears in the Radix box at the bottom of the screen. Press the <ENTER> key on the analyzer to execute the command and return to the state display.
16. Your next exercise with the logic analyzer is to produce a trace of the data words read in from memory as the SDK-86 executes the test program. Use the timing diagram in Hall: Figure 7-1*b* to help you decide what signal you should clock the analyzer on to get a trace showing the just the data words read in from memory. Also determine which edge of this signal is best to use.
17. Connect the CLK1 input of the analyzer to the 8086 signal you decided was best to clock on for this trace and set the switch in the pod for the correct edge of this signal.
18. Reset the SDK-86, check that the test program is still correct in memory, and then run the test program again.
19. Go to the Conditions menu (#3), set the trigger word to FEEB, the first data word that will be read from memory when the program executes, and press the ENTER key.
20. Press the START button to do a trace. When the trace appears, record one sequence of the data words in the trace, and compare the results with the code words in the test program.
21. In the next part of this exercise you will use a clock qualifier to produce a trace which shows data bytes written to a port but does not show data bytes written to memory. If the clock qualifier is enabled, the analyzer will respond to a clock signal only if a specified logic level is present on the qualifier input at that time.

    To start, examine an 8086 timing diagram such as that in Hall: Figure 7-1*b* to determine which signal and which edge of that signal to clock the analyzer on for a trace of just data words written to memory or ports by an 8086. Also determine which edge of this signal is best

```
0000                    CSEG SEGMENT
                        ASSUME CS:CSEG
0000 BA FFFE              MOV DX,0FFFEH          ; Point to port control register
0003 B0 99                MOV AL,99H             ; Control word to set up output port
0005 B4 55                MOV AH,55H             ; Number to send to memory
0007 EE                   OUT DX,AL              ; Send port control word
0008 BA FFFA              MOV DX,0FFFAH          ; Point to output port addr
000B EE                 A:OUT DX,AL              ; Send incrementing count to port
000C 88 26 0200           MOV DS:BYTE PTR[200H],AH ; Write AH to memory
0010 FE C0                INC AL                 ; Increment port data
0012 EB F7                JMP A                  ; Do it again!
0014                    CSEG ENDS
                            END
```

FIGURE 17-1 SDK-86 test program.

to clock on.

22. Connect the CLK1 input of the analyzer to the 8086 signal pin you determined, and set the CLK1 switch in the pod for the edge you determined.
23. Next, look at the 8086 timing diagram to determine which 8086 signal indicates whether the 8086 is doing a memory operation or a port operation. Then determine the logic level that will be on this signal line when the 8086 is doing a write to port operation.
24. Connect the QUAL input of the analyzer to the 8086 signal you determined and turn on the QUAL switch in the pod. Set the QUAL level switch in the pod for the level that you determined will be present when the 8086 is writing to a port.
25. Enter the test program shown in Figure 17-1 into RAM, starting at address 100H. Remember that addresses contained in instructions are shown in high-byte, low-byte order.

    As you should see from the comments, this program outputs an incrementing sequence of numbers to port FFFAH and writes 55H over and over to memory location 0200H.
26. In the conditions menu set the trigger word for XX00 so the trace will start when the analyzer outputs 00 to the port. Press the MENU key to exit the conditions menu.
27. Press the START key to do a trace. If the trace does not correctly show the incrementing data bytes being written to the port as the low data bytes of the trace samples, see if you can determine why. When the sequence is correct, record 10 sequential values from the trace.
28. Modify the logic analyzer setup so that it will do a trace of only the data bytes being written to *memory* by the test program. Do another trace with this setup, and record the values for the first 10 samples of this trace.

EXPERIMENT 18

Further Practice with a Logic Analyzer

REFERENCES

Hall: Chapter 7

Instruction manual for your logic analyzer

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

Tektronix 1230 or similar logic analyzer

40-pin glomper clip

INTRODUCTION

The preceding experiment showed you how to use a logic analyzer to observe logic states present on the buses as a microcomputer executes a program. The goal of this experiment is to give you further practice using a logic analyzer to observe bus states, and to show you how to use an analyzer to measure the actual time between two events.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Use the analyzer to determine the sequence of addresses output and the instructions fetched as a microcomputer executes a program.
2. Use the analyzer in internal clock mode to determine the time that an address is present on the bus and the time that data is present on the bus.
3. Use the analyzer in internal clock mode to measure the time between RD going low and valid data appearing on the outputs of a ROM.
4. Use the analyzer to observe bus activity as the 8086 prefetches and executes instructions.
5. (Optional) Use a disassembler pod to produce a listing of the mnemonics for a sequence of instructions.

PROCEDURE

1. For the first exercise in this experiment, you will be using the analyzer to determine the sequence of addresses output and the sequence of code bytes fetched by the 8086 after the RESET input is asserted. As usual, you have to decide what signal to clock on and what word to trigger the analyzer on. To start, look at the 8086 timing diagram in Hall: Figure 7-1*b* to determine which system signal has active edges both when addresses are on the bus and when data is on the bus. Also determine which edge of this signal seems best to clock on.
2. With the power off, carefully put a glomper clip on the 8086, and connect the data inputs of the logic analyzer pod to the 8086 AD0-AD15 pins in order as you did in the preceding experiment. Connect the CLK1 input of the analyzer to the signal you decided to clock on for this trace. Set the CLK1 edge switch in the pod for the edge you decided would best catch addresses and data.
3. Turn on the power to the logic analyzer and then turn on the power to the SDK-86.
5. The logic analyzer menu setups for this trace are the same as those for the first exercise in Experiment 17 except that the trigger word is different. To refresh your memory, here is a summary of the menu settings.

   Menu 0— Sync, probes linked, TTL +1.4V
   Menu 1— A, HEX, +
   Menu 2— IF [A]*[0001] THEN [TRIG] & [FILL]
   Menu 3— A FFF0. This address is the lower 16 bits of the address that an 8086 goes to after RESET is asserted.
   Menu 4— Update Memory 1, Display [state]
   Trigger position [0128]
   Look for Trigger [Immediately]
   Compare [Manual]
   Display Data at least [9] seconds

6. Press the START button on the analyzer, and then press the RESET key on the SDK-86 board. After a short pause you should see a trace of the addresses output and the code words read in by the 8086 as it executes the start of its monitor program.

7. If you scroll the display so that the trigger point is visible on the screen, you should see a display such as that in Figure 18-1. Study the analyzer display to determine which values in the table represent addresses output by the 8086. Remember that the display shows only the lower 16 bits of the addresses, and note that address values are present for several clock cycles. Write the first 10 address values down in sequence in a column. Then study the display to determine the data words that are being fetched from each address. Note that the data words may be present for more than one clock cycle and that the bus may contain random values between addresses and data and between data and addresses. Write the fetched data words to the right of the address from which they are being fetched. These data words are the instructions being fetched from the monitor ROMs to be executed.

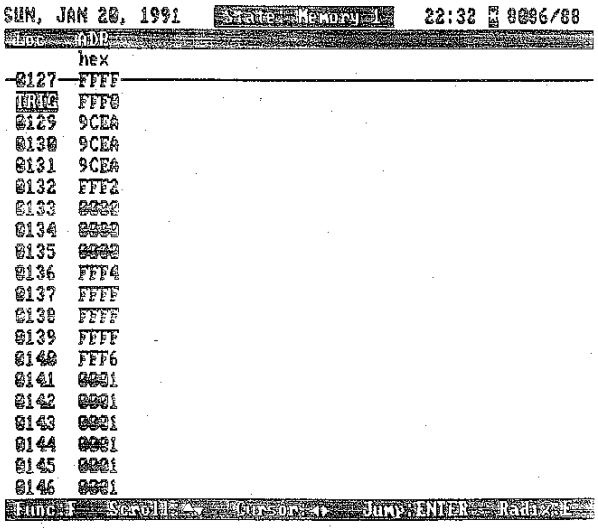


FIGURE 18-1 Logic analyzer display for RESET operation on SDK-86.

8. The first instruction (opcode) byte which is being fetched from the monitor ROMs is EAH, which represents a JMP instruction. In your 8086 reference book, determine the format for the instruction which corresponds to this opcode. Use the template for this instruction to determine the meaning of the next few fetched bytes. Calculate and write the destination address for this JMP instruction.
9. As part of your report, explain why the trace shows that the 8086 output address FFF06 and fetched a data word from that address before executing the JMP instruction.
10. Scroll through the display until you find the trace sample that represents the address where the 8086 jumped. Use the opcode templates in Hall to help you determine the first instruction that the 8086 fetches from this location and include it in your report.

The preceding section of this experiment should have shown you how you can use a logic analyzer to determine the basic execution sequence of an instrument for which you do not have a program listing. Doing it in the way shown here works, but it is somewhat slow and tedious. In a later, optional section of this exercise, we show how a logic analyzer with a *Disassembler Pod* is used to directly produce a trace which shows the mnemonics for a sequence of instructions.

11. The next exercise in this experiment involves using the internal clock feature of the analyzer to measure some simple bus timing. For this exercise leave the analyzer data inputs connected to AD0–AD15 as they were for the last exercise. Connect the QUAL input to the 8086 RESET pin, and then, on the pod, turn the qualifier on and set it for low. This will eliminate the effect of switch bounce during the manual reset operation.
12. Make the following modifications to the analyzer menu setups.
    a. Menu 0—Format = Async, rate = 40 ns, glitch = no
    b. Menu 1—Format = BIN
    c. Menu 2—No change
    d. Menu 3—No Change; triggger word still FFF0
    e. Menu 4—Display = timing
13. Press the START button to start the analyzer looking for a trigger and then press the SYSTM RESET button on the SDK-86 board. After a short pause a timing diagram should appear on the analyzer screen.
14. Use the left and right arrow keys to move the vertical line cursor across the display and note that binary values at the cursor location are displayed along the right edge of the screen.
15. Observe that the specified trigger word FFF0H is now present on the bus for several sample times. Determine and record how many sample times (internal clock cycles) this value is present on the trace by moving the cursor along one sample at a time with the left and right arrow keys. Multiply the number of sample times by the number of nanoseconds per clock that you set the analyzer for in menu 0. Record the number you calculated, and in your report discuss the resolution in nanoseconds for this measurement.
16. The next value on the bus represents a data word coming in from memory. Use the cursor to determine how many nanoseconds this data word is present on the bus and record your results.
17. For the next part of this experiment, we want you to measure the time between $\overline{RD}$ going low and valid data appearing on the outputs of the monitor ROMs. Because you don't need to see all the data lines to make the measurement, you can use one input to observe the $\overline{RD}$ signal. Remove the analyzer data input line from AD0 and connect it to the 8086 $\overline{RD}$ pin. In menu 3 set

```
0000                    CSEG SEGMENT
                        ASSUME CS:CSEG
0000 BA FFFE              MOV DX,0FFFEH              ; Point to port control register
0003 B0 99                MOV AL,99H                 ; Control word to set up output port
0005 B4 55                MOV AH,55H                 ; Number to send to memory
0007 EE                   OUT DX,AL                  ; Send port control word
0008 BA FFFA              MOV DX,0FFFAH              ; Point to output port addr
000B EE                 A:OUT DX,AL                  ; Send incrementing count to port
000C 88 26 0200           MOV DS:BYTE PTR[200H],AH   ; Write AH to memory
0010 FE C0                INC AL                     ; Increment port data
0012 EB F7                JMP A                      ; Do it again!
0014                    CSEG ENDS
                            END
```

FIGURE 18-2 SDK-86 test program.

the least significant bit of the trigger word for don't care (X), because you really don't know what level will be on this line when the trigger word occurs.

18. Press the analyzer START button and then the SDK-86 RESET key to do a trace. Move the display cursor to determine the number of clock cycles between $\overline{RD}$ going low and valid data appearing on the outputs of the monitor ROMs. Convert this time to nanoseconds and record it. Again, in your report discuss the resolution of this measurement.
19. The 1230 and several other common analyzers allow you to make timing measurements with greater resolution if you use fewer channels. In Menu 0 change the rate to 10 ns and then press the START button to do another trace. (Because the analyzer is looking at only the lower 4 lines, it will trigger immediately rather than waiting for a RESET.) Scroll along the waveforms until you find a place where $\overline{RD}$ goes low. Measure and record the time from this point until a data word appears on AD1–AD3. In your report compare this result with the previous result and explain why they might be different.
20. For the next part of the experiment, we first want you to take a more detailed look at how the 8086 reads instruction bytes into its queue during instruction decoding times and other times when the bus is not being used for reading or writing data.

    To do this, first enter the test program shown in Figure 18-2 into RAM, starting at address 100H, and run it. Remember that addresses contained in instructions are shown in high-byte, low-byte order.
21. Reconnect the analyzer's least significant data input to AD0 on the 8086. Set the Rate in Menu 0 to 200 ns. Set the trigger word in Menu 3 for FFFA, the address that will be on the bus when the analyzer outputs a data byte to a port. Set the Display in Menu 4 to State. Press the Start button to do a trace.
22. The result you get should be similar to that in Figure 18-3. Copy the first 40 samples from your trace and explain what each sample or group of samples represent in the execution of the test program. To help you decipher this trace, remember that when the 8086 outputs a byte to one-half of the data bus, the contents of the other half are unpredictable.

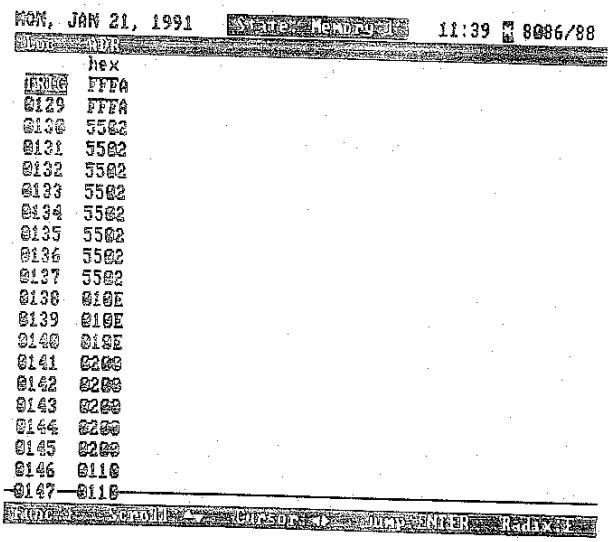


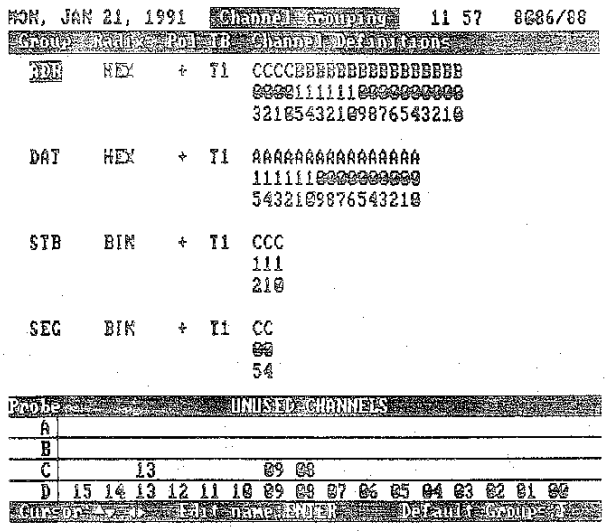
FIGURE 18-3 Logic analyzer display produced by test program in Figure 18-2.

Using a Logic Analyzer with a Disassembler Pod (Optional)

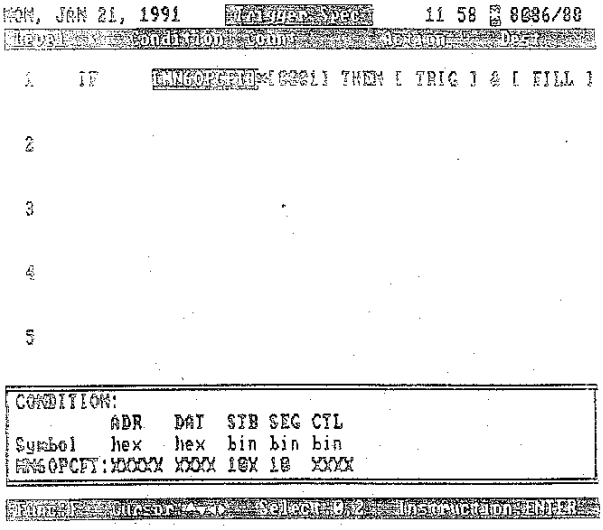
As we mentioned earlier, a disassembler pod allows you to produce a trace with the mnemonics for a sequence of instructions.

NOTE: This exercise requires a 1230 analyzer with at least 48 data channels.

1. Turn off the power to the SDK-86 board and then turn off the power to the analyzer. Plug the ribbon cables from the disassembler pod into the analyzer. Using anti-static precautions, gently remove the 8086 from the SDK and place it on some conductive foam. Plug the DIP at the other end of the pod cable into the 8086 socket on the board, and insert the 8086 into the socket on the probe cable.



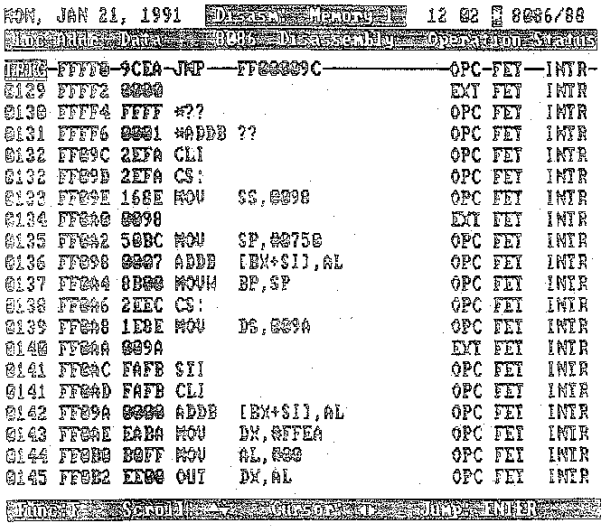
(a)



(b)

FIGURE 18-4 Logic analyzer disassembly pod setup and resulting trace for trace of SDK-86 RESET sequence. (*a*) Channel grouping setup. (*b*) Trigger spec and trigger conditions setup. (*c*) Resultant trace of SDK-86 RESET sequence.

2. Turn on the power to the analyzer and then turn on the power to the SDK-86 board.
3. If you give it permission, the analyzer will automatically do most of the necessary menu setups, based on the particular disassembler pod you have installed. For the particular measurement you want to do here, modify the menu settings so they to correspond as much as possible to those shown in Figure 18-4*a* and *b*.
4. Press the START button on the analyzer, then press the SYSTM RESET key on the SDK-86 board. The trace you get should resemble that in Figure 18-4*c*. A careful analysis of this trace should help you understand the trace you produced in the first part of this exercise.



(c)

EXPERIMENT 19

Microcomputer Troubleshooting

REFERENCES

Hall: Chapter 7

MCS-86 System Designer's Kit User's Guide

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

Oscilloscope

INTRODUCTION

When troubleshooting a malfunctioning microcomputer, a sequential systematic approach is much more effective than random poking, probing, and hoping that the machine will cure itself.

The purpose of this experiment is to show you a systematic troubleshooting approach, which we have successfully used on many microcomputer boards.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Describe several visual and tactile checks you should make on a malfunctioning microcomputer.
2. Describe how a power supply should be checked for proper operation.
3. Do a signal "roll call" on a malfunctioning microcomputer.
4. Describe a hardware single-stepping method of checking microcomputer bus signals during read and write operations.
5. Write a simple procedure to test microcomputer RAM.
6. Write a simple procedure to test a microcomputer port.

PROCEDURE

1. Assuming that you have been given an SDK-86 board which does not function correctly, the first step is to determine the problems it has. In other words, is it totally dead, or does it simply refuse to do certain operations, such as reading the keyboard or writing to ports? Apply power to the board and write the serial number of the board and a list of the problems you find.
2. Assuming the worst-case condition that the board does not function at all, first check that the power supply voltage is actually getting to the board. Then check that the power supply voltage is correct and has no more than 100–200 mV of noise or ripple.
3. If the supply voltages are present and correct, then do a visual and tactile check of the board. Some of the things you should do are

   a. Visually check the board carefully for any foreign objects, such as wire clippings, pencil lead, hair, solder, or anything else that might be shorting IC pins or circuit-board traces.

   b. Compare the board with an operating board. Check that all wire-wrap or jumper blocks are in the correct locations. Jumper blocks may have been removed or placed on the wrong pins.

   c. Gently touch all the ICs to see if any are excessively hot. As you touch the ICs on the SDK-86 board, you should notice that several ICs are quite warm. The question that probably occurs to you at this point is, How warm is too warm? The rule of thumb that we use for most devices is, if you can hold your fingers on the device for a few seconds, the device is not too hot. To give you a feeling for this, touch the 8086 on a functioning SDK-86 board. The 8086 should be quite warm to the touch, but bearable. If you find any hot ICs, make a note of which ones they are. Visually check to make sure they are inserted in their sockets with pin 1 pointed in the correct direction. Many ICs overheat if they are inserted backwards.

   d. Press on all ICs to make sure that they are firmly inserted in their sockets. ICs tend to walk out of their sockets if they are in printed-circuit boards that are flexed by keypresses or vibration. Also, visually check to make sure

none of the ICs have bent pins. A common problem we have found is that a pin will get bent when the IC is inserted. The IC will initially work; however, after some vibration or flexing of the board, the bent pin will no longer make contact, and so the IC stops working.

4. If the visual and tactile checks do not find any problems, your next step should be to do a "signal roll call" of the major CPU signals to point you towards the source of the problem. Here are some signals to check out with an oscilloscope.

   a. Check that the clock signal is present on the CLK pin of the 8086.
   b. Check to see if there is any activity on the address/data bus of the 8086.
   c. If there is activity on the bus, do a quick check to see if ALE and RD signals are present. Check to see if addresses are changing on the outputs of the address latches. Check to see if the OE signals are being produced for the PROMs containing the monitor program.
   d. If there is no activity on the address/data bus, check that the RDY input of the 8086 is not being held low by some malfunction in the READY circuitry. Check that the RST input of the 8086 is not being held high by some malfunction in the RESET circuitry. Check that the INTR and HOLD inputs of the 8086 are low.

5. Quick signal checks such as those listed in the previous step usually point you to a few ICs that may be causing the problem. For example, if there is no clock signal at the 8086 and the clock jumper is in place, then it is quite likely that the 8284 clock generator is not working. List the ICs you would suspect for each of the following signal conditions.

   a. The RDY input of the 8086 is continuously low.
   b. The RST input of the 8086 is continuously high.
   c. There are signals present on the address/data bus, but the signals on the outputs of the address latches do not change.
   d. The enable signal is not present at the OE inputs of the monitor ROMS.
   e. The address and data field LEDs all show 8's.

6. If you have an SDK-86 board which works correctly, you can use it to test the suspect ICs from the malfunctioning board, one at a time. This is a better approach than trying known good ICs in the bad board one at a time, because the malfunctioning board may have more than one bad IC, and it will still not work if only one of the bad ICs is replaced.

   NOTE: TURN OFF THE POWER BEFORE REMOVING OR INSERTING ANY ICs!

6. Before removing any of the ICs from the good board, mark each IC with a water-soluble marking pen so that you can always put the good board back to its original condition.

   From the good board, remove an IC that corresponds to an IC that you want to test, and put it on some conductive foam to avoid static damage. Then remove the suspect IC from the bad board, insert it in the good board, and see if the good board works correctly.

   If the good board works correctly, put the tested IC back in the bad board, put the good IC back in the test board, and test the next suspect IC from the bad board.

   If the good board doesn't work correctly, make sure you have inserted the suspect IC correctly. If you have and the board still doesn't work, try putting a known good IC of this type in the bad board. Then see if the bad board now works correctly.

   Repeat the IC test procedure until you have found and replaced any defective ICs in the bad board. Hopefully, the board is now ready to go back to work. One final test we like to make before returning an SDK-86 board to the lab is to make a RAM check.

7. Write an algorithm and an assembly language program to test the SDK-86 RAM in the address range 400H–7FFH. The program should write all 1's to each location, read the word back, and see if the word read back is equal to the word sent out. If an error is detected, the error word read back and the offset of that word should be left in registers. If the "write 1's" section of the program detects no errors, then the next section of the program should write all 0's to each word in the specified range and compare each word read back with the word sent out. Again an error word read back and the offset of that word should be left in registers. Locate your program in RAM starting at address 400H, so that it is not in the RAM area that you want to test.

8. Use a good board to run and test your program, then run it on the board you have just fixed. If an error is detected during the "write 1's" or "write 0's" tests, the error word and the offset left in registers can easily be used to determine the defective memory device. To see how this is done, look at the SDK-86 RAM circuitry in Hall: Figure 7-8, sheet 6, or the appropriate section of the URDA SDK-86 board schematics.

   On the original SDK-86 board the devices XA38, XA41, XA45, and XA43 contain RAM in the address range 0000–07FFH, so one of these devices must be defective if an error occurs in the range 400–7FFH. From the connections of these devices to the data bus, determine which device must be defective if the word read back during a "write 1's" test is FF7FH.

NOTE: Hall: Chapter 9 shows how to initialize ports for testing.

EXPERIMENT 20

Using 8086 Interrupts— A Real-time Clock

REFERENCES

Hall: Chapter 8

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

SDK-86 functional block diagram, Hall: Figure 7-5

SDK-86 schematics, Hall: Figure 7-6

1-Hz TTL level signal source or parts for circuit shown in Hall: Figure 8-11

Oscilloscope

INTRODUCTION

Many microcomputer systems keep track of real time (seconds, minutes, and hours) using an interrupt input and an interrupt-service procedure. A signal with a standard, stable frequency is applied to an interrupt input. The interrupt-service procedure for that interrupt then simply increments a count of interrupts (time intervals) kept in memory locations. If the interrupt signals are coming at a rate of one per second, then the count being kept is a count of seconds. The purpose of this experiment is to develop a program which updates a count of seconds, minutes, and hours in three memory locations and displays the current time on the SDK-86 LEDs.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Set up the structure for a multimodule program which consists of a mainline program, an interrupt procedure, and a display procedure.
2. Write the 8086 instructions necessary to load the interrupt-vector for a given interrupt type in the 8086 interrupt vector table.
3. Write an algorithm for a procedure that updates a count of seconds, minutes, and hours kept in three successive memory locations.
4. Write and test an interrupt procedure.

PROCEDURE

Hardware

1. Find the NMI input circuitry on sheet 2 of the SDK-86 circuitry in Hall: Figure 7-8. Note that a key labeled INTR on the keypad is connected through an inverter to the NMI input of the 8086. For this experiment you want to apply a 1-Hz signal to the input of this inverter. If your SDK-86 board has not already been modified, remove C33, a 1-μF capacitor. Insert and solder some connecting wires into the holes left vacant by removing the capacitor.
2. If you do not have a 1-Hz TTL level signal source available, build the 555 timer circuit shown in Hall: Figure 8-11. The 555 circuit is not very accurate, but it is sufficient for developing and testing your program. In the next experiment we show you how to produce a more accurate signal source. Do not connect the 1-Hz signal to the NMI input wires until you are ready to test your program. (Think about what will happen if you connect the signal before you run your program.)

Software

1. Your program for this experiment will consist of three major modules: a mainline, the interrupt service procedure, and a procedure which displays the contents of a register on the LED displays of the SDK-86. As pointed out in Hall, the advantages of writing a program in modules are that the modules can be individually written and tested and then linked together. Also, the individual modules can be reused in other programs. We give you the display procedure here in Figure 20-1, so all you have to develop are the mainline and the interrupt procedure.

   The mainline you need for your program is very similar to that in Hall: Figure 8-9*a*. Write your mainline, keeping the following points in mind.

   a. You need to set aside memory locations to store the current values for seconds, minutes, and hours.

```
%PAGESIZE 66,132
;8086 PROCEDURE TO DISPLAY DATA ON SDK-86 LEDs
;INPUTS:    Data in CX, control in AL.
;           AL  = 00H  data displayed in data-field of LEDs
;           AL <> 00H  data displayed in address field of LEDs.
PUBLIC  SHOWIT
DATA    SEGMENT WORD    PUBLIC
SEVEN_SEG       DB      3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H
                ;        0    1    2    3    4    5    6    7
                DB      7FH, 6FH, 77H, 7CH, 39H, 5EH, 79H, 71H
DATA ENDS       ;        8    9    A    b    C    d    E    F
CODE    SEGMENT WORD PUBLIC
        ASSUME CS:CODE, DS:DATA
SHOWIT  PROC    NEAR
        PUSHF                       ; save flags
        PUSH DS                     ; save caller's DS
        PUSH AX                     ; save registers
        PUSH BX
        PUSH CX
        PUSH DX
        MOV  BX, DATA               ; init DS as needed for procedure
        MOV  DS, BX
        MOV  DX, 0FFEAH             ; point at 8279 control address
        CMP  AL, 00H                ; see if data field required
        JZ   DATFLD                 ; yes,load data-field control word
        MOV  AL, 94H                ; no,load address-field control word
        JMP  SEND                   ; go send control word
DATFLD: MOV  AL, 90H                ; load control word for data field
SEND:   OUT  DX, AL                 ; send control word to 8279
        MOV  BX, OFFSET SEVEN_SEG   ; pointer to seven-segment codes
        MOV  DX, 0FFE8H             ; point at 8279 display RAM
        MOV  AL, CL                 ; get low byte to be displayed
        AND  AL, 0FH                ; mask upper nibble
        XLATB                       ; translate lower nibble to 7-seg
        OUT  DX, AL                 ; send to 8279 display RAM
        MOV  AL, CL                 ; get low byte again
        MOV  CL, 04                 ; load rotate count
        ROL  AL, CL                 ; Move upper nibble into low position
        AND  AL, 0FH                ; Mask upper nibble
        XLATB                       ; translate 2nd nibble to 7-seg
        OUT  DX, AL                 ; send to 8279 display RAM
        MOV  AL, CH                 ; Get high byte to translate
        AND  AL, 0FH                ; Mask upper nibble
        XLATB                       ; Translate to 7-seg code
        OUT  DX, AL                 ; send to 8279 display RAM
        MOV  AL, CH                 ; get high byte to fix upper nibble
        ROL  AL, CL                 ; move upper nibble into low position
        AND  AL, 0FH                ; mask upper nibble
        XLATB                       ; translate to 7-seg code
        OUT  DX, AL                 ; 7-seg code to 8279 display RAM
        POP  DX
        POP  CX
        POP  BX
        POP  AX
        POP  DS
        POPF
        RET
SHOWIT  ENDP
CODE    ENDS
        END
```

FIGURE 20-1 Display procedure for SDK-86 board.

b. You need to leave out the STACK directive in your program because you will be converting your program to binary using EXE2BIN so that you can download it to the SDK-86.

c. Use the PUBLIC directive to identify any labels or data in this module that must be accessible from one of the other modules. Use the EXTRN directive to tell the assembler that the interrupt procedure is located in another module.

d. You need to include instructions to load the starting address of the interrupt-service procedure into the interrupt-vector table, as shown in Hall: Figure 8-9*a*.

e. The program can simply cycle around a HERE: JMP HERE loop while it waits for an interrupt.

2. When you get the mainline module written, assemble it and correct any assembly errors you find.

3. The next step is to develop an algorithm for the interrupt-service procedure which updates the seconds, minutes, and hours counts in memory. After the seconds, minutes, and hours are updated, you will use the display procedure to update the display with the new values. If you think about how a digital clock functions, this algorithm is quite simple to write.

   To start, you know that each time an interrupt occurs, you want the seconds' count to be incremented by one and adjusted to decimal (BCD) format. If the resultant count is not equal to 60, you simply want to jump to the display update section and then return to the interrupted program. If the resultant count is equal to 60, you want to reset the seconds count to zero, increment the minutes count, and so on. In writing your algorithm, it is better programming form to group all the updating operations together and all the display operations together. As we said before, you will use the procedure in Figure 20-1 to actually update the display.

   After you have done a lot of thinking about the algorithm, write a flowchart or pseudocode representation for it.

4. The assembly language interrupt procedure for this algorithm will be written in a separate assembly module (file). After assembly, the object file for this module will be linked with the object module for the mainline. The .EXE file produced by the linker will then be converted to a .BIN file by EXE2BIN. The .BIN file will be downloaded to the SDK-86 board, run, and debugged. Here's an overview of the fastest way to develop and debug this module.

   First, write a simple version which just updates a decimal count of seconds in the memory location you set aside for this and then returns to the mainline. This simple step will allow you to determine if your interrupt-vector table is initialized correctly and if the interrupt mechanism is working correctly. After the program runs for a short time, you can stop execution and see if the contents of the seconds location in memory have changed. If this part works correctly, you can then add the instructions which update the minutes' and hours' locations. Here are some specific notes to help you develop this module of the program.

   a. Use Hall: Figure 8-9*b* to help you write the required PUBLIC and EXTRN statements.

   b. Don't forget that an interrupt procedure has to be type FAR, because the 8086 interrupt mechanism changes both the IP and the CS.

   c. To increment a decimal number in AL, you must use the ADD AL,01 instruction or clear the carry flag before using the INC AL instruction, so that the DAA instruction will work correctly.

   d. You must use an IRET instruction at the end of the interrupt procedure.

5. Assemble the interrupt module and correct any assembly errors.

6. Use the TASM or MASM linker to link the .OBJ module for the mainline with the .OBJ module for the interrupt procedure.

   For the TASM linker the command is

   TLINK filename1 filename2,, <CR>

   In this command filename1 and filename2 represent the names of the .OBJ files for your program, but you don't need to include the .OBJ extension when you type in the filename. The commas tell TLINK to generate a .EXE file with the same name as the first .OBJ file name and to generate a map file with the same name and a .MAP extension. When the linker finishes, it will respond with the message, "Warning: No STACK segment." Don't be concerned about this message; it is not important at the moment.

   To use the MASM LINK program, just type

   LINK<CR>

   You will then be prompted for the name of the object file(s) you want LINK to act on, so type

   filename1 filename2<CR>

   The linker assumes that these files have .OBJ extensions.

   You will then be prompted for the name of the run (.EXE) file. To answer this prompt, press

   <F3><CR>

   This gives the run file the same name as your first object file, but with a .EXE extension.

   Next you will be prompted for the name of the LIST (.MAP) file. In response to this prompt, press

   <F3><CR>

   Finally, you will be prompted for the name(s)

of any library (.LIB) file(s) that you want to link in. You are not using any library files in this example, so just press

<CR>

When the linker finishes, it will respond with the message, "Warning: No STACK segment. There was 1 error detected." Don't be concerned about this message; it is not important at the moment.

7. Use the EXE2BIN program to produce a .BIN file by typing

   EXE2BIN filename <CR>

   where filename is the name of the .EXE file.
8. Download the .BIN file to the SDK-86 board and run it as described in Experiment 8. Remember to use the .MAP file to find the offset of the start of the actual code in memory so you have the correct address for the GO command.
9. *After the program is running,* connect the 1-Hz signal from a signal generator or the 555 timer circuit to the NMI input leads. After a few seconds, disconnect the 1-Hz signal and stop the program. Examine the current value of the seconds' count and write the value down. Then start the program again, connect the 1-Hz signal for a few seconds, and examine the seconds' count in memory to see if it has changed. Testing this simple program allows you to make sure execution gets to the interrupt procedure when an interrupt occurs and that the basic "increment seconds' count" portion of the service procedure works correctly.

   NOTE: It is somewhat dificult to connect the 1-Hz signal to the NMI input without disrupting the program. You can usually get it in a couple of tries. An alternative would be to use a NAND latch debounced switch.

   If the seconds' count has not changed, you need to debug your program. For a start, you might check if the interrupt-vector table locations were correctly loaded with the starting address of the interrupt procedure. You might also run to a breakpoint at the start of the interrupt-service procedure to see if execution ever gets to that point in the program.
10. When you get the interrupt procedure correctly incrementing the seconds' location, the next logical step would probably be to add the instructions for the minutes and hours sections of your algorithm. However, it is much more fun to add the display section so you can watch the seconds', minutes', and hours' counts increment on the displays instead of having to examine memory locations.

    Carefully create an assembly module containing the display procedure shown in Figure 20-1. Note that you have to add appropriate SEGMENT and ENDS directives and the required PUBLIC and EXTRN directives to the module.
11. Assemble the display module and correct any errors.
12. Study the display procedure to find how parameters are passed to it. At the end of your *interrupt-service procedure* add a section which loads the current value of the seconds' count in the CL register and calls the display procedure to display seconds on the data field displays.
13. Reassemble the interrupt service procedure and correct any assembly errors.
14. Link the .OBJ files for the three modules using the command

    TLINK filename filename filename,,

    Use EXE2BIN to produce a .BIN file. Download the .BIN file to the SDK-86 and run it. After the program is running, connect the 1-Hz signal to the interrupt input. You should see an incrementing count of seconds on the SDK-86 display.
15. When you get the seconds' location incrementing correctly in response to the 1-Hz signal, add the instructions for the minutes' and hours' sections of your algorithm to the interrupt-procedure module. Also add instructions that will cause minutes to be displayed in the upper half of the data field and the hours value to be displayed in the lower half of the address field displays.
16. Reassemble this module; then link it with the mainline module and the display module. Convert the resulting .EXE file to a .BIN file with EXE2BIN, and download the .BIN file to an SDK-86.
17. Run and test the new program. Remember that because of the way NMI operates, you have to connect the 1-Hz signal after you start your program and disconnect it before you stop your program.

    NOTE: To test the minutes' and hours' sections without having to wait for a long time, you can load the memory locations with values such as 11:58:50 before you run the program.

18. When you get your clock working, congratulate yourself. The program you have developed is a real program used in many microcomputer-based control systems.

EXPERIMENT 21

Using an 8254 to Generate Real-time Clock Interrupts

REFERENCES

Hall: Chapter 8

EQUIPMENT AND MATERIALS

SDK-86 board with 8254 counter and address decoding circuitry added, as shown in Figure 21-1

SDK-86 schematics, Hall: Figure 7-6

Oscilloscope

INTRODUCTION

In Experiment 20 you wrote an interrupt procedure which updates counts of seconds, minutes, and hours. For that experiment, timed interrupt signals were produced by applying a 1-Hz signal to the NMI input of the 8086. The use of a 1-Hz signal from a signal generator or the 555 timer circuit is appropriate for developing the basic clock interrupt procedure, but for two reasons it is not practical for use in a final product.

- The 555 timer circuit is not very accurate or temperature-stable. In addition to its large size, a lab-bench signal generator may have similar problems.
- With a 1-Hz signal source, the smallest increment of time you can count off (resolve) is 1 s.

Both these problems can be solved by using a signal source with crystal-controlled frequency, such as the microcomputer system clock. The frequency of the system clock is too high to be applied directly to an interrupt input, so it must first be divided down in some way to produce a signal with the desired frequency. For this experiment you will use an 8254 programmable timer/counter to produce a 1-kHz interrupt signal from the SDK-86 PCLK signal.

The frequency of the PCLK signal is relatively stable, and with a 1-kHz interrupt signal the time resolution is 1 ms. To accommodate this higher-frequency interrupt signal, you can modify your program from Experiment 20 so that the basic seconds-minutes-hours section of the clock-interrupt procedure is entered only after 1000 interrupts have occurred.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Determine the system address for the counters and the control register in an 8254 counter in an 8086 system.
2. Construct the control word necessary to initialize an 8254 in a specified mode, and write the instructions required to send the control word to the 8254.
3. Determine the count that must be loaded into an 8254 for a given application, and write the instructions necessary to send the count to the 8254.
4. Modify a previously written clock interrupt procedure to work correctly with a different-frequency interrupt signal.

PROCEDURE

1. Figure 21-1 shows the circuit used to add an 8254 on your SDK-86 board. This circuitry can easily be wire wrapped on an SDK-86 board if wire-wrap connectors are used for J1 and J3 and wire-wrap sockets are used for the added ICs. Study this circuit, and make a truth table to determine the lowest address that must be present on the system address lines to assert the Y4 output of the 74LS138 and the $\overline{CS}$ input of the 8254. This address is the system base address for the device. Refer to Hall: Figure 8-15 to check if you determined the address correctly.
2. Refer to Figure 21-2 to determine the internal addresses for each of the counters and the control register in the 8254. Use these internal addresses and the base address from step 1 to determine the system address for each of the counters and the control register in this 8254. Note in Figure 21-1 that system address line A1 is connected to the A0 input of the 8254, and system address line A2 is connected to the A1

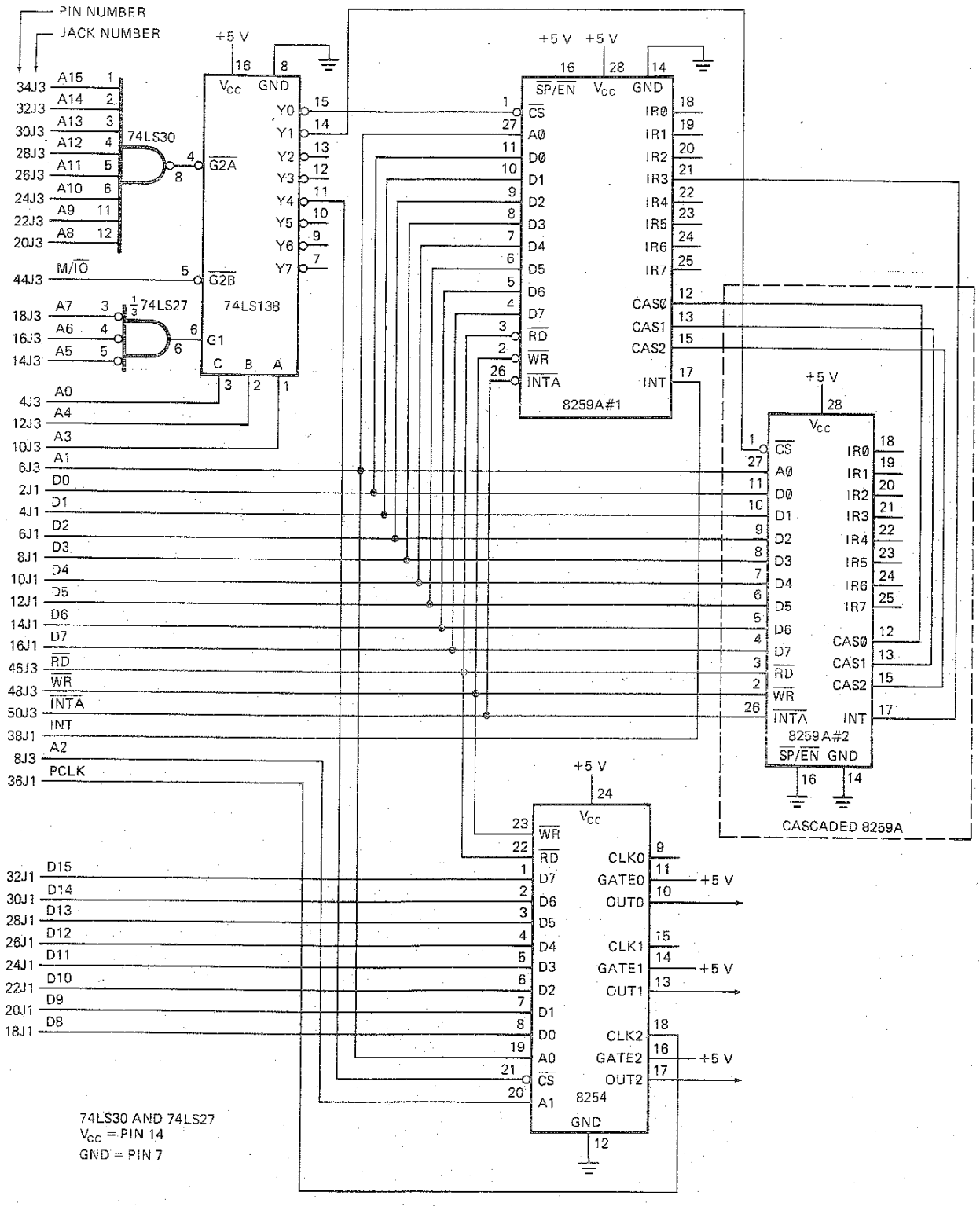


FIGURE 21-1 Circuit showing how to add an 8254 and 8259A to an SDK-86 board. This circuitry can easily be wire wrapped on an SDK-86 board if wire-wrap connectors are used for J1 and J3 and wire-wrap sockets are used for the ICs.

input of the 8254. Record these addresses so that you will know where to send the control word and count value.

3. For this experiment you will be dividing the 2.45-MHz PCLK signal down to produce the desired 1-kHz signal. Calculate the number by which the PCLK frequency must be divided to produce 1 kHz. This is the count that you will be loading into counter 2 of the 8254.
4. The next step is to build the control word you need to send to the 8254 to initialize it as needed for this experiment. Study the control word format and the bit descriptions for this control word in Figure 21-3.

   The two most significant bits of the control word specify which counter you want the control word to initialize. Assume that you are using counter 2, and record the appropriate bit pattern for these two bits of the control word. The bits labeled M1, M2, and M3 in the control word specify the desired operating mode. Square-wave mode is best to use for this case, so write the bit pattern which will select this mode in your control word. The bit labeled BCD in the control word is used to tell the counter whether to count down in binary or in BCD (decimal). If the count you are using is less than 9999, you can count it down in BCD instead of having to convert the count to hex. Write the appropriate bit in this position of your control word. Finally, you must determine what to put in the RW1 and RW0 bits of the control word. Since the count you need to write to the 8254 is larger than 255 decimal (FFH), you need to send the count as 2 bytes. Write the bit pattern for these bits that tells the 8254 to expect the low byte and then the high byte of the count.
5. In a previous step you determined the system address of the 8254 control register. Write the instructions which will send the control word to the 8254 control register.
6. You also determined the system address of each of the counters in the 8254 in a previous step. Write the instructions required to send first the low byte of the required count and then the high byte of the count to counter 2.
7. Copy the source versions of the mainline program and the interrupt procedure you wrote for Experiment 20 to files with new names so that you can develop the new program without altering the original working program. You can include a number in the filenames for the copies to help you identify them. (When developing programs, it is important to always keep a copy of your most recent version that works and assign version numbers so you can identify the latest version!)
8. Add the instructions you have developed for initializing the 8254 to the *copy* of the mainline for your clock program.
9. Assemble the new mainline module, link the .OBJ file with the .OBJ files for the other two modules, convert the .EXE file to a .BIN file, download the .BIN file to the SDK-86, and run it.
10. Use an oscilloscope to determine if the desired 1-

A_1	A_0	SELECTS
0	0	COUNTER 0
0	1	COUNTER 1
1	0	COUNTER 2
1	1	CONTROL WORD REGISTER

(*a*)

SYSTEM ADDRESS	8254 PART
F F 0 1	COUNTER 0
F F 0 3	COUNTER 1
F F 0 5	COUNTER 2
F F 0 7	CONTROL REG

(*b*)

FIGURE 21-2 8254 internal and system addresses. (*a*) Internal. (*b*) System. (*Intel Corporation*)

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC – SELECT COUNTER:

SC1	SC0	
0	0	SELECT COUNTER 0
0	1	SELECT COUNTER 1
1	0	SELECT COUNTER 2
1	1	READ-BACK COMMAND (SEE READ OPERATIONS)

RW – READ/WRITE:

RW1	RW0	
0	0	COUNTER LATCH COMMAND (SEE READ OPERATIONS)
0	1	READ/WRITE LEAST SIGNIFICANT BYTE ONLY.
1	0	READ/WRITE MOST SIGNIFICANT BYTE ONLY.
1	1	READ/WRITE LEAST SIGNIFICANT BYTE FIRST, THEN MOST SIGNIFICANT BYTE.

M – MODE:

M2	M1	M0	
0	0	0	MODE 0 – INTERRUPT ON TERMINAL COUNT
0	0	1	MODE 1 – HARDWARE ONE-SHOT
X	1	0	MODE 2 – PULSE GENERATOR
X	1	1	MODE 3 – SQUARE WAVE GENERATOR
1	0	0	MODE 4 – SOFTWARE TRIGGERED STROBE
1	0	1	MODE 5 – HARDWARE TRIGGERED STROBE

BCD:

0	BINARY COUNTER 16-BITS
1	BINARY CODED DECIMAL (BCD) COUNTER (4 DECADES)

NOTE: DON'T CARE BITS (X) SHOULD BE 0 TO INSURE COMPATIBILITY WITH FUTURE INTEL PRODUCTS.

FIGURE 21-3 8254 control word format. (*Intel Corporation*)

kHz signal is present on the OUT2 pin of the 8254 counter. If not, check that the PCLK signal is getting to the input of the counter, and check the addresses and control word you wrote are correct.

11. Once the 1-kHz signal is being correctly produced, the next point to think about is how this 1-kHz signal can be used to update the seconds', minutes', and hours' counts. An easy way to do this is to add another interrupt counter at the start of your interrupt procedure. If 1000 interrupts have occurred, 1 s has passed, so execution is sent to the instructions which update seconds, minutes, and hours. If 1000 interrupts have not occurred, execution is simply returned to the interrupted mainline. In flowchart or pseudocode form, write an algorithm for how you want execution to proceed when an interrupt occurs.
12. In the *copy* of the interrupt module make the additions needed to implement this algorithm. Here are some questions you might ask yourself to help with this.
    a. What addition do you probably want to make to the data segment to keep track of the number of interrupts?
    b. What must you do to the interrupt counter after updating the time?
13. After you have carefully made all the required additions to the interrupt module and corrected any assembly errors, link the three .OBJ modules, convert the .EXE file to a .BIN file, download the .BIN file, and run the program.
14. With the program running, connect the 8254 OUT2 pin to the 8086 NMI input pin. (Remember, you don't want to connect the interrupt signal to the NMI input until your program is running.)

    If you need to debug your program, think about the best places to put breakpoints so that you can see if execution reached those points correctly. In this program you should see obvious decision points that are good places to put breakpoints.
15. When your clock works correctly with this 1-kHz signal, watch it tick away for a while to celebrate your success. Then remove the connection to the NMI input before stopping your program.

In the next experiment you will add an interrupt mechanism that does not require you to manually connect and disconnect the interrupt signal.

EXPERIMENT 22

Initializing and Using an 8259A Priority Interrupt Controller

REFERENCES

Hall: Chapter 8

EQUIPMENT AND MATERIALS

SDK-86 board with 8254 counter, 8259A priority interrupt controller, and address decoding circuitry added as shown in Hall: Figure 8-14

SDK-86 schematics, Hall: Figure 7-8

Oscilloscope

INTRODUCTION

In Experiment 20 you wrote an interrupt procedure which updates counts of seconds, minutes, and hours. For that experiment, timed interrupt signals were produced by applying a 1-Hz signal from a 555 timer to the NMI input of the 8086. In Experiment 21 you improved the accuracy of the clock by using an 8254 programmable counter to produce the interrupt signal. The 8254 divides down the crystal-controlled PCLK signal to produce the desired interrupt signal. The next step in the evolution of this system is to use an 8259A priority interrupt controller to handle the interrupt signal instead of using the somewhat inconvenient NMI input of the 8086. The major goal of this experiment is to give you practice in initializing and using an 8259A. Devices such as this are very important in almost all microcomputer systems.

As explained in Hall, when an 8259A receives an interrupt signal on one of its IR inputs, it sends an interrupt request signal to the 8086 INTR input. If the INTR input has been enabled with an STI instruction, then the 8086 will push the flags on the stack, disable the trap and interrupt flags, and push the return address on the stack. The 8086 will also put its data bus in the input mode and send a couple of INTA pulses to the 8259A. In response to these INTA pulses, the 8259A sends the 8086 an interrupt type which you have programmed it to send for that IR interrupt input. When the 8086 receives the interrupt type on its data bus, it executes the interrupt service procedure pointed to by that interrupt type. The advantages of an 8259A are that it allows several different interrupt signals to be "funneled" into the 8086, and it allows interrupts to be selectively enabled or disabled as needed. This last feature means that you do not have to manually connect and disconnect the interrupt signal as you did when you used the NMI interrupt input.

OBJECTIVES

At the conclusion of this experiment, you should be able to

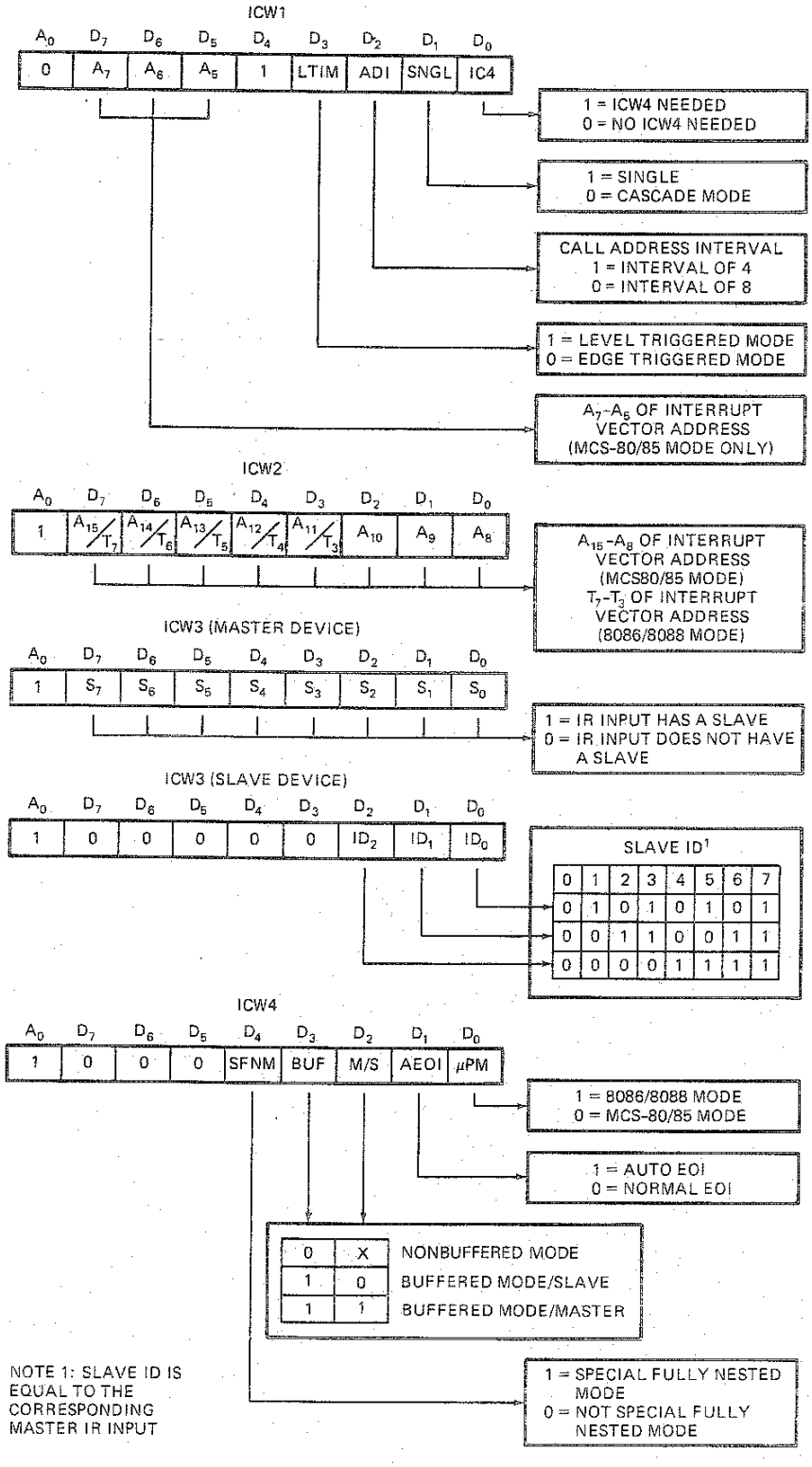
1. Determine the system addresses to which control words for an 8259A in a system must be sent.
2. Construct the control words necessary to initialize an 8259A in a specified mode, and write the instructions required to send the control words to the 8259A.
3. Write the instructions necessary to initialize the interrupt-vector table for the interrupt types sent by an 8259A.
4. Modify a previously written clock procedure to work correctly with an 8259A.

PROCEDURE

1. Figure 21-1 shows the circuit used to add an 8254 and an 8259A to your SDK-86 board. Study this circuit and locate the added ICs on your SDK-86 board. Make sure the PCLK signal is connected to the CLK2 input of the 8254. Connect a jumper wire from the OUT2 pin of the 8254 to the IR4 pin of the 8259A.

   For this connection to an IR input of the 8259A, you can simply use a *short* piece of wire. However, for longer connections in which the interrupt signal is coming to the IR input from hardware off the board, we found it necessary to use a line driver and terminated cable for the connection to prevent noise signals from causing unwanted interrupts.
2. Read the reference material in Hall to determine the additional tasks that you must do in the mainline of your interrupt-driven clock program so that it will properly initialize and use an

FIGURE 22-1 8259A initialization command word formats and sending order. (*a*) Formats. (*b*) Sending order requirements. (*Intel Corporation*)



8259A. Pay particular attention to Figure 8-31. The following questions should help you make up your list.

a. What mode instructions are needed to initialize the 8259A mode?
b. How do you enable the 8259A to respond to an interrupt signal on its IR4 input?
c. When the 8086 receives an interrupt type from the 8259A, where will it go to get the starting address for the interrupt procedure?
d. How is the 8086 enabled so that it will respond to an interrupt request from the 8259A?

3. The first task on your list should be initializing the 8259A. To start, determine the system base address for the 8259A. Refer to Hall: Figure 8-15 to check if you have determined the address correctly.
4. The 8259A has two internal addresses, which are determined by whether its A0 input is high or low. Note in Figure 21-1 that system address line A1 is connected to the A0 input of the 8259A. Determine the two system addresses that this will give the 8259A. Record these addresses, so that you will know where to send the command words.
5. The format for the 8259A command words is shown in Figure 22-1. Construct and write the ICW command words you need to send to the 8259A to initialize it in the following mode:

a. 8086 mode, single 8259A, edge triggered
b. Base interrupt of type 64 decimal
c. Nonspecial nested mode, nonbuffered mode, normal EOI

6. Use the A0 bit shown next to the control word formats in Figure 22-1 to determine the system address to which each of these control words must be sent. Then write the instructions needed to send these control words to the 8259A.
7. The next task in modifying the program is to write the instructions which initialize the interrupt vector table to point to the start of the clock interrupt procedure. Refer to Hall: Figure 8-31 to see how this is done. Specifically, look at the segment labeled AINT_TABLE and the related instructions in segment CODE_HERE. In your program, write the additions you need in AINT_TABLE and the instructions you need to initialize these locations.
8. The 8259A IR4 input must be enabled so that the 8259A will respond to an interrupt signal. This is done by sending an OCW1 command word to the 8259A. Figure 22-2 shows the formats for the 8259A OCWs. Write the OCW1 word needed to unmask the IR4 input and the instructions needed to send this word to the correct address for the 8259A.
9. The next task in your program modification is to determine the instruction needed to enable the 8086 INTR input so that the 8086 will respond to an interrupt request from the 8259A. Determine this instruction, and record it for future reference.
10. Make copies of the source files for the mainline and interrupt modules you developed in Experiment 21. For this experiment you will work with these copies, so that you will always have the working program from Experiment 21 as a backup.
11. Use Hall: Figure 8-31 and the answers you wrote previously to help you modify the copy of the mainline module as needed for this experiment. Note that the instructions which initialize the 8259A must be inserted in the program before the instructions which initialize the 8254. This is done so that the 8259A is initialized before the 8254 starts sending interrupt signals.
12. Assemble the mainline module and correct any errors.
13. Finally, at the end of your clock interrupt procedure, the 8259A must be told that you are done servicing this interrupt and that the 8259A is free to respond to other interrupts of the same or lower priority. If you don't do this, the 8259A will only respond to the first interrupt signal from the 8254 timer.

    This required end-of-interrupt operation is done by sending an OCW2 control word to the 8259A. For this program you want to send a nonspecific end-of-interrupt command word. Use the OCW2 format in Figure 22-2 to determine the OCW2 you must send to do this and the 8086 instructions necessary to send the word to the correct address.
14. Add these instructions to the copy of the source module for the interrupt procedure and assemble the module.
15. Link the .OBJ files for these two new modules with the .OBJ module for the display module. Note that the display module did not require any modification, so you can just link it into this program without any additional work.
16. Convert the .EXE file produced by the linker to a .BIN file, download the .BIN file to the SDK-86, and run it.
17. If the program does not work correctly, think about how you can determine if the problem is in hardware or software. To help, ask yourself how you would determine answers to the following questions.

a. Is the 1-kHz clock signal being produced, and does it reach the IR4 input of the 8259A?
b. Is the 8259A sending interrupt request signals to the INTR input of the 8086?
c. Is the interrupt-vector table getting properly initialized to point to the start of the interrupt procedure?
d. Is the 8086 INTR input getting enabled?
e. Is execution reaching the start of the interrupt procedure?

18. When you get your program running correctly,

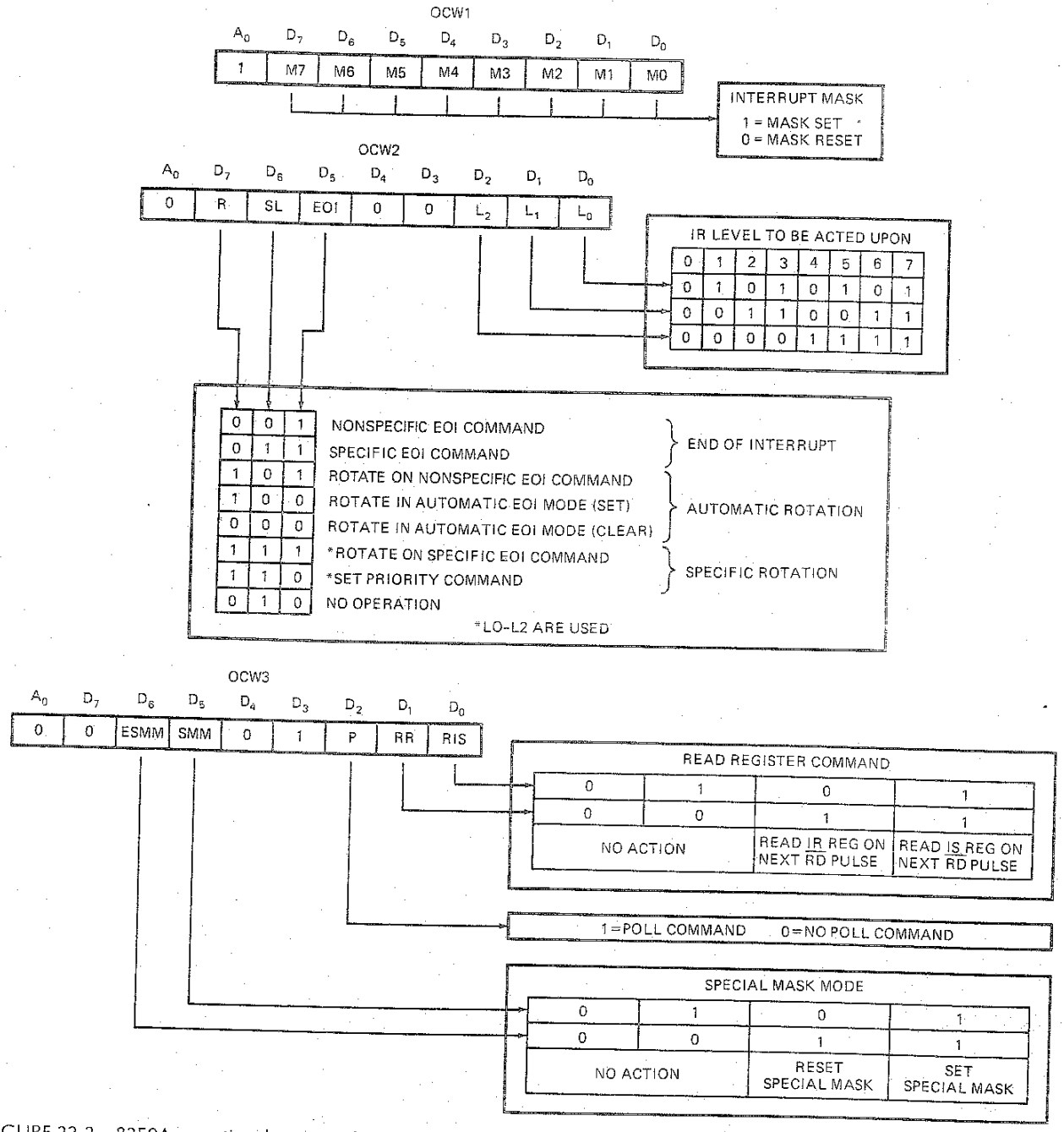


FIGURE 22-2 8259A operational command words. (*Intel Corporation*)

congratulate yourself on successfully writing a real program. As described in Chapter 11 of Hall, microcomputers such as the IBM PC use the same devices and program to produce a real-time clock which controls the timing of most operations in the system.

EXPERIMENT 23

Interfacing an Unencoded Keyboard to a Microcomputer

REFERENCES

Hall: Chapter 9

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

IBM PC or compatible computer

63-key unencoded, matrix-type keyboard

8×germanium or small-signal silicon diodes

16×4.7-kΩ resistors

74148 eight-line to three-line priority encoder

INTRODUCTION

A common method of entering programs, data, or commands into a microcomputer is with a keyboard. The keyboard may be a simple hex-type, such as that on the SDK-86, or it may be a typewriter-type keyboard. In either case, the keyboard is a matrix of switches arranged in columns and rows. To obtain meaningful data from a keyboard, you have to detect when a key is closed, debounce the key-pressed signal, and then produce the desired code for the pressed key (encode it).

In the hardware approach used in Experiment 9, scanning the keyboard matrix, debouncing, and encoding a keypress are done by a keyboard interface IC inside the keyboard. The advantage of this approach is that the processor is not tied up with the detect, debounce, and encode tasks. For many applications, however, the processor has plenty of time available. In that case, a software approach is used to eliminate the cost and PC board space of the hardware encoder.

In this experiment you will write a program that uses a procedure to detect, debounce, and encode a keypress from an unencoded keyboard. The most important goal of this experiment is to teach you the most effective method of developing a program which interacts with external hardware. The main points in this method are the following:

a. Develop the program in small, almost trivial, steps that are easy to debug. Your first tendency may be to write the entire program and then attempt to debug it. However, years of experience have shown us that the fastest route to a working program is one small step at a time.

b. Use software to output signals which stimulate the hardware; then use a scope to see if the hardware responds correctly.

c. Put known signals on the hardware and see if the software reads and interprets these signals correctly.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Systematically develop an application which involves both software and hardware.
2. Initialize an 8255A programmable port device.
3. Write and test an assembly language procedure to scan a matrix keyboard, detect a key pressed, debounce it, and read in the code for the key pressed.
4. Demonstrate the use of a look-up table and the XLAT instruction to convert a keypress code to ASCII.

PROCEDURE

1. Figure 23-1 shows the connections of an 8×8 matrix keyboard. The eight rows of the matrix are connected to output port lines. Initially lows are output on all these lines.

   The eight column lines are connected to the inputs of a 74148 priority encoder. Initially these lines are held high by the pull-up resistors to +5 V. However, if a key is pressed, the row and the column for that key will be connected, and that column will be pulled low. The 74148 puts out a 3-bit binary code which corresponds to the input pin that has a low on it. In other words, these outputs identify the column in which a key is pressed. The $\overline{GS}$ output of the 74148 is high when all the columns are high and low when any

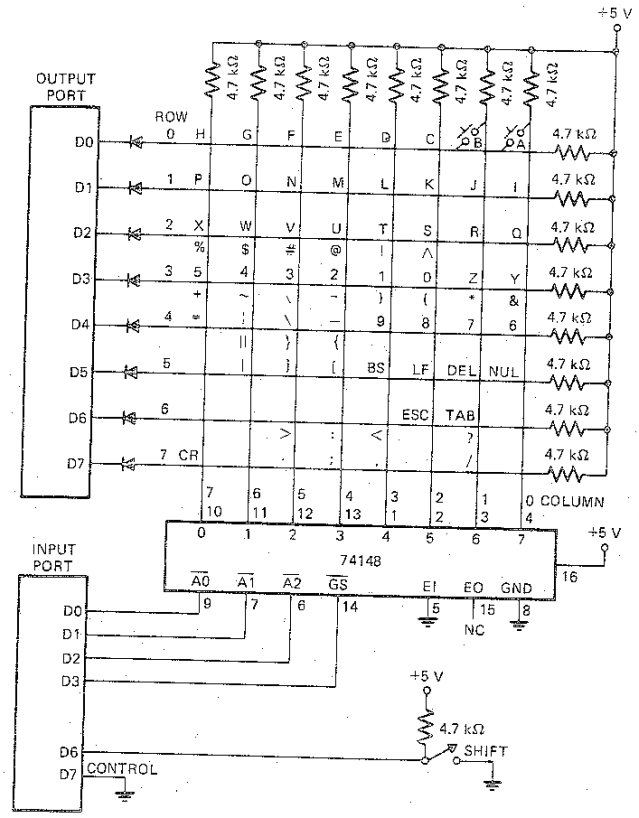


FIGURE 23-1 Unencoded keyboard circuit and connections.

column is low. Therefore, the $\overline{GS}$ output functions as a key-pressed strobe which can be polled to detect a key-press. Notice that the columns are connected to the 74148 in reverse order because the output code from the 74148 is inverted. If connected as shown, the 74148 will output 000 on $\overline{A2}$, $\overline{A1}$, and $\overline{A0}$ if a low is present on column 0.

2. Connect an unencoded keyboard to the input and output ports of an SDK-86 board, as shown in Figure 23-1. The rows should be connected to port P2A (FFF8H), and the outputs from the 74148 should be connected to port P2B (FFFAH). If a matrix keyboard is not available, build the circuit as shown. Use a jumper wire to temporarily connect a row to a column to simulate a key-press.
3. Write the mainline for the program which
   a. Sets up an array to hold the ASCII codes for at least 10 pressed keys.
   b. Initializes the segment registers.
   c. Initializes a stack pointer.
   d. Initializes port P2A (FFF8H) as an output port, mode 0, and port P2B (FFFAH) as an input port, mode 0. Note that the control word used to initialize these ports is different from the control word used in previous experiments. Use Hall: Chapter 9 to help you build this control word.
   e. Outputs a high on bit D0 and lows to the other bits of port P2A.
   f. Clears the AL register and inputs a byte from port P2B into it.
4. Assemble, run, and test this part of the program to see if you have initialized the SDK-86 ports correctly. Use an oscilloscope or multimeter to check if port P2A outputs a 1 on DO. Check the AL register to see if anything was input on port P2B. Remember, the $\overline{GS}$ output is high if no keys are pressed. If steps 3*e* and 3*f* did not work as expected, the control word you used to initialize the ports was incorrect. Return to step 3.
5. The test program put lows on all the rows of the key matrix except row 0. If a key is pressed and held in any other row, the $\overline{GS}$ output of the 74148 should be low and the inverted code for the column of the pressed key should be present on the A2-A0 outputs of the 74148. Use a scope to verify that the hardware works correctly for several keys.
6. Once the ports are behaving correctly, remove the port-checking code from the mainline, and continue.
7. Write the algorithm for a procedure that will first wait for all keys open, then wait for a key-press, wait 20 ms, and check again to see if the key is still pressed. If the key is still pressed, then the procedure should, for now, just return. The second check for a key-press is included to ensure that the first check did not detect just a noise pulse. Later you will add program sections to determine which row and column the pressed key is in and convert the row and column code to ASCII.
8. After your mainline program, add an assembly language procedure containing the instructions for this section of your algorithm. (To determine whether a key is pressed, you can simply output lows to all rows and poll the $\overline{GS}$ output of the 74148.) To your mainline program, add an instruction to call this procedure.
9. Assemble and run the combined program to a breakpoint after the call instruction in the mainline. The program should run to the breakpoint when you press a key. If the program does not work, use an oscilloscope to check that lows are getting to all the rows and that $\overline{GS}$ goes low when a key is pressed.
10. Once a keypress has been detected and debounced, the next step is to determine the row and column of the pressed key. The basic principle is to output a low to one row at a time and see if that low produces a low on the $\overline{GS}$ output. If it does, the pressed key is in that row. If it does not, the low is rotated to the next row and GS is checked again. To your detect and debounce algorithm, add a section which determines the row and column of a pressed key.

11. Add the assembly language instructions for this new section to your procedure.

    HINT: Use a register or memory location as a row counter. The row counter will contain the 3-bit code of the row that is outputting a 0.

12. Once you have the 3-bit code for the row containing the pressed key and the 3-bit code for the column containing the pressed key, you then want to combine these codes into a single byte which can be converted to ASCII.

    The format for the byte that contains the shift, row, and column codes (SRC word) should be as follows:

    D7—control (zero)
    D6—shift
    D5–D3—row code
    D2–D0—column code

    To your algorithm for the procedure, add a section which describes how you can make sure that bits D7, D5, D4, and D3 of the byte containing the column code are zero. Rotate the row-code register or memory location byte so that the row code is in bits D5-D3. Then combine the two bytes. The word produced will be used to access the correct key code in a look-up table.

13. Add the code for these steps to your procedure and test the program again. Test the program for more than one key to make sure that you obtain the correct SRC word for different rows and columns.

14. When your program produces the correct SRC word for a key-press, the next step is to convert the SRC code to ASCII. Read the discussion of the 8086 XLAT instruction in Hall: Chapter 6 to get an idea of how you will do this conversion. Here's an overview of how it works.

    BX is loaded with the offset of the start of a look-up table containing the ASCII codes for all of the keys on the keyboard. The SRC code for the pressed key is put in the AL register. When the XLAT instruction executes, the 8086 internally adds the SRC code in AL to the offset in BX. The sum then points to a location in the look-up table. XLAT transfers the byte from this location in the look-up table to AL.

    In order for this scheme to work, the desired ASCII code for a pressed key must be at a displacement in the table equal to its SRC value. For example, if the A key and the shift key are pressed together, the SRC word will be 0000 0000. The 0000 0000, when added to the memory pointer in BX, will point to the first memory location in the table. Since the ASCII code for a capital A is 41H, insert this value in the first memory location. To help you see this, ask yourself the following questions:

    a. If the shift key and the B key are pressed, what SRC word will be produced?
    b. What table location will this address?
    c. What ASCII code should be placed at that location? (Hall: Table 1-2 contains the complete 7-bit ASCII code.)
    d. If the shift key is not pressed and the A key is pressed, what SRC word will be produced, what address in the table will be pointed to, and what ASCII code should be placed in that address?

15. Make a look-up table for at least the upper- and lowercase letters, and add this to the data segment of your program.

16. To your algorithm for the procedure, add the steps necessary to do the SRC-to-ASCII conversion.

17. To your assembly language procedure, add the instructions to convert the SRC code into ASCII code. Pass the ASCII code back to the mainline in a register.

18. Run and test the program to verify that the correct ASCII codes are produced for 10 different keys.

19. In your report discuss the following:

    a. Bit 7 of the input port was left available for a control key input. What effect would including this input have on the size of the look-up table?
    b. How could this program be modified to give an EBCDIC output for each key?

20. (Optional) Use the display routine in Figure 20-1 to display the ASCII codes on the data field LEDs of the SDK-86 LED display.

    NOTE: If you developed this display module in Experiments 20–22, you can just modify your program to call this procedure, add an EXTRN statement to tell the assembler that the procedure is in another module, and link the .OBJ file for the display procedure with the .OBJ file for your keyboard program. This demonstrates one major advantage of writing programs in modules.

EXPERIMENT 24

Speech Synthesis and Handshake Data Output

REFERENCES

Hall: Chapter 9

Hall: Chapter 13

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

Small 8-Ω speaker

DT1050 Digitalker Standard Vocabulary Kit

LM741, LM386 amplifiers

4×1N4001 diodes

4-MHz crystal

6×0.1-μF capacitors

20-pF, 50-pF, 0.05-μF, 20-μF capacitors

50-kΩ potentiometer

10-Ω, 1.5-kΩ, 9.1-kΩ, 10-kΩ, 1-MΩ resistors

Optional (parts for extra vocabulary circuit additions)

DT1057 Digitalker Added Vocabulary ROMs

74LS138, 74LS04

2×0.1-μF capacitors

INTRODUCTION

One simple method of producing speech under microcomputer control is with a National Semiconductor Digitalker. As explained in Hall: Chapter 13, the Digitalker uses data from actual recorded human speech to produce words and phrases. To summarize how this method works, a word or phrase is digitized with an A/D converter. The samples produced by the A/D converter are processed by a computer program to reduce the amount of memory required to store the data for the given word or phrase. This "compressed" data is then stored in ROMs. When you tell it to do so, a specialized speech processor, the MM54104, reads the compressed data from ROM and uses it to reconstruct the speech waveform for the desired word or phrase. Figure 24-1 shows how a Digitalker speech processor system can be built and interfaced to SDK-86 ports.

Figure 24-2*a* shows the words that the Digitalker can "speak" with the basic vocabulary data stored in its SSR1 and SSR2 ROMs. To get the Digitalker to speak a word or phrase, you simply send it the 8-bit code which corresponds to that phrase, shown in Figure 24-2*a*, and pulse its $\overline{WR}$ input low. When the Digitalker finishes speaking the specified word(s), it will assert its INTR output high. This tells you that you can then send the 8-bit code for the next word or phrase. Because the Digitalker takes different amounts of time to speak different words, the easiest way to send words to it is on an interrupt-driven handshake basis. The 8255A $\overline{OBF}$ unfortunately cannot be used as part of the handshake, because the 8255A $\overline{OBF}$ will not go low until the Digitalker's INTR goes high, and the Digitalker's INTR will not go high until the 8255A's $\overline{OBF}$ goes low. This could be called a hardware deadlock situation. To overcome this difficulty, the Digitalker's $\overline{WR}$ pin is connected to the P1C5 output pin on the 8255A so that $\overline{WR}$ can be generated directly by program instructions. P1C5 is initialized high in the mainline. Then, when a speech code is sent to the Digitalker, P1C5 is made low and then high by program instructions.

The goal of this experiment is to teach you to use an 8255A for handshake data output, give you further practice with interrupts, and introduce you to some common speech-synthesis devices.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Initialize a specified port in an 8255A for operation in handshake output mode.
2. Write an interrupt procedure which reads a speech code from a table in memory and sends it to the 8255A on an interrupt-driven handshake basis.

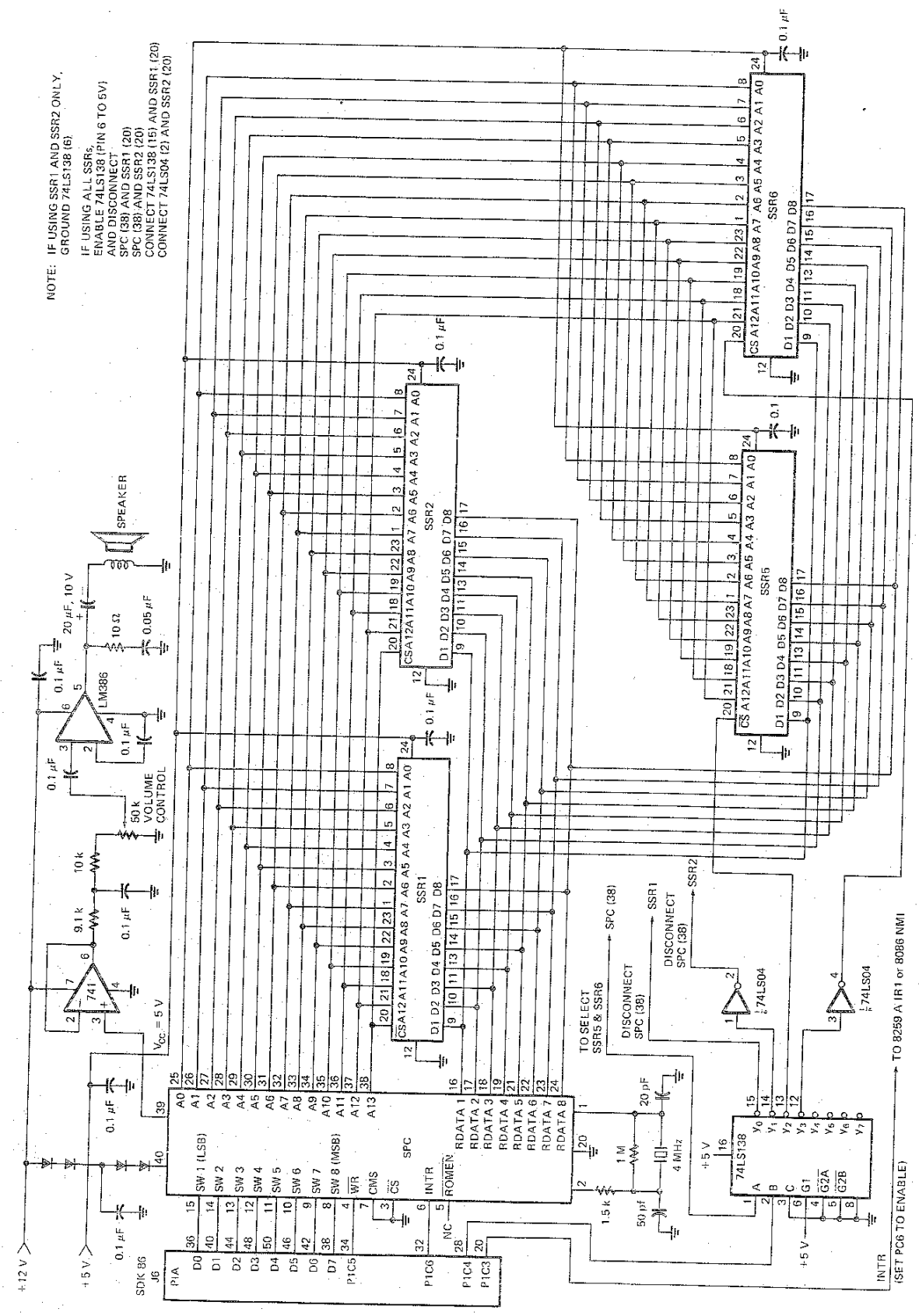


FIGURE 24-1 Circuit diagram for Digitalker speech processor.

PROCEDURE

Digitalker Speech Synthesizer

1. If it is not already built, build the circuit for the Digitalker shown in Figure 24-1. The data and signal inputs for the Digitalker are connected to the P1 port (connector J6, P1A address FFF9H) on the SDK-86 board. The handshake interrupt signal, P1C3, produced by the 8255A can be connected to the IR1 input of the 8259A priority-interrupt controller. The Digitalker $\overline{WR}$ input is connected to P1C5.

   NOTE 1: If you have not added an 8259A to your SDK-86 board, you can use the 8086 NMI interrupt input, as you did in Experiment 20.

   NOTE 2: The connection to the IR1 input of the 8259A should be made as short as possible so electrical noise does not cause unwanted interrupts.

2. The program for this experiment will consist of two parts:

   a. A mainline, which initializes everything and waits for interrupts.
   b. An interrupt procedure, which reads the next speech code from a table in memory and sends it to the 8255A for output to the Digitalker.

3. Write a list of tasks that you need to do in the mainline and then develop the instructions for each task. Here are some hints to help you.

   a. Don't forget to set up an interrupt-vector table, initialize the interrupt-vector table, and initialize the 8259A interrupt controller.
      Refer to Hall: Figure 8-31 to refresh your memory about how to initialize an 8259A and an interrupt-vector table for operation with an 8259A. Note that you have to slightly modify the instructions shown in Hall: Figure 8-31 for use here because you are using only the IR1 input of the 8259A.
   b. Set up a data table which contains a series of word codes from Figure 24-2*a* for the words you want the Digitalker to speak. Put a sentinel character of FFH as the last character in the table.
   c. Initialize the 8255A. Hall: Figure 9-5 shows the formats for the 8255A control words. Use this figure to help you make up the control word which initializes the 8255A P1A port in the needed handshake mode and P1C as output.
   d. Send the 8255A control words to the correct system address.
   e. Use Hall: Figures 9-5 and 9-9 to help you make up the bit set-reset control word that must be sent to the 8255A to enable the handshake interrupt output signal produced on the P1C3 pin.
   f. Make up the bit-set instruction to set P1C5 high, so that the $\overline{WR}$ signal starts off high.
   g. Enable the 8086 INTR input.
   h. Include PUBLIC and EXTRN statements as needed so that this module will properly connect to the interrupt procedure module.

4. After you have written and checked the instructions for the mainline module, assemble the module and correct any errors.
5. Next write the algorithm for the interrupt procedure. This interrupt procedure should do the following:

   a. Read a word code from the table and, if the character is not the sentinel character, send it to the 8255A for output to the Digitalker.
      For the Digitalker, the $\overline{OBF}$ cannot easily be made to work correctly. Therefore, you will have to write a speech code and then generate a $\overline{WR}$ signal to the Digitalker. You do this by first resetting and then setting the P1C5 line that is connected to the Digitalker's $\overline{WR}$ input.
   b. Perform the end-of-interrupt operation and return to the mainline.
   c. If the character read is the sentinel character, the procedure should mask the IR1 input of the 8259A to prevent further interrupts, perform the end-of-interrupt operation, and then return to the mainline.

6. Write the source program for the interrupt procedure module. Don't forget to include any required PUBLIC and EXTRN statements so this module will connect properly with the mainline.
7. Assemble the interrupt procedure module and then link the object files for the two program modules. Use EXE2BIN to convert the .EXE file to a .BIN file, download the .BIN file to an SDK-86 board, and then run the program.
8. The Digitalker should produce the words represented by the word codes you put in your data table. If it doesn't, this is a good chance to test your debugging skills. To help you, here are a few points to think about.

   a. How can you determine if execution ever gets to the interrupt-service procedure?
   b. If execution never gets to the interrupt procedure, how can you determine if the 8255A ever sends an interrupt signal to the 8259A?
   c. How can you determine if the 8259A sends an interrupt signal to the 8086?
   d. How can you determine if the 8086 responds to the interrupt signal from the 8259A?
   e. If execution gets to the interrupt procedure, how can you determine if codes are being output from the 8255A to the synthesizer?

9. (Optional) You can increase the Digitalker's vocabulary by adding the SSR5 ROM, SSR6 ROM, a 74LS138, and a 74LS04, as shown in Figure 24-1. Note that if this circuitry is added, the A13 line from the SPC must be disconnected from the $\overline{CS}$ input of SSR1 and the $\overline{CS}$ input of SSR2. These inputs are then connected to the outputs of the

The number shows the address (in hex) of the word that follows

Words produced using the SSR1 and SSR2 speech ROMS

00	This is digitalker						
01	one	31	R	61	it		
02	two	32	S	62	kilo		
03	three	33	T	63	left		
04	four	34	U	64	less		
05	five	35	V	65	lesser		
06	six	36	W	66	limit		
07	seven	37	X	67	low		
08	eight	38	Y	68	lower		
09	nine	39	Z	69	mark		
0A	ten	3A	again	6A	meter		
0B	eleven	3B	ampere	6B	mile		
0C	twelve	3C	and	6C	milli		
0D	thirteen	3D	at	6D	minus		
0E	fourteen	3E	cancel	6E	minute		
0F	fifteen	3F	case	6F	near		
10	sixteen	40	cent	70	number		
11	seventeen	41	400-Hz tone	71	of		
12	eighteen	42	80-Hz tone	72	off		
13	nineteen	43	20 ms silence	73	on		
14	twenty	44	40 ms silence	74	out		
15	thirty	45	80 ms silence	75	over		
16	forty	46	160 ms silence	76	parenthesis		
17	fifty	47	320 ms silence	77	percent		
18	sixty	48	centi	78	please		
19	seventy	49	check	79	plus		
1A	eighty	4A	comma	7A	point		
1B	ninety	4B	control	7B	pound		
1C	hundred	4C	danger	7C	pulses		
1D	thousand	4D	degree	7D	rate		
1E	million	4E	dollar	7E	re		
1F	zero	4F	down	7F	ready		
20	~A	50	equal	80	right		
21	B	51	error	81	ss		
22	C	52	feet	82	second		
23	D	53	flow	83	set		
24	E	54	fuel	84	space		
25	F	55	gallon	85	speed		
26	G	56	go	86	star		
27	H	57	gram	87	start		
28	I	58	great	88	stop		
29	J	59	greater	89	than		
2A	K	5A	have	8A	the		
2B	L	5B	high	8B	time		
2C	M	5C	higher	8C	try		
2D	N	5D	hour	8D	up		
2E	O	5E	in	8E	volt		
2F	P	5F	inches	8F	weight		
30	Q	60	is				

(a)

Words produced using the SSR5 and SSR6 speech ROMS.

00	abort				
01	add	31	first	61	record
02	adjust	32	floor	62	replace
03	alarm	33	forward	63	reverse
04	alert	34	from	64	room
05	all	35	gas	65	safe
06	ask	36	get	66	secure
07	assistance	37	going	67	select
08	attention	38	half	68	send
09	brake	39	hello	69	service
0A	button	3A	help	6A	side
0B	buy	3B	hertz	6B	slow
0C	call	3C	hold	6C	slower
0D	caution	3D	incorrect	6D	smoke
0E	change	3E	increase	6E	south
0F	circuit	3F	intruder	6F	station
10	clear	40	just	70	switch
11	close	41	key	71	system
12	complete	42	level	72	test
13	connect	43	load	73	th
14	continue	44	lock	74	thank
15	copy	45	meg	75	third
16	correct	46	mega	76	this
17	date	47	micro	77	total
18	day	48	more	78	turn
19	decrease	49	move	79	use
1A	deposit	4A	nano	7A	uth
1B	dial	4B	need	7B	waiting
1C	divide	4C	next	7C	warning
1D	door	4D	no	7D	water
1E	east	4E	normal	7E	west
1F	ed	4F	north	7F	switch
20	ed	50	not	80	window
21	ed	51	notice	81	yes
22	ed	52	ohms	82	zone
23	emergency	53	onward		
24	end	54	open		
25	enter	55	operator		
26	entry	56	or		
27	er	57	pass		
28	evacuate	58	per		
29	exit	59	pico		
2A	fail	5A	place		
2B	failure	5B	press		
2C	farad	5C	pressure		
2D	fast	5D	quarter		
2E	faster	5E	range		
2F	fifth	5F	reach		
30	fire	60	receive		

(b)

FIGURE 24-2 Code Words for Digitalker. (a) Words produced using the SSR1 and SSR2 speech ROMs (b) Optional words set using the SSR5 and SSR6 speech ROMs.

74LS138 decoder circuitry as shown, and A13 from the SPC is connected to the A input (pin 1) of the 74LS138.

A low on the B input of the decoder circuitry will select a specified word in one of the basic-vocabulary ROMs, SSR1 or SSR2. A high on the B input of the 74LS138 decoder will select a specified word in the extended-vocabulary ROMs, SSR5 or SSR6. This input is connected to port P1C4. If this pin is initialized for output, you can assert this line high or low with a bit set-reset control word. Figure 24-2*b* shows the additional words contained in these ROMs and the 8-bit codes you have to send for each word.

10. (Optional) If you would like to have your computer talk to you, you can add the Digitalker circuit and program from this experiment to the clock from Experiment 22, so that the time is announced to you verbally every minute or every hour.

EXPERIMENT 25

Lighting an LED Matrix—Timed Output and Multiplexing

REFERENCES

Hall: Chapter 9

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

IBM PC-compatible computer

8×8 LED matrix with driver transistors

Parts for matrix:

64×LEDs, minimum 25-mA capacity

64×150-Ω, 0.25-W resistors

8×2N2907 transistors

8×2N3904 transistors

8×2N2222 transistors

16×1-kΩ resistors

8×10-kΩ resistors

8×2.2-kΩ resistors

piece of 100-mil vector board, 4 by 6 in.

INTRODUCTION

The LED matrix shown in Figure 25-1 is made up of horizontal rows and vertical columns. To light an LED in the matrix, both the row-driver transistor supplying the current to its anode and the column-driver transistor sinking the current from its cathode must be on. Notice that the rows of the matrix are connected to one port, and the columns of the matrix are connected to another port. To light a pattern on the LEDs in a row, send to the column port (port B) a pattern of 1's and 0's that corresponds to the LEDs you want lit in that row. Then send a pattern of 1's and 0's to the row port (port A) to turn on that row. If you turn on row 1, for example, the pattern sent to the column port will be displayed on row 1 of the LED matrix.

By multiplexing you can display any pattern you want on the entire LED matrix. To do this, you first draw the design and convert it into 1's and 0's. The bit patterns for the eight rows are stored in a table in memory. To produce a display, the bit pattern for one row of the desired display is read from the table and output to the column port. A bit pattern which lights that row is then output to the row port. After a few milliseconds the pattern for the next row is output to the column port and that row is turned on. The process is continued until all rows have had a turn. If the entire sequence is repeated over and over quickly enough, it will appear that the matrix is displaying a complete design.

To make a more spectacular display you can store the patterns for several complete designs and cycle through these in order. You might, for example, display the letters of your name in order. An alternative is to make an expanding (exploding) box display or a scrolling display. Each design is displayed for an appropriate time and then the next design is displayed.

The display will be updated on an interrupt basis using an 8254A programmable timer and an 8259A priority interrupt controller.

NOTE: If your SDK-86 board does not have an 8254A and an 8259A, you can use delay loops to control the timing of the display refresh.

The major goals of this experiment are to give you more practice working with an 8254 programmable timer, working with an 8259A priority-interrupt controller, and developing a program which interacts with external hardware.

OBJECTIVES

At the end of this experiment, you should be able to

1. Use an 8254A programmable timer and an 8259A priority-interrupt controller to produce interrupts at desired intervals.
2. Display any desired pattern on an 8×8 LED matrix using a software multiplexing method.

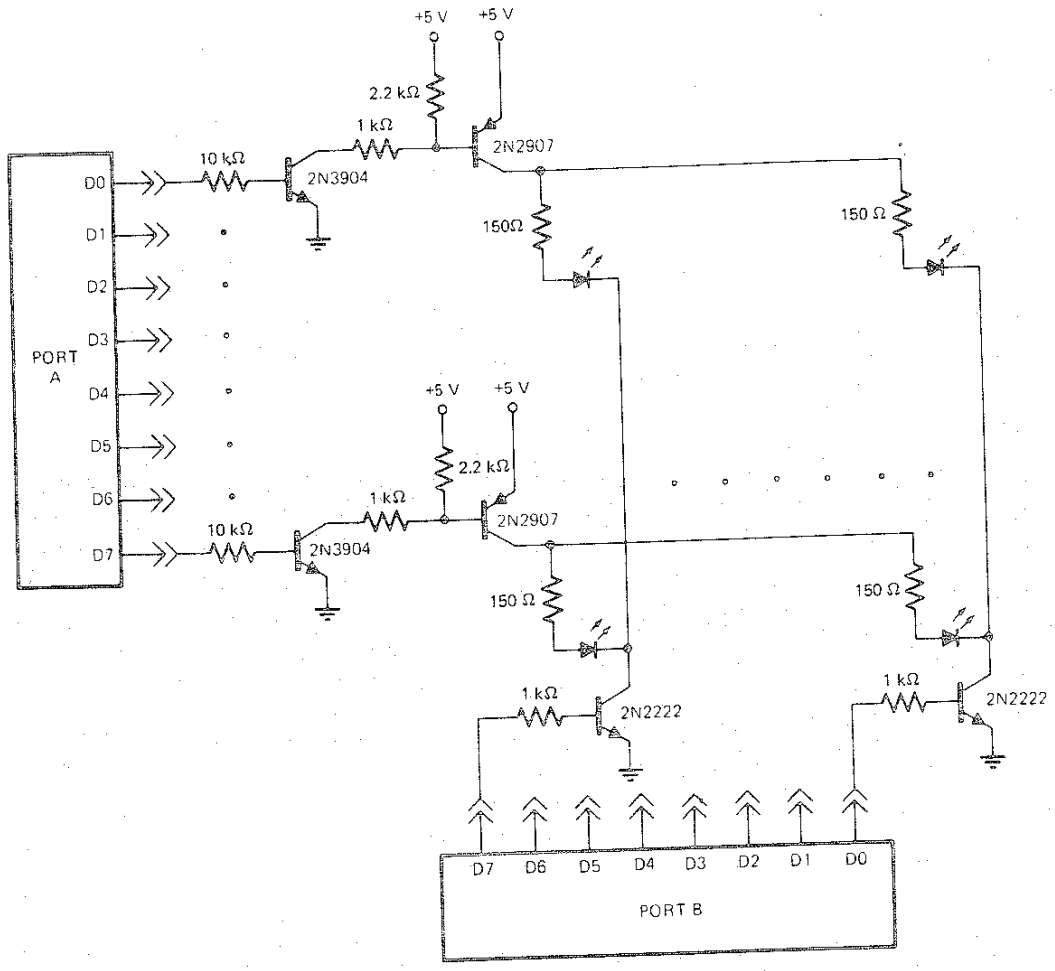


FIGURE 25-1 A schematic for an 8×8 LED matrix with driver transistors.

PROCEDURE

1. The first step is to write a simple program which initializes the port devices and tests the display module. Study Figure 25-1 and answer the following questions.
   a. What logic level is required on a bit of port A to turn on a row?
   b. What logic level is required on a bit of port B to turn on a column?
   c. What word should be output to port A and what word should be output to port B to light the entire top row of LEDs?
   d. What word should be output to port A and what word should be output to port B to light the entire leftmost column of LEDs?
2. Write an assembly language program which does the following:
   a. Initializes ports P1A and P1B on the SDK-86 board as output ports.
   b. Outputs the row and column patterns needed to turn on the LED in the top left corner of the matrix.
3. Connect the output ports to the matrix, run your program, and observe the results.
4. Once you verify the patterns needed to light one LED, change your program so that all the LEDs in the first row are lit. Run your program again.
5. To test the entire LED matrix, modify the program so that it lights a row, delays a fraction of a second, lights the next row, etc., until all rows have had a turn and then repeats. (For the timing here you can use the simple nested delay loop from Experiment 13.)
6. Now that you have tested and understand the operation of the hardware, write an algorithm for the mainline which initializes everything.
7. Write an algorithm for the interrupt procedure which refreshes the display every 4 ms. For now assume that you only want to display the letter X and that the patterns for the eight rows of the display will be kept in a table in memory.
8. Write the mainline program which initializes the 8255A ports, the 8259A, the 8254A, the interrupt vector table, etc.

NOTE: If you did Experiment 22, you can just make a copy of the mainline from it and modify

the timer count, etc., as needed for this project. If you did not do Experiment 22, use Hall: Figure 8-31 as a template to develop this module.

9. Assemble, link, convert, and download this module. Disconnect the timer output from the 8259A input and run the program. Use a scope to verify that the 8254A is generating pulses with the correct timing.
10. Write the assembly module which contains the procedure to update the display. Again, if you did Experiment 22, you can use a copy of the interrupt procedure module from it as a "skeleton" for this module. If you did not do Experiment 22, use the clock procedure in Hall: Figure 8-31 to help you write this module.
11. Assemble this module, link it with the mainline module, convert the result to a .BIN file, and download the .BIN file.
12. Connect the 8254A timer output to the appropriate input on the 8259A and run the program. You should see your design displayed on the LEDs. If not, use breakpoints to help you debug the program.
13. (Optional) Change the display refresh procedure so that it displays an X for 1 s, displays a 0 for 1 s, and then repeats. To do this, add a counter which keeps track of how many times the display has been refreshed. After a number of refresh times that corresponds to 1 s, switch the pointer to a table of values for the next character to be displayed. After that character has been displayed for 1 s, switch back to the table for the first character.
14. (Optional) Add the data tables and modify the procedure so that the display counts down from 9 to 0 with each digit being displayed for 1 s.

EXPERIMENT 26

Analyzing the SDK-86 Keyboard and Display Circuitry Operation

REFERENCES

Hall: Chapters 7 and 9

EQUIPMENT AND MATERIALS

SDK-86 board and power supply

16-pin and 40-pin IC glomper clips

Logic analyzer or dual-channel oscilloscope

INTRODUCTION

With a little help from two decoder chips and some buffer transistors, an 8279 refreshes the LED displays and scans the hex keypad on the SDK-86 board. The purpose of this experiment is to familiarize you with the waveforms typically found in keyboard scan and display refresh circuitry, so that you can recognize incorrect waveforms in a system that is not working correctly. In a troubleshooting situation you will usually use an oscilloscope to look at waveforms, but for your initial look at these waveforms, a logic analyzer gives a more complete picture. If you don't have a logic analyzer available, you can construct the composite waveforms one line at a time with a dual-trace oscilloscope.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Initialize an 8279 keyboard/display controller IC for a specified mode of operation.
2. Draw the waveforms for the signals which make sure that only one digit of a multiplexed display is turned on at a time.
3. Draw the waveforms found on the segment bus of a multiplexed display as known values are being output to the digits.
4. Explain the function of each set of states found on the segment bus as LED digits are refreshed.
5. Draw the waveforms found on the keyboard circuitry when a key is pressed.

PROCEDURE

1. The first step here is to develop a short test program which initializes the 8279 in a known mode and sends recognizable seven-segment codes to its display RAM for display. The control register address for the 8279 on the SDK-86 board is FFEAH, so this is where you will send control words to initialize the device and to tell it that you want to write to the display RAM. To initialize the 8279 you have to send it a Keyboard/Display Mode Set control word, a Program Clock control word, and a Clear Display control word.

   With the help of Hall: Figure 9-29, construct these three control words as needed to initialize an 8279 for the following mode of operation:

   8-bit character display—left entry
   Encoded scan keyboard—N-Key Rollover
   Program clock, divide by 24
   Blanking character, all zeros

2. Write the 8086 instructions necessary to send these three control words to the 8279. To these instructions add the instructions necessary to display the digits 1234 on the address field LEDs and the digits 5678 on the data field LEDs. (You can use the display module from Experiment 20, Figure 20-1, to do this). The test program can simply sit in an endless loop after sending the desired seven-segment codes to the 8279.
3. Run the test program and make sure it displays the desired digits correctly. With the test program running, you are now ready to analyze and observe the operation of the circuitry.
4. Analyze the circuitry for the SDK-86 LEDs in Hall: Figure 7-8, sheet 8, to determine the logic level that is required to turn on one of the digit-driver transistors. Write your prediction for this logic level.
5. As you can see in Hall: Figure 7-8, sheet 7, the signals which drive these transistors are produced by a 7445 decoder. The input signals for the 7445 come from the Scan Line outputs (SL0–SL3) of the 8279. To see the signals present on these lines, connect four channels of your logic analyzer to the SL0–SL3 outputs of the 8279. Set the analyzer for

internal clock, trigger on XX, and timing display. Do a trace and observe the displayed waveforms. Experiment with the internal clock period until you get one sequence of the waveforms displayed; then draw and label these waveforms for your report.

NOTE: If you do not have a logic analyzer, connect one scope probe to the SL0 output and the other scope probe to the SL1 output. Draw a section of the observed waveforms. Move the second scope probe to the SL2 output and add the waveform you see there to those you drew for SL0 and SL1. You should see that these waveforms represent a continuous binary count sequence from 0 to 7.

6. To see the waveforms that this binary count produces on the outputs of the 7445, connect eight data inputs of the analyzer to the 0 outputs of the 7445 in order. Set the analyzer for internal clock, trigger on FEH, and timing diagram display. Make traces and adjust the clock timing until the display shows one or two cycles for each signal. Draw the waveforms you see.

NOTE: If you do not have a logic analyzer, connect one input of a scope to the 00 output of the 7445 and the other input of the scope to the 01 output of the 7445. Draw a section of the waveforms you see. Move the second scope probe to look at each of the other 7445 outputs one at a time, and add these waveforms to your drawing.

Use your drawing to answer the following questions in your report.

a. Which digit gets turned on first in the refresh sequence?
b. Are two digits ever turned on at the same time?
c. How often is each digit turned on?
d. What is the multiplex rate? (Use the answer to *c* to calculate this.)
e. For how long is each digit turned on during a refresh cycle?

7. The next step is to observe the waveforms on the segment outputs of the 8279 as the LED displays are refreshed.

Connect the eight data inputs of your logic analyzer to the A3–B0 outputs of the 8279 in order, starting with the B0 output connected to the least significant digit input of the logic analyzer. Set the analyzer for internal clock, trigger on XX, and timing diagram display. Do a trace and observe the waveforms produced. Again, you may have to adjust the analyzer clock frequency and do another trace to get a display of the refresh operation for all eight digits. Switch to the state mode display to help you see the actual sequence of codes that are output on these lines. Write the sequence of codes. (If a code appears for more than one sample time, you only have to write it once.) In your report explain the function of the codes that are present between the actual seven-segment codes. If you need help with this, refer to Hall: Figure 9-27.

NOTE: If you don't have a logic analyzer, you can develop the waveforms on these signal lines the same way you did those for the 7445 outputs. Connect one scope probe to the B0 output of the 8279 and the other scope probe to the B1 output of the 8279. Adjust the scope time base to get a display of one refresh cycle for the eight digits. To help you identify one refresh cycle in the displayed waveforms, first remember that the "a" segment is connected to the B0 output of the 8279. Also, note that, starting from the right, the "a" segment is on for digits 8, 7, 6, and 5; off for digit 4; on for digits 3 and 2; and off for digit 1. Draw the waveforms you see. Then move the second scope probe to the B2 output and add the waveform from it to your drawing. Continue until you get all eight waveforms. Note that waveforms all start with a high; this is the seven-segment code for "8." The waveforms are low for a very short time between each of the seven-segment codes being output. Explain the purpose of these short low periods. Refer to Hall: Figure 9-27 if you need help with this. Identify the rest of the seven-segment codes being output.

8. The next circuitry to analyze is the keyboard scan section. Study this section of the circuitry on sheet 7 of the SDK schematics in Hall: Figure 7-8. A 74LS156 one-of-eight-low decoder converts the binary counts output by the 8279 SL0-3 lines to the signal patterns needed to scan the keyboard. The decoder outputs a low on the output that corresponds to the input binary code. When a key is pressed, the row containing that key will be connected to one of the Return inputs of the 8279. Therefore, that Return line will be pulled low. The 8279 Return inputs are normally held high by internal pull-up resistors.

Connect three data inputs of the logic analyzer to the 2Y0-2Y2 outputs of the 74LS156 decoder in order. Set the analyzer for internal clock, trigger on XX, and timing mode. Do a trace and observe the waveforms. In your report explain why no pulses are observed on the 74LS156 outputs.

HINT: The 74LS156 has open collector outputs.

Hold down the EB key on the SDK-86 keypad and again do a trace of the waveforms on the 2Y0–2Y2 lines. In your report explain why pulses are now seen on the 2Y0 output. Also, determine how often a low is put on each row to see if a key is pressed. In your report draw and label a section of the observed waveforms.

NOTE: If you do not have a logic analyzer, use a scope to observe and measure the timing of the waveform on the 2Y0 line when the EB key is held down. Draw and label this waveform in your report.

EXPERIMENT 27

Stepper Motors

REFERENCES

Hall: Chapter 9

EQUIPMENT AND MATERIALS

SDK-86 board

Power supply, 5-V and 12- or 15-V with 2-A capability

IBM PC-compatible computer

Digital voltmeter

Four-phase stepper motor such as Superior Electric MO61-FD302 or IMC Magnetics Corp. Tormax 200

Stepper-motor power driver board (see Figure 27-1*a*) parts:

7406 hex inverter

4×MJE2955 power transistors

4×IN4002 diodes

4×1-kΩ resistors

4×470-Ω resistors

2×8-Ω, 10-W resistors

INTRODUCTION

As described in Hall: Chapter 9, stepper motors rotate, or "step," from one fixed position to another instead of spinning smoothly, as most motors do. Many of these motors have four independent coils, as shown in Figure 27-1. At any given time, current is flowing through only one or two of them. The motor is stepped by changing which coil or coils have current flowing through them.

Figure 27-1*a* shows a high-current buffer which can drive a four-coil (phase) stepper motor. The inputs to this buffer are simple TTL logic signals which can be supplied by a microcomputer output port or some other digital device.

Figure 27-1*b* shows the sequences of switch patterns which will cause the motor to take full steps clockwise or counterclockwise. These switching sequences can be produced by several means, ranging from purely hardware to mostly software. In this experiment you will generate the stepping sequence with software.

If the four inputs of the stepper motor driver board are connected to the lower 4 bits of an output port as shown in Figure 27-1, the motor can be stepped by outputting a pattern to the motor, waiting a specified time, and then outputting the next pattern in the switch sequence. If you look closely at Figure 27-1*b*, you should see that to step in a clockwise direction, the pattern of 1's and 0's is simply rotated 1 bit position around to the right, then output. To step in a counterclockwise direction, the pattern is rotated 1 bit position around to the left and then output.

The program to generate a stepping sequence is very simple, but there is one problem. If the microcomputer outputs a sequence of steps faster than the motor can respond, the motor will skip steps. To solve this problem a delay must be inserted between steps. The delay must be at least a few milliseconds for commonly available motors. If your SDK-86 board has an added 8254A programmable timer and an 8259A priority-interrupt controller, you can generate the delays with timed interrupts. If you do not have the added interrupt capability, you can use a delay loop to produce the required delay.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write and test an assembly language program to control the number of steps moved, the direction of rotation, and the stepping speed of a stepper motor.

PROCEDURE

1. Assuming you are using interrupts to control the step timing, the program for this experiment will consist of two modules. One is a mainline which sets up the interrupt-vector table; initializes the

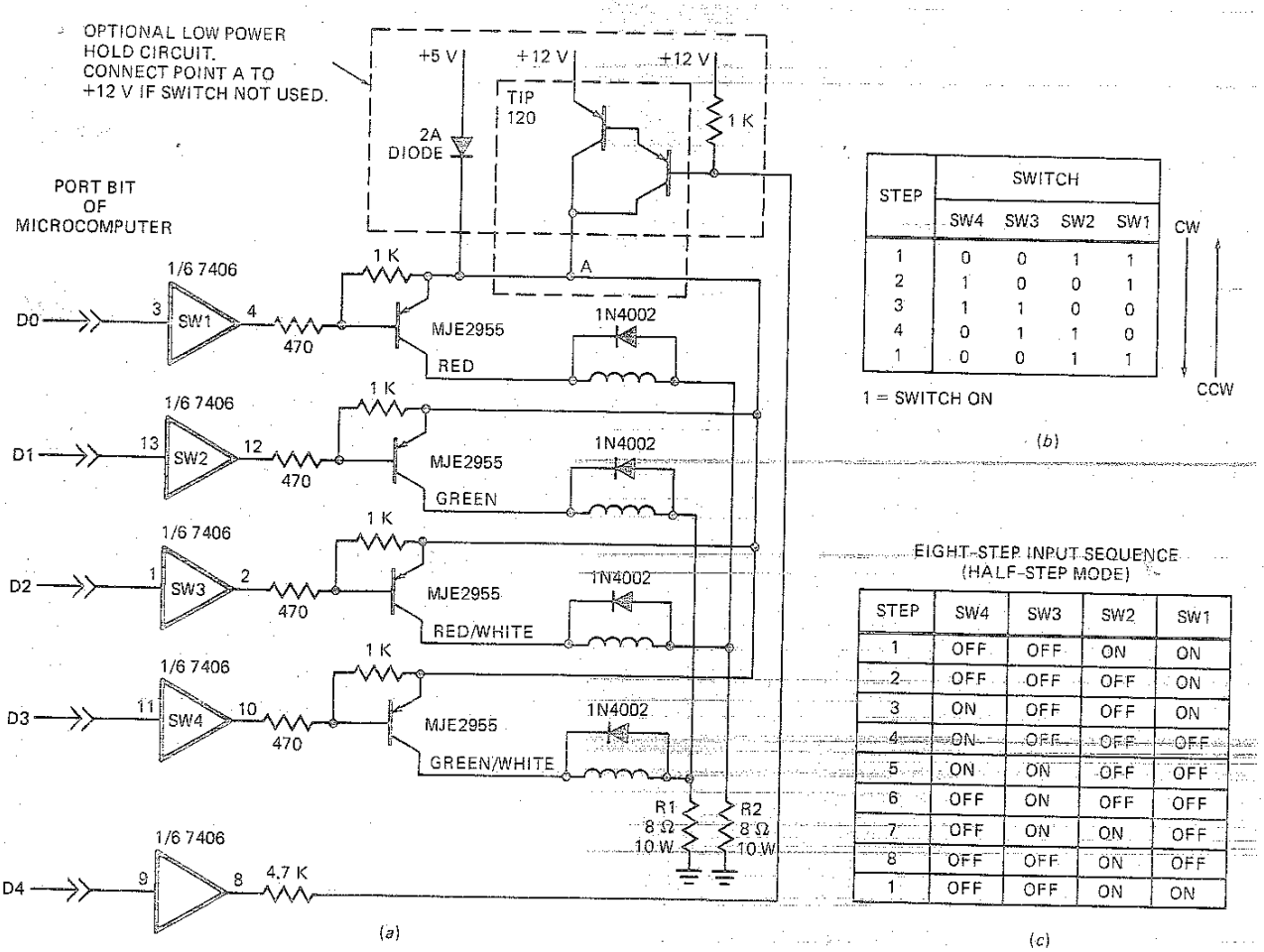


FIGURE 27-1 (*a*) Schematic of a stepper motor power-driver board. (*b*) Switch sequence for a stepper motor. (*c*) Eight-step input sequence.

port, timer, and PIC; and waits for an interrupt. The second module is an interrupt procedure which is called on each timer interrupt. If the desired time between steps has elapsed, this procedure outputs the pattern for the next step and returns to the mainline to wait for the next timer interrupt. Write an algorithm for the mainline module of this program. Include in this algorithm a list of all the required initializations and a mechanism to pass the step direction, the number of steps, and the number of milliseconds between steps to the interrupt procedure. Initially you can pass just one set of values.

2. Write the assembly language program for the mainline module. (You can use a copy of the mainline module you wrote for Experiment 22, a copy of the mainline module from Experiment 25, or Hall: Figure 8-31 to help you write this module.)

NOTE: A good starting value for the time between steps is 0.5 to 1 s.

3. Assemble the mainline module and correct any assembly errors.
4. Write an algorithm for the interrupt procedure which counts interrupts and at the appropriate time steps the motor one step in the specified direction. When the desired number of steps have been taken, the procedure should disable the timer interrupt before returning to the mainline.
5. Write the source program for the interrupt module. (You can use a copy of the interrupt module from Experiment 22, a copy of the interrupt module from Experiment 25, or Hall: Figure 8-31 to help you write this module.
6. Assemble the interrupt module and correct any errors.
7. Link the .OBJ files for the mainline and interrupt modules, convert the resulting .EXE file to a .BIN file, download the .BIN file to the SDK-86, and run the program.
8. If the program does not work, use a scope to check if the timer output is correct and if any pulses are present on the buffer inputs.
9. If the timer output is correct but there are no

pulses on the buffer inputs, use breakpoints to troubleshoot the program.

10. Many applications require a stepper motor to go through a series of moves, stepping first in one direction and then in the other. The direction, number of steps, and delay between steps for each sequence can be stored in an array in memory and a pointer used to access the parameters for the sequences in order.

    Modify your program modules so that the motor will step 5 steps clockwise, 10 steps counterclockwise, and then 100 steps clockwise.

11. Assemble, link, convert, download, test, and, if necessary, debug the new program.

12. The next step in developing this program is to experiment with faster stepping speeds. The delay values are stored in memory locations, so you can run the program with one set of delay values, observe the result, and then load new delay values and run the program again. Use this approach to determine the maximum stepping rate for your motor. (You may want to increase the number of steps in each sequence to make the motor's movement easier to observe.)

13. (Optional) For finer resolution, the half-step sequences shown in Figure 27-1c can be used instead of the full-step sequence shown in Figure 27-1*b*. Modify your program so that it uses the half step sequences.

HINT: Use a look-up table instead of the simple rotate technique.

EXPERIMENT 28

Using a D/A Converter with a Microcomputer

REFERENCES

Hall: Chapter 10

EQUIPMENT AND MATERIALS

SDK-86 board

Power supply with ±5-V and ±12-V outputs

Digital voltmeter

Oscilloscope

Parts for circuit shown in Figure 28-1

- MC1408 or DAC08 8-bit D/A
- 741 op amp
- 10-kΩ potentiometer
- 2×2.4-kΩ resistors
- 50-pF capacitor

INTRODUCTION

Connecting an inexpensive IC D/A converter to a microcomputer port allows you to generate a desired voltage or sequence of voltages (waveform) under program control. This experiment is intended to give you some practice in testing and calibrating a D/A converter and some fun generating custom waveforms with a D/A converter.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Draw a diagram showing how an IC D/A converter is connected to a microcomputer port.
2. Write a simple program to test and calibrate a D/A converter connected to a microcomputer.
3. Write simple programs to produce sawtooth and triangle waveforms on the output of a D/A converter.
4. Write a program which uses a look-up table to produce any desired waveform on the output of a D/A converter.

PROCEDURE

1. If it is not already built, build the circuit shown in Figure 28-1.
2. Connect the data input pins of the D/A converter to port P1C of the SDK-86 board. Be sure to get the input pins of the converter connected to the port pins in the correct order. Connect the specified power supply outputs to the circuit.
3. Write a simple test program which initializes port P1C for output, outputs all zeros to the port, and stops.
4. With zeros on all of the data inputs, the current switches in the D/A converter are all off, and so there is no output from it. Predict the voltage that should be present on pin 2 of the op amp and the voltage that should be present on the output of the op amp for these input conditions. Run your test program and measure the voltages present at these two points to test your predictions. Record your predictions and results.
5. The last step should have shown you the lowest voltage output for this D/A circuit. The next step is to test and, if necessary, calibrate the full-

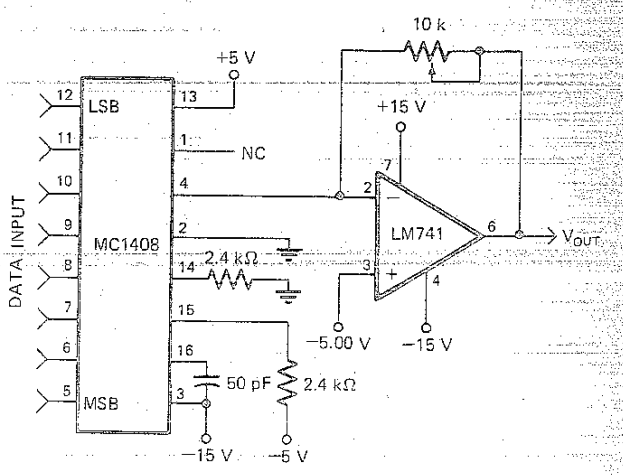


FIGURE 28-1 Schematic for an 8-bit D/A converter circuit.

scale output voltage for the converter. To do this, rewrite your test program so that it outputs all 1's to the D/A inputs. Run the new program and, if necessary, adjust the 10-kΩ potentiometer until the output voltage from the converter is about +4.961 V.

6. To help you understand the operation of this converter, think about and, in your report, answer the following questions.
   a. What is the resolution of an 8-bit D/A converter?
   b. What is the resolution of this converter in volts?
   c. Why is the full-scale output of this converter adjusted for 4.961 V instead of exactly 5 V?
   d. What binary word should be output from the microcomputer so that the output from the D/A is 0.00 V?

7. With relatively simple programs, you can produce almost any desired waveform on the output of the D/A converter. For your first demonstration of this, write a program for the following algorithm.

```
Initialize port for output
Initialize output value to 0
REPEAT
        output value to port
        increment value
FOREVER
```

8. Run the program and observe the D/A output waveform on an oscilloscope. Draw a section of the output waveform, showing voltage and time scales. Determine the frequency of the output signal. In your report explain what factors limit the output frequency.
9. Revise the program so that it increments the count by 2 before each output. Run the program and observe the output with a scope. Describe the effect this change had on the frequency of the output signal.
10. Revise the program so that the count is decremented before each output. Predict the output waveform this should produce; then run the new program and use a scope to see if your prediction was correct.
11. Write an algorithm and a program to produce a symmetrical triangle waveform on the output of the D/A converter. Run the program and observe the output waveform. For your report draw a labeled section of the observed waveform.
12. To produce more complex waveforms on the output of a D/A converter, the value for each point on the desired waveform is stored in a table in memory. A simple program then reads each value from the table in sequence, and outputs it to the D/A. As an example of this powerful technique, we will show you how to generate a sine wave.

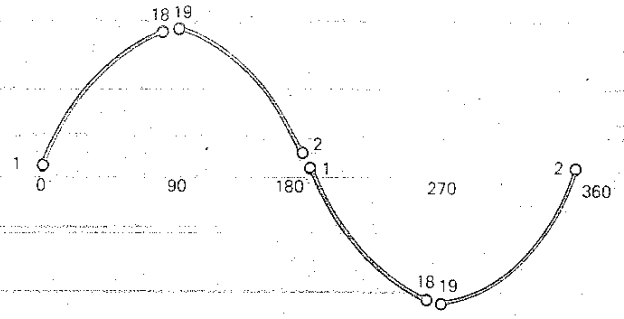


FIGURE 28-3 Diagram showing which table values to use to generate each quadrant of a sine wave.

The first step is to generate a table of values for each point on the desired output waveform. Figure 28-2 shows the basic values for the first 90° of a sine wave at intervals of 5°. Because a sine wave is symmetrical, you can easily determine the values for the other quadrants from these. If you look at the sine wave in Figure 28-3, you should see that for angles between 90° and 180°, these values repeat, but in reverse order. The values for 180° to 270° have the same magnitude as those for 0° to 90°, but they are below the zero reference (negative). For 270° to 360°, the values are negative and in reverse order.

The actual values in the table were produced as follows. The magnitude values for the table must be proportional to the sines of the desired angles. For an 8-bit converter, the basic table values must be in the range of 0 to 127 so that they can represent equal ranges above and below the center line of the waveform. Since the sine function ranges in value from 0.000 to 1.000, the basic values needed can be calculated by multiplying the sine of each angle by 127, as shown. The resulting values are rounded and converted to hex.

$\theta°$	$\sin\theta$	$127 \sin\theta$	Hex
0	0	0	00
5	0.087	11	0B
10	0.174	22	16
15	0.259	33	21
20	0.342	43	2B
25	0.423	54	36
30	0.500	64	40
35	0.574	73	49
40	0.643	82	52
45	0.707	90	5A
50	0.766	97	61
55	0.819	104	68
60	0.866	110	6E
65	0.906	115	73
70	0.940	119	77
75	0.966	123	7B
80	0.985	125	7D
85	0.996	126	7E
90	1.000	127	7F

FIGURE 28-2 Sine values converted to integer hexadecimal equivalents.

To produce the table values you need to generate the entire sine wave on this D/A, remember that the output voltage range of the converter is -5 V to 4.961 V. Since you would like the output waveform centered around 0 V, an offset of 80H must be used. For angles between 0° and 180°, the table values to be output are produced by adding an offset of 80H to the basic values in Figure 28-3. For angles between 180° and 360°, the basic table values in Figure 28-3 must be *subtracted from* the center point offset of 80H to produce the values to be output.

After you have thought your way through this, make up a table containing all the values to be output to the D/A. The table numbers next to the sine wave in Figure 28-3 should help you with this. Work through the four quadrants, one at a time, and make sure you don't use any value twice in succession or omit any value.

13. After you make up the table, write an algorithm for a program to output the table values to the D/A sequentially over and over. Include a simple delay loop so that you can vary the output frequency.
14. Write the program, run it, and observe the output it produces on the oscilloscope. Check the output waveform carefully to make sure it doesn't have any unwanted flat spots or notches.
15. (Optional) Change the values in the table to produce some other waveforms. If you want to hear what some strange waveform sounds like, you can connect a small speaker and a 220-Ω resistor in series between the output of the op amp and ground.

EXPERIMENT 29

Analog-to-Digital Conversion with a Microcomputer

REFERENCES

Hall: Chapter 10

EQUIPMENT AND MATERIALS

SDK-86 board

Power supply with outputs of ±5 V and ±15 V

IBM PC-compatible computer

Oscilloscope

Digital voltmeter

D/A circuit with current-to-voltage conversion and comparator (see Figure 29-1) or these parts:

MC1408L8 D/A converter

741 op amp

LM319 comparator

15-pF capacitor

10-kΩ trimpot

2.2-kΩ resistor

2×1-kΩ resistors

INTRODUCTION

A simple microcomputer-based A/D converter can be built by connecting a D/A converter and a comparator to ports, as shown in Figure 29-1.

The simplest type of A/D converter using this circuit with a D/A output is the counter type. An incrementing count is output to the D/A converter. After each increment, the comparator output is checked to find out whether the D/A output is less than or greater than V_{IN}. If the D/A output is greater than V_{IN}, the process is stopped and the count is proportional to the input voltage. If the D/A converter output is less than or equal to V_{IN}, the count is incremented and the new count is output to the D/A. Then the comparator output is checked again.

A faster but slightly more complicated method of A/D conversion is successive approximation. The circuit for this is the same as that in Figure 29-1, but the algorithm is different. A high is first output to the most significant bit input of the D/A and lows are output to all the other bits. If the comparator indicates that the resultant voltage is greater than V_{IN}, this bit is turned off and a 1 is output in only the next most significant bit position. If the resultant D/A output voltage is less than V_{IN}, this bit is kept on and a high on the next most significant bit is added to it. The output voltage produced by these two bits together is compared with V_{IN}. This bit is kept or reset based on the output of the comparator. The process continues until all 8 bit positions have been made a high, tried, and either kept or rejected.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Interface a D/A converter and comparator to a microcomputer to make a counter or a successive approximation A/D converter.
2. Write and test a program for a counter-type A/D converter.
3. Write and test a program for a successive-approximation A/D converter.

PROCEDURE

Counter-Type A/D Converter

1. Build the circuit shown in Figure 29-1 if it is not already built. Connect the D/A inputs to an output port on the SDK-86 board. Connect the output of the LM319 comparator to D0 of an input port on the SDK-86 board.
2. Write a simple program which initializes the output port and outputs all 0's to it. Run the program and use a voltmeter to check that the output of the 741 amplifier is very near 0 V, as it should be for this input value.
3. Change your test program so that it outputs all ones to the D/A converter. Run the program and adjust the 10-kΩ potentiometer until the output

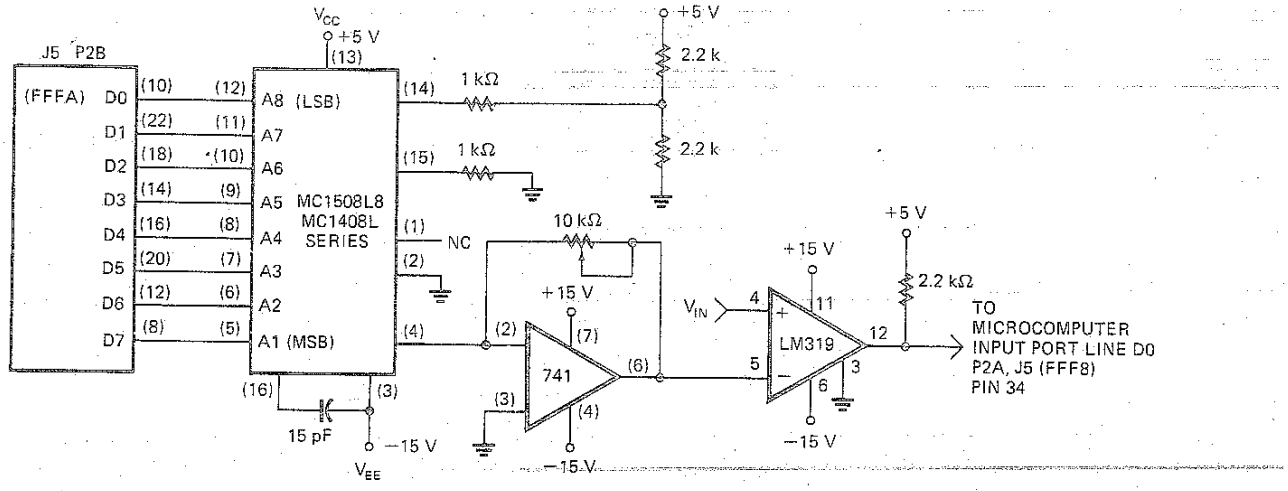


FIGURE 29-1 D/A converter and comparator circuit for a microcomputer-controlled A/D converter.

voltage from the 741 amplifier is 9.961 V. Your circuit is now set up to be a 0- to 10-V A/D converter.

4. Analyze the circuit in Figure 29-1 to determine if the comparator output is high or low when the D/A output is greater than the unknown input voltage. Record your result.
5. Write an algorithm for a program which uses the circuit in Figure 29-1 to perform A/D conversion using the counter method.
6. Write the assembly language program for the conversion algorithm and add the display procedure from Figure 20-1 to display the result of the conversion on the SDK-86 LEDs.
7. Apply 2.0 V to the V_{IN} point and run the program. Observe the value displayed on the LEDs. In your report compare the displayed value with the expected value.

NOTE: Do not apply an input voltage equal to or greater than the supply voltage of the comparator, or you may destroy it.

8. Run the program with some other input voltages, record the displayed binary (hexadecimal) result, and, in your report, compare the binary value you get with the expected value for each of these voltages. This system is functioning essentially as a digital voltmeter with a 0–10-V range. Include in your report a calculation showing the resolution of the converter in volts.
9. To measure the maximum conversion time for this A/D converter, make your program into an endless loop so that the conversion takes place over and over. Run your program with V_{IN} = 10 V, and observe the waveform on the output of the 741 op amp. Determine and record the time required for each conversion. Calculate the equivalent number of conversions per second.
10. Adjust V_{IN} to 5 V and run the program again to see the effect that V_{IN} has on the conversion time for this type of converter. Record the time taken for a conversion with this input voltage and calculate the equivalent number of conversions per second.

Successive-Approximation-Type A/D Converter

1. If you did not do the counter-type converter described previously, then initialize and calibrate the D/A converter as described in steps 1 to 4 of the counter section.
2. Write an algorithm for a program to perform A/D conversion by successive approximation as described before. To keep the program reasonably short, think about how bit testing can be done in a loop.
3. Write the source program for the algorithm and add the display procedure from Figure 20-1 to display the result of the conversion on the SDK-86 LEDs.
4. Apply 2.0 V to the V_{IN} point and run the program. Observe the value displayed on the LEDs. In your report compare the displayed value with the expected value.

NOTE: Do not apply an input voltage equal to or greater than the supply voltage of the comparator, or you may destroy it.

5. Run the program with some other input voltages, record the displayed binary (hexadecimal) result, and in your report compare the binary value you get with the expected value for each of these voltages. This system is functioning essentially as a digital voltmeter with a 0–10-V range. Include in your report a calculation showing the resolution of the converter in volts.
6. To measure the maximum conversion time for this A/D converter, make your program into an endless loop so that the conversion takes place over and over. Apply 10.00 V to the V_{IN} point and

run the program.

7. Use a scope to observe the waveform output on the output of the 741 op amp. Count the number of steps required to perform a complete conversion, and determine the amount of time required to perform a conversion. Draw and label the observed waveform. For comparison purposes calculate the number of conversions per second.
8. Adjust V_{IN} to 5 V and run the program again to see the effect that V_{IN} has on the conversion time for this type of converter. Draw the waveforms that you see for this conversion. Count the number of steps per conversion and calculate the number of conversions per second.
9. In your report compare the conversion time of this successive-approximation-type A/D converter with the conversion time of the counter-type A/D converter for the same V_{IN}.
10. Also in your report give one advantage and one disadvantage of using a microcomputer as part of the A/D conversion process.

EXPERIMENT 30

8087 Programming

REFERENCES

Hall: Chapter 11

EQUIPMENT AND MATERIALS

IBM PC or compatible computer with 8087 coprocessor installed

INTRODUCTION

As described in Hall: Chapter 11, the 8087 is a specialized coprocessor used for numerical calculations. If you install an 8087 in an IBM PC or compatible computer, it operates in parallel with the 8086 or 8088 in the system. The 8087 has its own instruction set. In a program, 8087 instructions are interspersed with 8086 instructions as needed to perform computations. When you run a program that contains both 8086 and 8087 instructions, each processor identifies and executes only those instructions in its own instruction set. The 8087 treats 8086 instructions as NOPs, and except for instructions which reference memory, the 8086 treats 8087 instructions as NOPs.

The goal of this experiment is to write a program which uses 8087 instructions to calculate the volume of a sphere.

OBJECTIVES

At the end of this experiment, you should be able to

1. Convert real numbers to and from the 8087 number formats.
2. Write an assembly language program which uses 8087 instructions to calculate the volume of a sphere.

PROCEDURE

Calculating the Volume of a Sphere

1. Read Hall: Chapter 11 (section on the 8087.)
2. Study the number formats shown in Hall: Figure 11-16.
3. Work through the example shown in Hall: Figure 11-17, which shows you how to convert numbers to the 8087 number format.
4. Write an algorithm for a program that calculates the volume of a sphere with a radius of 2.0 units using the formula

$$\text{Volume} = \frac{4}{3} \times \pi \times R^3$$

where R is the radius of the sphere.

5. Study the 8087 instruction set (Hall: Chapter 11, pages 371–375) and consider which instructions to use to implement your algorithm. You may want to work your way through the 8087 example program given in Hall: Figure 11-22 before you try to write your own program.
6. Write the required instructions, using the following points to help you.
   a. In the data segment, use the DD directive to reserve space for the volume to be calculated.
   b. Use the DD directive to set aside the memory locations for the radius of the sphere, and initialize those locations with the radius value of 2.0 units. Notice that you can use a real number in the data segment.
   c. Use the DD directive to set aside two memory locations and initialize them with the values 3.0 and 4.0.
   d. Do not forget to use the FINIT instruction to initialize the 8087.
   e. Find the instruction which allows you to load the 8087 with the value of π.
7. If you are using TASM as your assembler, type

   TASM /r filename,, <Enter>

   NOTE: /r tells TASM or MASM that your program contains 8087 instructions.

8. If you are using MASM to assemble your program, type

   MASM /r <CR>filename<F3><CR><F3><CR>

   <F3> means press the F3 key.
   <Enter> means press the Enter key.
   /r tells MASM that your program contains 8087

instructions.

9. Link your program and run it in DEBUG. If you use the DEBUG trace command, you should see each of the 8087 instructions displayed as it executes.
10. Use the D command to display the calculated volume left in memory after the program executes. Remember, the 8087 will write this number in short-real format.
11. Convert the short-real result you read from the memory location to a standard decimal number. Calculate the volume using a calculator and compare the result with the result produced by the 8087.
12. In your report describe the major advantage of using an 8087 to do numerical computations in a program.

EXPERIMENT 31

Introduction to the Turbo C++ Integrated Development Environment

REFERENCES

Hall: Chapter 12

EQUIPMENT AND MATERIALS

IBM PC or compatible computer

Borland Turbo C++ Professional System

INTRODUCTION

To develop a C program you need an editor to create a source program, a compiler to convert the source program to an object code file, a linker to link the various object code modules of your program into an executable (.EXE) file, and a debugger to help you get the program working correctly. For this exercise we chose the Borland Turbo C++ Integrated Development Environment (IDE,) which has all these features and many more.

The term integrated development environment means that you can access all the programming tools from one on-screen menu. We chose the Borland IDE system because it is very powerful but easy to use, the company has a generous educational pricing policy, and the company gives very good support if you encounter problems. Other available tool sets such as Programmer's Workbench from Microsoft are very similar, so you should have little trouble adapting the following discussion if you have some other set of programming tools.

OBJECTIVES

At the end of this exercise, you should be able to

1. Write, compile, run, and debug C programs using an integrated program development environment such as Borland's Turbo C++.

PROCEDURE

For the following discussions we assume your Borland Turbo C++ Integrated Environment tools and libraries are all installed in a hard disk directory named C:\tc as described in the Borland manual, and the path command in your autoexec.bat file contains C:\tc and c:\tc\bin. We further assume that your work disk is a floppy in the A drive.

1. To bring up the Turbo C++ environment, type tc and press the <Enter> key. After a short pause the main menu screen shown in Figure 31-1*a* will appear. Each of the entries in the banner at the top of the display represents a pull-down menu of commands. To get to one of these menus you hold down the <Alt> key and press the letter key which corresponds to the first letter of the desired menu's name. If you have a mouse on your system, you can use the mouse to move the cursor to the desired menu box and click the left mouse key to bring down a menu.

   To see how this works, hold down the <Alt> key and press the <F> key to get to the File menu. After you read the menu, press the <Esc> key to leave the menu without executing a command.

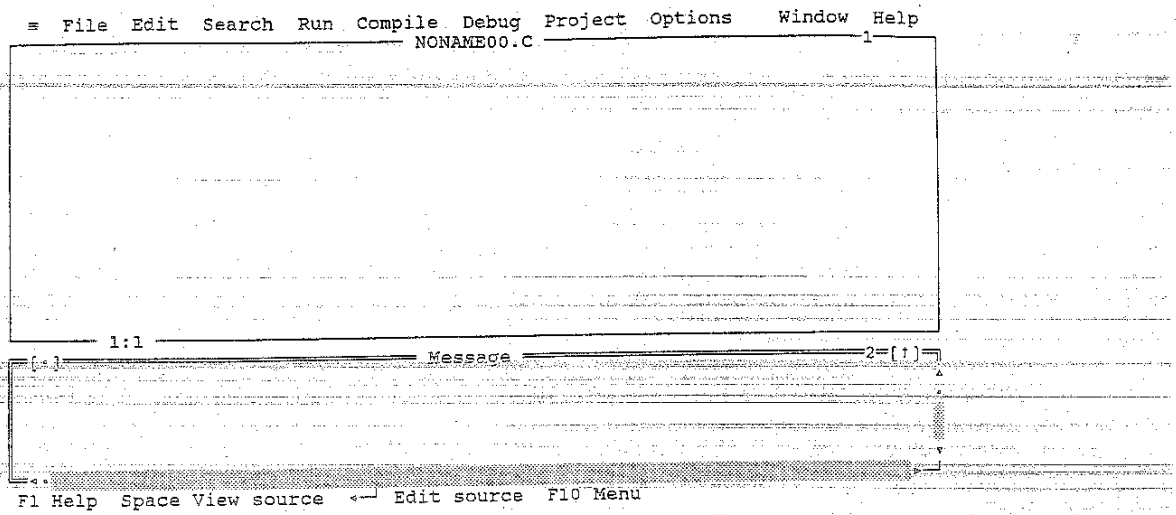
NOTE: Almost all commands in the Turbo C environment can be executed by pressing a "hot key" combination such as <Alt>-<F1>, which executes the command directly, or by working your way through a sequence of menus with a mouse or the arrow keys. When you are first learning a new system, the menu method helps you better learn the available features, so we will emphasize that approach.

2. One of the first things you have to do when you create a program is to tell the compiler, linker, etc., where to put the .OBJ and .EXE files they create. You use a command in the Options menu to do this. To get to the Options menu you hold down the <Alt> key and press the <O> key. Figure 31-1*b* shows the Options menu that appears. The command you want in the options menu is the Directories command. To get to the submenu for this command you simply press the <D> key, and the window display shown in Figure 31-1*c* will appear. You want to assign an Output directory, so you hold the <Alt> key down and press the <O> key to get to that line.

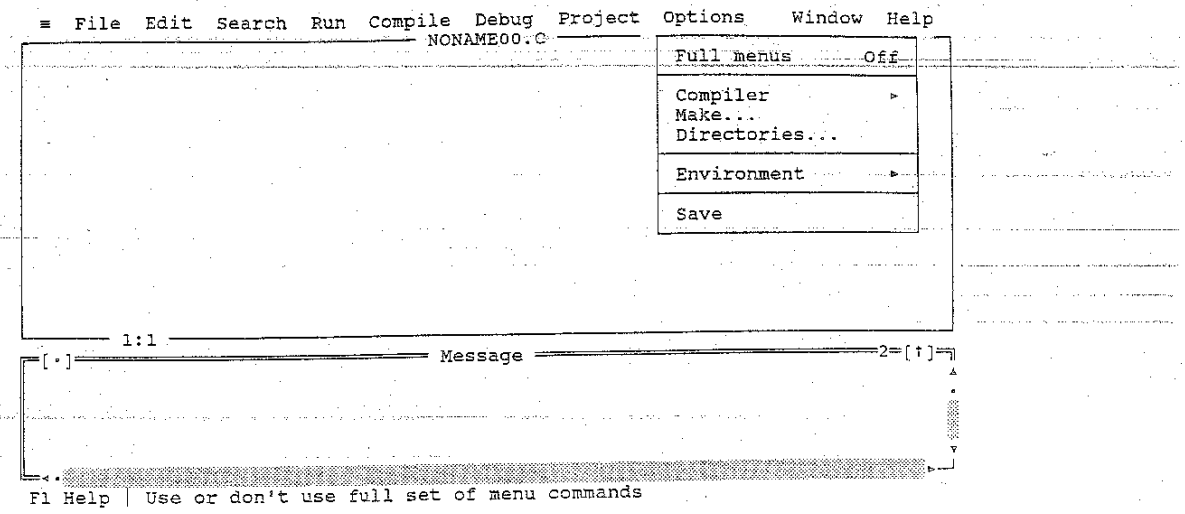
Now, suppose that you don't understand just exactly what you are supposed to put in this line. Press the <F1> key to display an explanation of the command requirements and, in some cases, examples. In this case the help window tells you that the .OBJ, .EXE, and .MAP files will be stored in the location you enter as the output directory. After you have read the help window, press the <Esc> key to close it.

Since you want your output files to go to your work disk in the A drive, just type A: and press <Enter>.

3. The next step in developing a program is to use the editor to enter the source text for the program. For a new program you go to the File menu as described in step 1 and press the <N> key. A blinking cursor will appear in the large window on the screen, and you can type in the text of your source program. Type in the example program in Figure 31-2. You may find it helpful

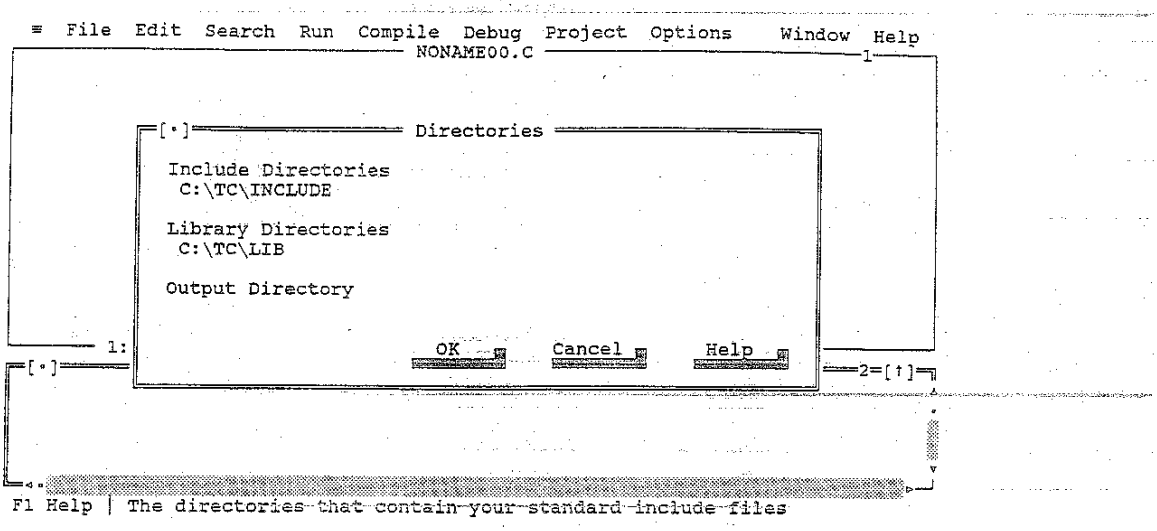


(*a*)



(*b*)

FIGURE 31-1 Turbo C++ Integrated Development Environment (IDE) screen displays. (*a*) Main menu edit window. (*b*) Options submenu. (continued, p. 94)



(c)

FIGURE 31-1 (continued from p. 93) (c) Directories submenu of Options menu.

to use spaces instead of tabs to format your programs, because the default tab setting of most printers is 8, and with this setting C programs do not usually fit easily on 8.5-in.-wide paper. Incidentally, the IDE editor accepts Wordstar commands such as Ctrl-G to delete a character, Ctrl-T to delete a word, and Ctrl-Y to delete a line, so if you are familiar with Wordstar you should find the editor very easy.

4. After you type in the source file, you need to save it on your work disk. To do this go to the File menu, select the Save As line, and press <Enter>. When a small window appears, you can type in a filename such as a:SELL.C and press the <Enter> key.

5. The next step is to compile the program to generate the OBJ file. To do this, hold down the <Alt> key and press the <C> key to go to the Compile menu and press the <C> key again to run the compiler. If the compiler finds any errors, it will display a window with a flashing error message. Press a key to see the error message(s) displayed in the message window at the bottom of the screen. For this example you should see the error messages shown in Figure 31-3 because we intentionally left out a required # sign in front of the include directive in the program in Figure 31-2. A highlighted line in the source program indicates the statement that caused the error highlighted in the message

```
/* COMPUTE THE SELLING PRICE OF 10 ITEMS */

include <stdio.h>
#define PROFIT 15
#define MAX_PRICES 10
int cost[] = { 20,28,15,26,19,27,16,29,39,42 }; /* array of 10 costs */
int prices[10];                                 /* array to hold 10 prices */
int index;                                      /* variable to use as index */

main()
{
        for (index=0; index <MAX_PRICES; index++)    /* for loop to compute */
        prices[index] = cost[index] + PROFIT;        /* 10 prices */

        for (index=0; index <10; index++)            /* loop to display results */

        printf("cost = %d, price = %d, \n", cost[index], prices[index]);

}
```

FIGURE 31-2 Example C program for learning IDE features.

window. Press the <Enter> key to switch from the message window to the edit window and correct the error. For this error all you have to do is insert the missing # before the include directive.

NOTE: A major error such as this will cause many errors throughout the program, so when you find such an error it is a good idea to compile the program again before you start chasing down the other indicated errors.

6. To recompile the program all you have to do is go to the Compile menu and press the <C> key. Once the compile is successful you should always save your source file before continuing. To do this go to the File menu and press the <S> key. This is important, because if your program locks up the machine when you run it, your program will be lost.
7. The next step in developing a program is to generate an executable file which you can run. For this example the file will be given the name sell.exe by the linker which generates it. There are two ways to generate the .EXE file. One way is to go to the Compile menu and press the M key to make a .EXE file. The second way to generate the .EXE file is to go to the Run menu and press the <R> key to run the program. This second sequence of commands generates the .EXE file and also runs it. For a single module program this second method is obviously the easiest, so use it for your program.
8. When the program has finished running, the display will return to your source program. For a program such as sell.c, which outputs to the screen, this display will be left in an alternate screen buffer. To see the output of your program, hold down the <Alt> key and press the <F5> key. To toggle back to the IDE screen, hold down the <Alt> key and press the <F5> key again.
9. Now, suppose that your program doesn't work correctly the first time you run it. The Turbo C++ IDE contains a powerful "source-level" debugger. A source-level debugger allows you to view your source program on the screen and execute through it one statement at a time or run to a breakpoint you placed on a statement and watch the values of variables change as program statements execute. The debugger is integrated with the editor, compiler, and linker, so when you find an error, you can just go back to the edit window, fix the error, recompile the program, and run the program again, all from the same main menu.

   Here's a short example of how you watch the values of some variables change as you single step your way through the example program, sell.c.

NOTE: For this process to work as described, the program must have been just compiled and linked so the debugger has the needed "hooks."

   Let's assume that you want to observe the values of cost[index] and prices[index] change as you single step through the program. To get ready to single step, go to the Run menu, then press the <T> key to Trace into the program. A highlighted bar will appear on the first line of your program.
10. To put a "watch" on one of the desired variables

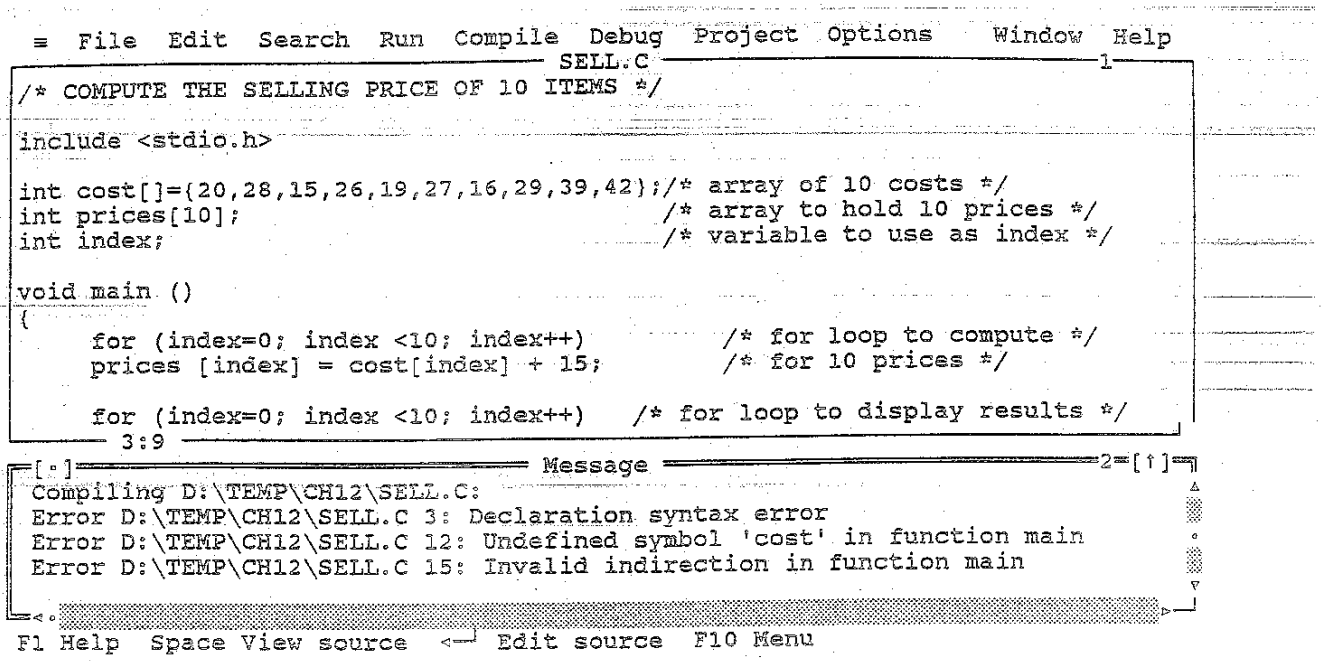


FIGURE 31-3 Error messages produced by omiting a # symbol before the include directive.

go to the Debug menu and press the <W> key to bring up the Watch submenu. Then press the <A> key to add a watch. When a small window appears, type in cost[index] and press the <Enter> key. The name of the variable on which you placed a watch should appear in the Watch window at the bottom of the screen. Repeat the procedure to put a watch on prices[index].

11. Now, to execute the first line of the program, press the <F7> key. The highlighted bar will move to the next statement, and the values of the variables in the Watch window will be updated to show the results of executing that instruction. Step through the for loop in the program a few times and record the displayed values for the variables.

12. If at any time you want to determine the value of some variable that you forgot to put in the Watch window, you go to the Debug menu and then press the <I> key to execute the Inspect command. When a small window appears, you type in the name of the variable that you want to look at and press the <Enter> key. The current value of the specified variable will be returned in the Inspect window. You press the <Esc> key to get back to stepping through the program. Use this procedure to evaluate the current value of index and record the value you get.

13. If you find an error as you step through your program, you can just edit the source code version. After you save the new version you can run the program again by simply going to the Run menu and pressing the <T> key. In this case the IDE tools will detect that the source program has been modified, and they will automatically compile, link, and put the highlighted bar on the first line of your program. You can then single step through the program by just pressing the <F7> key. As you work your way through later C experiments, you will become more familiar with the capabilities of the IDE, but this exercise has introduced you to all you need to develop many programs. To reinforce the basic procedures in your mind you should quickly work your way through this exercise again.

    Hold down the <Alt> key and press the <X> key or go to the File menu and press the <Q> key to leave the IDE. Then type tc and press the <Enter> key to bring up the environment again. When the main screen appears, go to the File menu and press the <O> key to execute the Open command. When a window appears, there are three ways you can specify the file you want to edit.

    a. You can just type in the name of the file you want to work on and press the <Enter> key.

    b. You can use the arrow keys to step down through the list of files displayed in the file window until the highlighted box is on the file you want and press the <Enter> key. To complete the selection process, use the <Tab> key to step highlighting to the Open button in the upper right corner of the window and press the <Enter> key again.

    c. If you have a mouse, you can move the mouse to the desired file in the displayed list, click the left mouse button, and then move the cursor to the Open box in the display and click the left mouse key again.

    After any of these operations the selected file should appear in the edit window, and you can proceed with compiling and running the program as described before.

14. You can run the .EXE file for this example program directly from the command line. To do this, at the DOS prompt just type the name of the file without any extension and press the <Enter> key. For the example here just type sell and press the <Enter> key. After the program runs, the display shown in Hall: Figure 12-1*b* should appear on the screen.

EXPERIMENT 32

C Programming Practice

REFERENCES

Hall: Chapter 12

EQUIPMENT AND MATERIALS

IBM PC or compatible computer

Borland Turbo C++ Professional System or equivalent

INTRODUCTION

In Experiment 31 you used C programming tools to develop an executable file for an example program. In this experiment you will write, compile, and run two simple C programs of your own. These programs use techniques demonstrated in the figures in Hall: Chapter 12, so they should not be difficult if you are very careful with each step.

The major tasks in the programs for this experiment are as follows:

1. Declare an array for 50 characters.
2. Prompt the user to enter his or her name.
3. Read an entered name into the array.
4. Determine the number of characters in the name.
5. Print out appropriate text and the number of characters in the name.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Declare and initialize C int and char type variables.
2. Use a predefined function to read a string from the keyboard.
3. Use the index method to access the elements of an array.
4. Use the C while structure to implement a desired sequence of actions.
5. Use the C if structure to choose one of two actions.
6. Use a predefined function to send text to the display.

PROCEDURE

Index method

1. The task list in the introduction gives most of the major steps in this program, so all you have to do is figure out how to implement each step and write the C statements for the steps. Use the figures and the accompanying text in Hall: Chapter 12 to help you write the C statement for the following.
   a. A preprocessor directive which tells the compiler that the program calls predefined I/O functions.
   b. A statement which declares a character array called name for up to 50 characters.
   c. A statement which declares an int variable called index and initializes it with a value of 0.
   d. A statement which declares an int variable called count and initializes it with 0.
   e. A statement which calls a predefined function to send a prompt message to the display. This statement should tell the user to type in his or her name and press <Enter>.
   f. A statement which calls a predefined function to read the user-entered string and put it in the array called name.
2. You now have the entered string of characters in the array in memory. The next task is to read through the string and count the number of characters, excluding spaces and periods. When one of the predefined functions reads a string from the keyboard and writes it into a char array, the null character, 00H, is automatically written as a sentinel at the end of the string. In program statements you can use the predefined constant NULL to determine when you have reached the end of the string.

   Use the program examples in Hall to help you set up a WHILE structure and the indexed array method to step through the elements of the string as long as the value is not equal to NULL. Remember to increment the index value each time through the loop.
3. If the value of the character at any index value in

the array is not a space and not a period, you want to increment the count. If the value of the character is one of these, you simply want to go on to the next value without incrementing the count.

Use the program examples in Hall to help you write the C if structure which implements these actions.

4. Now that you have the letter count, the next step is to tell the world about it. Write a statement which uses a predefined function to write your name and the number of letters it contains on the CRT.
5. Terminate your program with an exit() statement.
6. Invoke the Turbo C++ IDE environment as described in Experiment 31; then enter your program. Remember to save your source program to disk so that you don't lose your program if the power fails.
7. Compile your program. To refresh your memory, you can do this by pressing the <F10> key, the <C> key, and then the <C> key again. Repeat the edit-compile cycle until you see the "success" message.
8. To run your program from the IDE, press the <F10> key, the <R> key, and then the <R> key again. If your program works correctly, it will prompt you to enter your name, count the number of characters, and then return to the IDE screen. To see the results of your program on the alternate display, hold down the <Alt> key and press the <F5> key. To return to the IDE hold down the <Alt> key and press the <F5> key again.
9. If your program does not work correctly, use the Trace, Watch, and Evaluate features described in Experiment 31 to help you debug it.
10. When your program works correctly from the IDE, hold down the <Alt> key and press the <X> key to exit to DOS.
11. To run your program from the DOS prompt, simply type the name of the program with no extension and press the <Enter> key.

Pointer Method

1. As described in Hall, the array index method works, but the pointer method is more efficient. The structure of this program is basically the same as that in the preceding section except that you will use a declared pointer to access the elements of the char array.

   Study the example programs and the accompanying text in Hall Chapter 12 to help you write a C statement which declares a char type pointer called nameptr and initializes the pointer to point to the start of the array called name.
2. Also, from the examples in Hall determine how you can use the pointer to access a desired value in the array instead of using a reference such as name[index].
3. Use the DOS copy command to make a copy of the source program you wrote for the first section of this experiment.
4. Invoke the IDE and open the copy you made. Add the pointer declaration statement and modify the program as needed to use the pointer to access the elements in the array.
5. Compile the program and correct any errors or warnings.
6. Run and, if necessary, debug the program.
7. Exit from the IDE and run your program from the DOS prompt.

EXPERIMENT 33

Displaying Characters on an IBM PC Screen

REFERENCES

Hall: Chapter 13

EQUIPMENT AND MATERIALS

IBM PC, PS/2, or compatible computer

INTRODUCTION

Characters to be displayed on an IBM PC or PS/2 display are stored in a special section of memory called the frame buffer. Two consecutive bytes in the frame buffer are used to store the data for each character position on the screen, so a display of 25 rows with 80 characters per row uses a 25×80 or 4000-byte frame buffer.

Each even-addressed byte holds the ASCII code for the character to be displayed in the corresponding position on the screen. As shown in Figure 33-1*a*, the next higher odd-addressed byte holds the attribute code for that character. The attribute code determines the character's appearance on the screen—i.e., blinking, intensified, or reverse video or normal, color, etc. Figure 33-1*b* shows the bit format for monochrome attribute bytes and Figure 33-1*c* shows the codes that are used in the attribute byte for color character displays.

The goals of this experiment are to fix in your mind the correspondence between display memory locations and screen positions, familiarize you with the concept of attribute codes, and give you some practice using the BIOS INT 10H procedure to write characters to desired positions on the screen.

OBJECTIVES

At the end of this experiment, you should be able to

1. Calculate the frame buffer memory location which corresponds to a given position on the display.
2. Write a character with specified attributes at a specified position on the screen.
3. Use the BIOS INT 10H procedure to display characters with a desired attributes at any location on the screen.

PROCEDURE

1. Use the reference manual for your computer to determine the starting address of its display memory. For the IBM PC monochrome display adapter this address is B0000H. For most other systems this address is B8000H. In this experiment we will be using the address B8000H, so you will have to remember to substitute the correct address in our instructions if the base address for the frame buffer is different for your system.
2. To start, let's demonstrate that the code for the character displayed in the top left corner of the screen is stored at the first location in the frame buffer.

   At the DOS prompt type

   DEBUG

   and press the <Enter> key. When the (-) prompt appears, type

   RDS

   and press the <Enter> key. Then type

   B800

   and press the <Enter> key again. This sequence will load the DS register with the segment base address for the frame buffer.

   Next use the debug Examine (E) command to place the ASCII code for A at offset 0 in DS. To do this, type

   E0

   and press the <Enter> key. Then type

   41

   and press the <Enter> key again. You should now see the character A displayed at the top left corner of the screen. To display the contents of the first few locations in the frame buffer, type

   D0 40

   and press the <Enter> key. You should see that

DISPLAY-CHARACTER CODE BYTE								ATTRIBUTE BYTE							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
EVEN ADDRESS								ODD ADDRESS							

(*a*)

ATTRIBUTE FUNCTION	ATTRIBUTE BYTE							
	7	6	5	4	3	2	1	0
	B	R	B	G	I	R	B	G
	FG	BACKGROUND			FOREGROUND			
NORMAL	B	0	0	0	1	1	1	1
REVERSE VIDEO	B	1	1	1	1	0	0	0
NONDISPLAY (BLACK)	B	0	0	0	1	0	0	0
NONDISPLAY (WHITE)	B	1	1	1	1	1	1	1

I = HIGHLIGHTED FOREGROUND (CHARACTER)
B = BLINKING FOREGROUND (CHARACTER)

(*b*)

I	R	G	B	COLOR
0	0	0	0	BLACK
0	0	0	1	BLUE
0	0	1	0	GREEN
0	0	1	1	CYAN
0	1	0	0	RED
0	1	0	1	MAGENTA
0	1	1	0	BROWN
0	1	1	1	WHITE
1	0	0	0	GRAY
1	0	0	1	LIGHT BLUE
1	0	1	0	LIGHT GREEN
1	0	1	1	LIGHT CYAN
1	1	0	0	LIGHT RED
1	1	0	1	LIGHT MAGENTA
1	1	1	0	YELLOW
1	1	1	1	HIGH INTENSITY WHITE

(*c*)

FIGURE 33-1 Data storage formats for IBM graphics board operating in alphanumeric mode. (*a*) Character byte and attribute byte. (*b*) Attribute byte format. (*c*) Attribute byte coding for color displays.

the first location, which corresponds to memory address B8000, contains 41. The next higher location and all odd addresses after that contain attribute codes. Use Figures 33-1*b* and 33-1*c* to determine the attributes specified by the codes in these bytes.

3. Calculate the frame buffer memory address that corresponds to the bottom right corner of the screen. Then use the DEBUG E command to write the code for an X to this location so you can see if your calculation was correct.
4. When you get the X displayed in the lower right corner of the screen, work out the attribute code that will cause the X to be displayed in reverse video. Then use the DEBUG E command to write the attribute byte to the correct address for that character.
5. If you have a color monitor, experiment with attribute codes that produce different character (foreground) and different background colors.
6. Press the <Q> key and then the <Enter> key to exit DEBUG.
7. The next step is to write a simple program which displays your name with attributes you choose in the center of the screen. Write an algorithm for a simple program which reads character and attribute codes from an array in memory and writes them to the appropriate locations in the frame buffer.
8. Write the assembly language program for your algorithm.

NOTES: You can use the ES register as a pointer to the base address of the frame buffer.

You will need the instructions that allow you to make a clean exit from your program to DOS. Do this by adding the following instructions to the end of your program:

```
MOV AX,4C00H
INT 21H
```

9. Assemble and link it to produce a .EXE file. Run the program directly from DOS by typing the name of the program (without any extension) and pressing the <Enter> key. Debug the program if necessary.
10. As you demonstrated in the last three steps, you can produce a desired display by writing character and attribute codes directly to locations in the frame buffer. The disadvantage of this approach is that you have to calculate the memory addresses which correspond to desired screen coordinates. An easier method for writing characters to the display is to use the BIOS INT 10H procedure.

    Use the discussion in Hall: Chapter 13 and the description of the BIOS INT 10H subprocedures in Hall Figure 13-19 to help you write a simple program which clears the screen, positions the cursor at the appropriate location, and writes your name in the center of the screen with attributes you choose.

HINT: You may find INT 10H subprocedures 2, 6, and E helpful in your project.

11. (Optional) Experiment with enhancements such as drawing a box of blinking asterisks around your name.

EXPERIMENT 34

Assembly Language Color Graphics

REFERENCES

Hall: Chapter 13

EQUIPMENT AND MATERIALS

IBM PC or PS/2-compatible computer with color monitor

INTRODUCTION

As described in Hall there are now a wide variety of color-display modes available. In an attempt to make this experiment applicable for the widest range of systems we based our discussion on the IBM CGA display mode. Most of the more advanced systems can be programmed to operate in this mode as well as in the higher-resolution modes. If you have an EGA or a VGA system, you can use the discussion in Hall: Chapter 13 and the examples in Figures 13-20 and 13-21 to adapt the experiment so that it takes better advantage of the higher resolution available in these systems.

As described in the reference, an IBM PC-CGA can be operated in a character mode or in one of three graphics modes. In the character mode, which is used for displaying text, a character generator is used to generate the dot patterns from ASCII codes stored in the display RAM.

In the medium-resolution graphics mode, which you will be using for this experiment, the entire screen is treated as a matrix of dots (pixels) 320 pixels wide by 200 pixels high. Figure 34-1*a* shows how the pixel data is arranged in memory bytes. Note that the data for the first pixel is stored in the upper 2 bits of the byte. Since each pixel is represented by 2 bits, each pixel can be specified to be one of four colors, as shown in Figure 34-1*b*. The default background color, specified by a pixel code of 00, is black. Figure 34-1*c* shows the two available color sets. The default color set is color set 2, which will produce a blue (cyan) dot for 01, a magenta dot for 10, and a white dot for 11. Later we will tell you how to change the background color and the color set. The point for you to realize here is that, within the limits shown, you can specify the color of a dot by the bit pattern you put in the corresponding location in the display refresh RAM.

The second point you need to understand before getting started is related to the fact that the color adapter board uses interlaced scanning. In order to make memory access more efficient while refreshing the screen, the pixel data for even-numbered scan lines is put in successive memory locations between B8000H and B9F3FH. The pixel data for odd-numbered scan lines is put in successive memory locations between BA000H and BBF3FH. For the first part of this experiment you will use debug commands to experiment with loading various bit patterns directly into the display refresh RAM. Next you will use the BIOS INT 10H procedure to write a line of dots on the screen. Finally, you will write a program which enables you to draw colored lines on the screen using the cursor move keys, similar to the way you use an Etch-A-Sketch™.

7	6	5	4	3	2	1	0
C1	C0	C1	C0	C1	C0	C1	C0
FIRST DISPLAY PEL		SECOND DISPLAY PEL		THIRD DISPLAY PEL		FOURTH DISPLAY PEL	

(a)

C1	C0	FUNCTION
0	0	DOT TAKES ON THE COLOR OF 1 of 16 PRESELECTED BACKGROUND COLORS
0	1	SELECTS FIRST COLOR OF PRESELECTED COLOR SET 1 OR COLOR SET 2
1	0	SELECTS SECOND COLOR OF PRESELECTED COLOR SET 1 OR COLOR SET 2
1	1	SELECTS THIRD COLOR OF PRESELECTED COLOR SET 1 OR COLOR SET 2

(b)

COLOR SET 1	COLOR SET 2
COLOR 1 IS GREEN COLOR 2 IS RED COLOR 3 IS BROWN	COLOR 1 IS CYAN COLOR 2 IS MAGENTA COLOR 3 IS WHITE

(c)

FIGURE 34-1 (*a*) Memory arrangement of pixel data for medium-resolution mode of IBM PC. (*b*) Four colors specified by pixel codes. (*c*) Available color sets.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Use the debug commands to fill areas of the CRT screen with lines of a specified color.
2. Write a simple program which uses the BIOS INT 10H call to draw a specified line on the screen.
3. Write a program which draws a colored line on the screen in response to commands entered by a user on the cursor move keys.

PROCEDURE

1. The first step is to switch the computer to medium-resolution color graphics mode. To do this, edit, assemble, link, and execute the initialization program shown in Figure 34-2. This program uses the BIOS INT 10H procedure to set the mode of the color adapter card.

   NOTE: To return to the normal, low-resolution graphics mode at any time, make sure that the DOS prompt is present, type

   MODE CO80

   and press the <Enter> key

2. Now that the system is in graphics mode, you can use the debug commands to load some pixel codes into the display RAM and see the effect this produces on the display. To start, run DEBUG, type

   RDS

   and press the <Enter> key.

   After debug responds with the current contents of the DS register, type

   B800

   and press the <Enter> key.

   This sequence of commands points DS at the start of the display refresh RAM to make it easier to write in pixel codes.

3. Using Figure 34-1*a* and *b*, make up a pixel data byte which will specify blue for each of the four dots represented by a byte. Then use the fill command to write this byte in the first 200H locations of the display RAM. To refresh your memory, the format for the fill command you need here is F(starting offset in data segment) (ending offset in data segment) (value to put in each byte).

   An example is

   F0 100 11 <Enter>

4. After you run the fill command, you should see some blue lines across the top of the screen. Note that the blue lines have black lines between them. For the next bit of magic, we want you to display magenta lines between the blue lines.

5. To start, use Figure 34-1*b* to help you figure out the code you need to specify magenta for the four pixels represented by a byte. Then use debug commands to write this byte to the first 200H locations in the ODD scan line region of the display refresh RAM. Remember, the odd scan line region of the display refresh RAM starts at address BA000H.

6. If this step worked correctly, experiment with putting some different-colored lines at other locations on the screen, just for fun.

   NOTE: You can "erase" any section of the screen by simply writing black lines to that section.

7. Next determine the addresses you need to draw a single blue line all the way across the screen on scan line 40H. Use the fill command to draw this line; then quit DEBUG.

8. Step 7 should have shown you that drawing lines is a great deal of work when you have to figure out the starting address and the ending address. Fortunately, the BIOS INT 10H procedure has a subprocedure which will plot a point at the screen coordinates you pass to it in the CX and DX registers. You can call this procedure over and over to plot the points of a desired line.

   Write and run a simple .EXE program which initializes the display adapter, draws a horizontal magenta line from point (40H, 0) to point (40H, 80H) using the BIOS INT 10H procedure, and returns to DOS. (If you have an EGA or VGA system, you can use the "write dot" procedure in Hall: Figure 13-21 to draw the line.)

9. After your program draws the horizontal magenta line correctly, add a program section which draws a vertical magenta line from point (40H, 80H) to point (20H, 80H). Then add program sections which complete the rectangle. You now have a graphics program with which you can draw any size square or rectangle by modifying the coordinate values you use in it.

10. The final part of this experiment is to write a program which allows a user to draw lines on the screen one dot at a time using the four cursor-move keys on the numeric keypad of the keyboard. The program should wait for the user to enter a key, determine if the key is a cursor-move key, and, if it is, write one dot in the direction indicated by that key. If the <Esc> key is pressed, execution should exit to DOS.

    Write an algorithm for a basic command recognizer which reads user-entered codes and takes appropriate action. Initially assume that any key codes other than cursor-move codes and the <Esc> key are ignored.

11. Use the following hints to help you write the assembly language instructions for your algorithm.

    a. The BIOS INT 16H procedure can be used to read in characters from the keyboard. If the INT 16H instruction is executed with 00H in

```
;ABSTRACT:
;       This program sets up the IBM PC CGA adapter
;       for 320 x 200 medium resolution graphics mode.
;       To return to normal character mode after running this
;       program, when you see the DOS prompt type: MODE CO80
;       then press the return key.

CODE    SEGMENT
        ASSUME CS:CODE
START:  MOV AL, 4               ; Mode byte
        MOV AH, 0               ; Subfunction number
        INT 10H                 ; BIOS video function
EXIT:   MOV AX, 4C00H           ; Return to DOS code
        INT 21H                 ; DOS function call
CODE    ENDS
        END START
```

FIGURE 34-2 Program to initialize IBM PC CGA adapter to the medium-resolution graphics mode.

the AH register, the called BIOS procedure will wait for the next key to be pressed on the keyboard and then return the ASCII code for that key in the AL or AH register. The ASCII codes for the letter and number keys are returned in the AL register. The codes for the cursor keys are returned in the AH register. The following list summarizes the codes returned for certain keys after the INT 16H procedure has executed.

UP arrow	AH = 48H
DOWN arrow	AH = 50H
RIGHT arrow	AH = 4DH
LEFT arrow	AH = 4BH
ESC key	AH = 01H or AL = 1BH
0 key	AL = 30H

For further information about the INT 16H procedure consult HALL Figures 13-1 and 13-2 and read the accompanying discussions.

b. The following instructions will clear the screen.

```
MOV AH, 6       ;Scroll up
MOV AL, 0       ;Blanking code
MOV CX, 0       ;Screen coordinates 0,0
MOV DL, 79      ;Screen coordinates
MOV DH, 24      ;79,24
INT 10H
```

c. Include instructions to set the dot color to blue and start the dot in the home (0,0) position.

12. Once you get the basic draw-line program working, add the following improvements to it.

a. Add instructions which recognize other keys, so that the user can enter, for example, numbers 0–3 to specify the color for the following dots. A dot or line can then be erased by switching the color to black (background) and moving backward over the line. Also, by adding this feature you can draw disconnected figures.

b. Add instructions which change the background color from black to some other color. This is done by loading 0BH in AH, loading 00H in BH, loading a number between 00H and 0FH in BL to specify the desired background color from the choices in Figure 34-3, and executing the INT 10H instruction.

c. Add instructions which change the color palette from the blue/magenta/white to the green/red/yellow palette. This is done by loading the AH register with 0BH, loading the BH register with 01H, loading the BL register with 00H, and executing the INT 10H instruction. To change back to the blue/magenta/white palette, do the same call with BL = 01.

13. (Optional) If you have a mouse, consult the discussion in Hall Chapter 13 and the manual for your mouse to see how you can use the mouse to move the cursor around on the screen.

NOTE: You have to reduce the mickey/pixel ratio so that the display can keep up with the mouse.

EXPERIMENT 55

C Graphics Programming

REFERENCES

Hall: Chapters 12 and 13, Appendix C

Borland Turbo C++ Library Reference

EQUIPMENT AND MATERIALS

IBM PC or compatible computer with color display

Borland Turbo C++ Professional System or equivalent

INTRODUCTION

The main goal of this exercise is for you to have some fun producing graphics displays on a PC or PS/2-type computer. Unlike the previous experiments, which had prescribed outcomes, this experiment is open ended. We will give you a running start, and then you can be as creative as your time and energy permit.

Hall: Figure 13-23 shows a C program which uses Turbo C library functions to initialize the graphics adapter, draw an eight-segment pie graph on the screen, open a graphics window, draw some figures in the window, and close the window on user command.

You will use this program as a start and then use the techniques it demonstrates to create a program which impresses your friends and, we hope, your instructor.

OBJECTIVES

At the conclusion of this experiment, you should be able to use the predefined C functions to

1. Initialize a graphics adapter to the desired mode.
2. Set background color and drawing color.
3. Draw pie graphs.
4. Draw lines, circles, and rectangles.
5. Open a window and save the region under the window.
6. Close a window and restore the original display.

PROCEDURE

1. To get an overview of how the program in Hall: Figure 13-23 works, carefully read the comments. Then read the discussion of the program in Hall: Chapter 13 to get the details of how each operation is done.
2. The program in Hall: Figure 13-23 is designed to run on a VGA system in 640 × 480 × 16 color mode. If you have some other system, you need to change some of the values in the program so it will run on your system. Appendix C shows the drivers and modes available in the Turbo C++ library. Use these to help you determine the correct initialization for your system. Also, calculate new values for the screen coordinates so that the pie, circles, window, etc., fit on your screen.
3. Invoke the TC IDE and type in your modified version of the program in Hall: Figure 13-23.
4. Compile the program and correct any errors.
5. Run the program; after you tire of the display, press the <e> key and then press the <Enter> key to return to the IDE.
6. Hold down the <Alt> key and press the <F5> key to see the pie display without the window. Hold down the <Alt> key and press the <F5> key again to get back to the IDE.
7. Now that you have a basic graphics program executing, the next step is make some simple creative modification. For a start you might try drawing the pie with more pieces so you can serve more people. This will give you some quick practice modifying, compiling, and running the modified program.
8. Plan a "modern art" or action display for the screen and use the predefined C functions to produce it. To help you, Appendix C shows a summary of the major Turbo C++ graphics functions, including some that we did not use in the program in Hall: Figure 13-23. You might, for example, use the outtextxy function described there to sign your name to the display. If you have the Turbo C++ Library Reference available, consult it for more details and examples of these functions.

EXPERIMENT 36

Vector Graphics with D/A Converters

REFERENCES

Hall: Chapter 10

EQUIPMENT AND MATERIALS

IBM PC or compatible computer

SDK-86 board

Power supply with ±5-V and ±12-V outputs

Digital voltmeter

Oscilloscope

Parts for circuit shown in Hall: Figure 36-1

- 2×MC1408 or DAC08 8-bit D/A
- 2×741 op amp
- 2×10-kΩ potentiometer
- 4×2.4-kΩ resistors
- 2×50-pF capacitor

INTRODUCTION

Vector-type CRT displays work well where the information to be displayed is mostly straight lines, as it is on some engineering work stations. An oscilloscope operating in its *X*–*Y* mode can be easily used to demonstrate how vector graphics are produced.

In the *X*–*Y* mode, the position of the beam spot on the screen is determined by both the voltage applied to the *X* input and the voltage applied to the *Y* input. Each pair of voltages applied to the *X* and *Y* inputs represents a position on the screen. If the voltages on the inputs are changed, the beam will sweep across the screen in a straight line from the old position to a new position. By applying a sequence of voltage pairs to the *X* and *Y* inputs, you can draw patterns on the screen.

The desired voltage for each axis of the oscilloscope can be generated under program control by connecting inexpensive IC D/A converters to microcomputer ports, as shown in Figure 36-1. To refresh a display so that it remains visible on the screen, you have to draw the display over and over.

In this experiment you will first write a program that displays a square shape on the oscilloscope. Then you will write a program that displays a square that increases in size after every 100 screen refreshes. Once the square reaches a predetermined maximum size, it is returned to its original size and the process is repeated.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write a program that uses two D/A converters to display a square on an oscilloscope.
2. Write a program that displays a square that enlarges by a specified increment after 100 screen refreshes.

PROCEDURE

1. If it is not already built, build the circuit shown in Figure 36-1. Connect the output of one D/A converter to the *X*-axis input of the oscilloscope and connect the output of the other D/A converter to the *Y*-axis input of the oscilloscope. Put the scope in *X*–*Y* display mode so that the display represents voltage versus voltage (rather than voltage versus time).
2. Connect the data input pins of the *X*-axis D/A converter to port 2A of the SDK-86 board. Connect the data input pins of the *Y*-axis D/A converter to port P2B. Connect the specified power supply voltages to the circuit.
3. To produce a horizontal or vertical line on the oscilloscope, the voltage on one axis is held constant while the voltage on the other axis is swept, over and over, through a series of values. The series of voltages is produced by cycling through a sequence of counts sent to the D/A on that axis. Because the D/A converters are 8-bit units, there are 256 possible horizontal positions and 256 possible vertical positions.

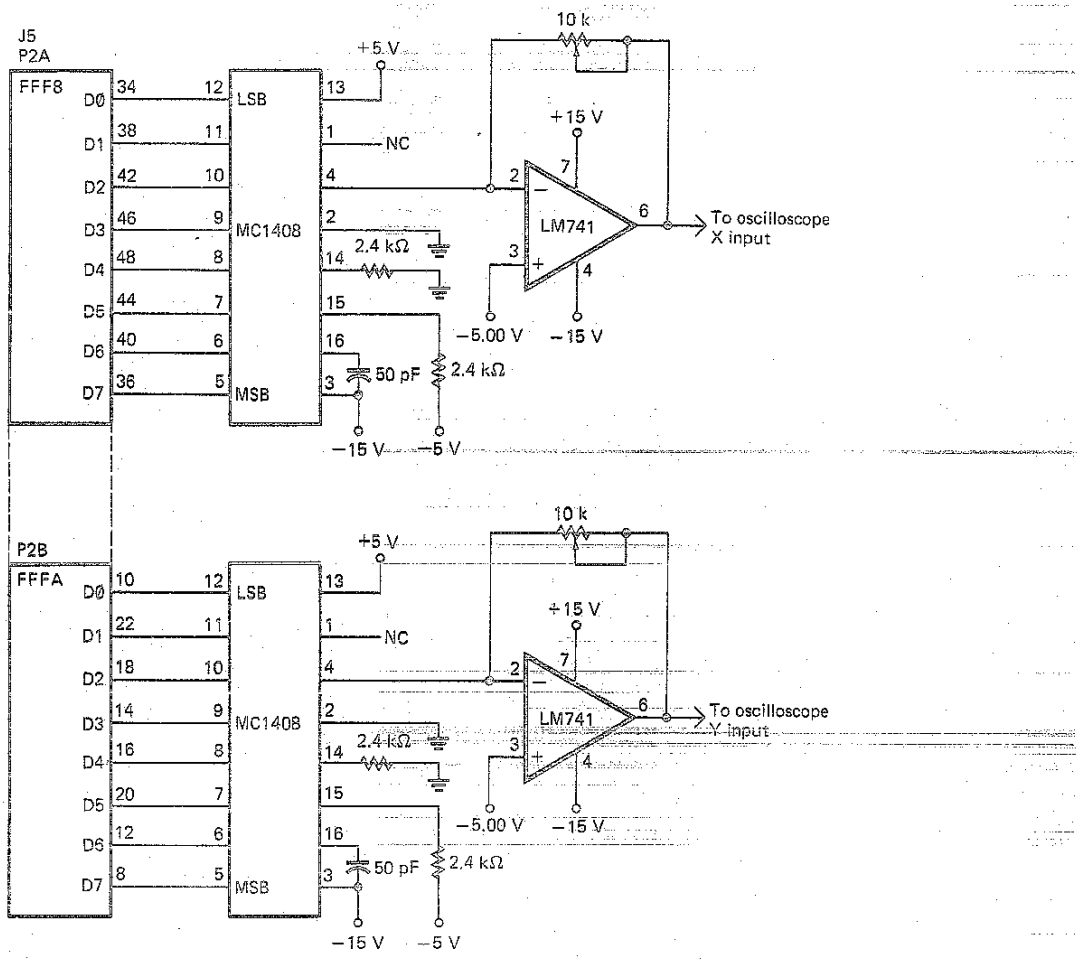


FIGURE 36-1 Two 8-bit D/A converters with voltage output used to produce vector graphics on an oscilloscope.

4. To test the hardware, write a test program that initializes the two 8255A ports, outputs a value of 00H to the vertical D/A converter, and outputs the sequence of values needed to draw a full-scale horizontal line across the screen. Run the program and adjust the volts/div control on the scope so that the line just fits on the screen. Adjust the vertical position control so that the line is positioned along the bottom of the screen.
5. Modify the test program so that it draws a full-scale vertical line on the screen. Run the program and adjust the volts/div control so the vertical line just fits on the screen. Adjust the horizontal position control so the line is positioned along the left edge of the screen.
6. The (0,0) coordinate position for the display is now located in the lower left corner of the screen. Write an algorithm for a program that will draw a square that is 100 units on a side and has one corner at the origin. Remember, the program must repeat to keep the display refreshed. Also, include a delay loop so that you can investigate, if you want, the effect of refresh rate on the display.
7. Write the assembly language program that implements your algorithm. Do not forget to initialize ports P2A (FFF8H) and P2B (FFFAH) as outputs.
8. Run your program and observe the results on the oscilloscope. You can change the size of the square by just changing the count values.
9. The next step is to put some action into the display. To do this, display in the center of the screen a square that increases in size by two counts after each 100 refreshes. When the size of the square reaches a maximum value, you will return the size to its minimum size and repeat the expansion. Write the algorithm for a program that produces this increasing square.

   HINTS: Use some graph paper to help you determine the coordinates for the center of the screen. The coordinates of this point can be thought of as the coordinates of an extremely small square. The next square will be centered on this point and two units on each side. The next square will be centered on this point and four units on each side.
10. Write the assembly language program for the enlarging-square algorithm.

11. Run your program and observe the oscilloscope screen. Note that as the size of the square increases, the rate of expansion decreases. In your report, explain why.

12. (Optional) Experiment with producing triangles, star bursts, and other interesting shapes and actions.

EXPERIMENT 37

A Text Editor—Using DOS Function Calls to Manipulate Disk Files

REFERENCES

Hall: Chapter 13

Appendix B and/or IBM DOS Technical Reference Manual

EQUIPMENT AND MATERIALS

IBM PC or compatible computer

MS DOS version 2.0 or later

INTRODUCTION

An operating system such as IBM PC DOS is, for the most part, a collection of procedures which are called to carry out DOS commands you enter on the keyboard. If DOS has been booted, these procedures are present in memory, so you can call these procedures from your programs in about the same way you call BIOS procedures. DOS has procedures for creating disk files, writing to disk files, reading disk files, printing files, and manipulating files in other ways. Also available are DOS procedures that read characters from the keyboard and/or send characters to the CRT. Appendix B gives short descriptions of a number of commonly used DOS function calls. Among these are the procedures you will need for this experiment.

This experiment shows you how to create your own very simple text editor using DOS function calls. If you were to attempt to do this with PC DOS versions before version 2.0, you would find it quite messy because, as explained in the reference, earlier DOS versions require you to manipulate parameters in a file control block (FCB) when working with files. DOS versions 2.0 and later, however, assign a 16-bit number called the *file handle* to each file when it is opened. (There is a DOS function you can call to determine the file handle that DOS has given to a file you told it to open.) Only this file handle has to be used when referring to the file for reading, writing, or other operations. DOS automatically manipulates file parameters in the file control block as needed.

As an example of how this works, you can read a file from a disk to a buffer in memory by simply loading the file handle in BX, loading the maximum number of bytes to read in CX, loading the starting address of the memory buffer in DS:DX, loading the function call number 3FH into AH, and executing the INT 21H instruction. The number of bytes actually read from the file will be returned in AX.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write a program which uses DOS function calls to create a text file and save it on disk.
2. Write a program which uses DOS function calls to read a text file from disk and display it on the CRT.
3. Write a program which uses DOS function calls to read a text file from disk and send it to a printer.

PROCEDURE

1. For the first part of this experiment, you will write a program which creates a disk file and saves a line of text, entered on the keyboard, in the disk file. Any time you write a program which uses DOS and/or BIOS calls, you should write a list of the major tasks you want to do and then read through the list of calls to find the call that will most easily do each of the tasks on your list. To help you get started, here is a list of the steps needed in your first program.

   a. Display on the CRT a message which tells the user to enter a name for the file that is to be created.
   b. Read in the file name entered by the user.
   c. Create and open a file with the given name.
   d. Prompt the user to enter a line of text followed by a carriage return.
   e. Read the line of text entered by the user into a buffer in memory and display it on the CRT.
   f. Write the line of text in the buffer to the file you created for it.

g. Close the file.
h. Return control to DOS so that you can enter new commands.

Determine and write down the number of the DOS function call that will do each of the tasks in this list most easily.

2. Each DOS function call requires that parameters such as buffer starting addresses, file handles, character counts, and the like, be passed to it in registers. Determine the buffers and parameters you need to set up for the DOS function call you chose to prompt the user for a file name, for the DOS function call you chose to read in the reply, and for the DOS function call you chose to return execution to DOS.
3. To develop a program such as this, it is best to start with the basic mainline which sets up required data structures, initializes everything, calls one or two function calls, and exits to DOS. After this basic part is tested and debugged, other calls can be added and tested in sequence to complete the program.

   Write a program which sets up a stack and any data buffer(s) needed for the first two function calls, sets up the parameters for these first two calls, calls these first two functions, and then calls the function which exits to DOS. Note that the data and parameters returned by one function call may not be in quite the format required for passing to another function call. In this case you have to make the necessary adjustments.
4. Run and test this basic program. When the basic program works correctly, determine the parameters which must be passed to each of the function calls needed to complete the program. Add the instructions which set up the parameters for each of these additional calls and the instructions which call the functions.
5. Run and test the completed program. To see if the entered line of text was actually written to a disk file, use the DOS TYPE command to display the contents of the file on the CRT.
6. The second part of this experiment is to write a program that uses DOS function calls to read a text file from disk and display it on the CRT. This is similar to the operation of the TYPE command, which you just used to test the previous part of the experiment. Here is a list of the tasks you will want to use for this program.

   a. Prompt user for name of file to read.
   b. Read in file name from user.
   c. Open file if it exists.
   d. If file does not exist, send error message to CRT and exit to DOS.
   e. Read file into buffer.
   f. Send contents of buffer to CRT.
   g. Close file.
   h. Exit to DOS.

   Determine and write down the number of the DOS function call which does each of these tasks most easily. Next to each number, summarize the buffers and parameters that must be used for each of these function calls.
7. Write a program that sets up any data buffer(s), etc., that are needed, initializes everything, and does the instructions necessary to set up for and call the sequence of DOS function calls.
8. Test and, if necessary, debug this second program.
9. The final part of this experiment is to write a simple program which sends a specified file to the printer. As discussed in the reference, the printer has a fixed file handle of 0004H, and the printer file is permanently open, so you do not need to use a DOS function call to open or close it.

   Write a task list for a simple program which sends a user-specified file to the printer and returns execution to DOS.
10. Identify the DOS function calls that can be used to accomplish each task in your list.
11. Write, test, and, if necessary, debug the program for this task list.

    NOTE: You must send a carriage return code after sending the last line of text in a file, or the printer will not print the line.
12. (Optional) The three programs you wrote for this experiment should give you an idea of how DOS function calls can be used to manipulate files. However, each of these programs had to be run individually. An obvious improvement would be to have a single program which calls one of these programs as a procedure in response to a command letter you enter on the keyboard.

    Write a simple program that prompts the user to enter a command letter followed by a carriage return and then calls one of the three procedures, based on the letter entered. Rather than returning to DOS after each procedure, the program should return to DOS only if some specific letter, such as a Q, is entered as the command.

    As further options, you might experiment with modifying the read-text procedure so that it reads in and stores text lines until the user types a <Ctrl>-D (04H).

    A more complex option you might want to explore is to write a procedure which reads in your file from disk to a buffer in memory, displays a page of text on the CRT, allows you to move the cursor around on the screen, and permits you to change the character at the cursor location. To help you with this latter part, refer to Experiment 33, which shows you how characters are stored in memory for display on the CRT. As a further help, the codes for the cursor move arrow keys are as follows:

Up arrow	—48H
Down arrow	—50H
Right arrow	—4DH
Left arrow	—4BH

EXPERIMENT 50

A Text Editor—Using C Functions

REFERENCES

Hall: Chapters 12 and 13

Borland Turbo C++ Library Reference

EQUIPMENT AND MATERIALS

IBM PC or compatible computer

DOS 2.0 or later

Borland Turbo C++ Professional Package

INTRODUCTION

As explained in the references, the libraries of most C compilers contain many predefined functions you can use to read input from the user, write text on the screen, create disk files, write to disk files, read disk files, print files, and manipulate files in other ways. This open-ended experiment gives you a chance to create your own very simple text editor using mostly predefined C function calls. The basic C functions you need for this experiment are discussed in Hall: Chapters 12 and 13. This experiment is very important because it pulls together many of the concepts from these chapters.

OBJECTIVES

At the conclusion of this experiment, you should be able to

1. Write a function that uses C function calls to create a text file and save it on disk.
2. Write a function that uses C function calls to read a text file from disk and display it on the CRT.
3. Write a function that uses C function calls to read a text file from disk and send it to a printer.
4. Implement a simple "command recognizer" that chooses one of several actions based on user pressed keys.
5. Write a program section that implements a simple "backspace"-type line editor.

PROCEDURE

1. In Hall: Chapter 13 read the section at the start of the chapter which shows how to implement a C program that reads characters from the keyboard. Carefully work your way through the program in Hall: Figure 13-3.
2. In Hall: Chapter 13 read the section which describes how predefined C functions can be used to open a disk file, write to a disk file, read from a disk file, and close a file. Carefully work your way through the program in Hall: Figure 13-42. You will build on the basic operations in this program for your program.
3. To start, create a source file containing the program in Hall: Figure 13-42 as written. Compile the program, test it, and (if necessary) debug it. This will give you a working program to experiment with.
4. The program in Hall: Figure 13-42 illustrates some very important programming techniques, but it is not very useful as an editor. It only allows you to enter one line of text; it does not allow you to edit a line after you read it into memory from disk, and the text is automatically sent to the printer, whether you want it to be or not.

   The first improvement this editor needs is a command recognizer in the mainline so you can select just the operation you want the program to perform. Use Hall: Figure 13-3 to help you write the algorithm for a simple command recognizer which allows you to use the <Ctrl> key and letter keys to specify functions such as creating a file, saving a file, reading a file into memory, or printing a file as follows:

   Ctrl O—Open a new file and close

   Ctrl P—Print a specified file

   Ctrl Q—Quit to DOS

   Ctrl R—Read a file from disk to a buffer and display

   Ctrl S—Save text in buffer to a specified file

   HINT: Basically you need to replace the gets(tp) line in Hall: Figure 13-42 with the program from Hall: Figure 13-3 and to convert each major

operation in the program from Hall: Figure 13-42 to a function so you can call it from your command recognizer as needed.

5. Add the command recognizer and function modifications to your source program. Compile, test, and (if necessary) debug the result. You now have a program which can perform a variety of file operations on command.
6. The next step is to add a program section that allows you to modify a line of text read in from disk or from the keyboard. With the simplest type of line-editor you do this by "erasing" characters as you backspace along the line from the right. You then retype the desired text for the line. This is the type of editor you will initially implement for this experiment. If time and interest permit, you can later figure out how to make a more sophisticated screen-based editor.

   To start, remember that when you read a line of text from disk to a buffer in memory and send a copy of the text to the display, the characters for the text are not only stored in the memory buffer, they are also stored in the frame buffer RAM for the display. This means that if you make any changes, you have to make them in both buffers.

   Read the section of Hall: Chapter 13 that describes how the characters for a CRT display are stored in the frame buffer RAM. Then think about and write down the actions that are needed to "erase" characters in the two buffers and move the cursor to the left as the backspace key is pressed.

   NOTE: You "erase" a character by writing a blank (20H) to the character locations in each buffer.

7. Write an algorithm for this backspace section of your program, then implement the algorithm in C and add it to your program.

   HINT: You can use the BIOS INT 10H procedures with the int86 function call, as shown in Hall: Figure 13-3, to get the cursor position, move the cursor around on the screen, and write blanks at the current cursor position. An alternative is to use the predefined C functions wherex(), wherey(), gotoxy(), and putchar() to do the job.

8. Add the required statements to a copy of your program, then compile, test, and if necessary debug the combinbed program.
9. You now have a program which allows you to read a single line of text from a file, edit the line, and write the edited line back to disk on command. This is sufficient to fulfill the requirements of this exercise, but here are a few suggestions for further improvements.

   a. Declare an array of 80 character buffers for lines so you can enter several lines of text. Each time the <Enter> key is pressed, you can move the line pointer to the start of the buffer for the next line. To edit a line in an old file, you can prompt the user to enter the line number and then put the cursor and pointer at the end of that line. Once the cursor is at the end of the line, you can use the backspace method you developed before to make corrections in that line.
   b. Use the arrow keys to move the cursor to the position of the incorrect letter on the screen. Next use the cursor coordinates to calculate the location of the letter in the frame buffer and in the memory buffer. Finally, modify the contents of the two buffers as desired. If you implement a "delete character at cursor" operation and an "insert character at cursor operation," you can correct errors without erasing and retyping the right end of a line. Note that if you delete more characters than you add or add more characters than you delete, you have to move the characters in the remainder of the line accordingly. Consult Herbert Schildt's *Advanced Turbo C* (Borland-Osborne/McGraw-Hill, 1987) for an example of how to do this.
   c. To develop a full "screen-oriented" editor, you need to store the characters in a complex data structure called a double-linked list. This structure allows you to more easily insert or delete characters or blocks of text. Consult John Ogilvie's *Advanced C Struct Programming* (Wiley, 1990) for an explanation of how this is done.

EXPERIMENT 39

SDK-386 Programming

REFERENCES

Hall: Chapter 15

EQUIPMENT AND MATERIALS

IBM PC or compatible computer

URDA SDK-386 board

DOS 2.0 or later

Borland Turbo Assembler or other assembler that can assemble 80386 code

INTRODUCTION

As explained in Hall: Chapter 15, the URDA SDK-386 board operates its 80386 in protected mode. After reading the discussion of protected-mode operation in Hall, the thought of writing a program for a protected-mode system might be a little scary to you, but the SDK-386 makes it relatively easy.

First, the monitor program on the SDK-86 sets up all the required descriptor tables and initializes almost everything you need for many programs. Second, the SDK-386 operates in "flat" protected mode so all the segments start at absolute address 00000000H and extend to FFFFFFFFH, the full addressing range of the 80386. This means that you can access any code or data word with a simple 32-bit offset. Third, the SDK-386 board has a good debugger to help you step through and debug your programs.

The major difference between programs for the SDK-386 board and programs you may have written for the SDK-86 board is the way interrupts are handled. As explained in Hall: Chapter 15, an 80386 in protected mode uses an interrupt-descriptor table instead of a simple interrupt-vector table to access interrupt procedures. We show you an example of how to use the interrupt descriptor table, so this should not be a problem.

The purpose of this experiment is to show you how to write some simple assembly language programs which you can download to the SDK-386, run, and debug. With the practice gained here and in the preceding experiments, you should be able to program the SDK-386 to do almost anything you want it to.

At the conclusion of this experiment you should be able to

1. Download, run, test, and debug programs on an SDK-386 board.
2. Write a 386 program that calls a monitor procedure to display an incrementing count on the LCD display.
3. Write a program that initializes the interrupt descriptor for a specified interrupt type.
4. Write a program that counts user-entered interrupts and displays the result on the LCD of the SDK-386 board.

PROCEDURE

1. The SDK-386 display procedure is located in EPROM at address FF9678H. This procedure uses the ASCII codes stored in the display buffer starting at address 000002ECH to produce the display. Up to 17 characters can be displayed on the LCD, so the buffer is 17 bytes long. The program in Figure 39-1 shows how you can copy a message to the display buffer and call the monitor procedure to display it. Work your way carefully through the program and note anything that seems new. In particular, note the following.
   a. The .386 at the top of the program tells the assembler that the program uses 386 instructions and 32-bit addressing
   b. The user RAM on the board starts at 300H. The ORG directive tells the assembler to start the program code at 300H and compute offsets so that the program will execute correctly when loaded into memory at that address.
   c. All the segments start at the same physical address, so for simplicity we put everything in one segment.
   d. We declared an uninitialized buffer at 000002ECH so we could refer to the display buffer by name.

```
%PAGESIZE 66,132
; Test program for URDA SDK-386 board
; Displays a welcome message on the LCD

.386
CODE SEGMENT USE32
        ASSUME CS:CODE; DS:CODE; ES:CODE

        ORG 300H                        ; Start of user RAM

        MOV ESI, OFFSET MESSAGE         ; Point at message
        MOV EDI, OFFSET DISPLAYBUF      ; Point at display buffer
        MOV ECX, 17                     ; Number of characters
        CLD                             ; Increment pointers
        REP MOVSB                       ; Copy string to display buffer
        CALL LCD                        ; Use monitor display procedure
        MOV ECX, 0FFFFFFFH
CNTDN:  LOOP CNTDN

AGAIN:  NOP
        JMP AGAIN                       ; Endless loop, press Break key
                                        ;    to exit

        MESSAGE DB 'Hello World     '

        ORG 000002ECH                   ; Access to monitor
        DISPLAYBUF DB 17 DUP(?)         ;    display buffer

        ORG 00FF9761H
        LCD LABEL NEAR                  ; Label for monitor display
                                        ;     procedure

CODE ENDS
END
```

FIGURE 39-1 Test program to display a message in the LCD of an SDK-386 board.

e. We declared a label called LCD at the starting address of the display procedure in EPROM so we could call the procedure by name.

2. Create a source file containing the test program in Figure 39-1. To assemble the file type TASM filename,, and press the <Enter> key. To link the program type Tlink /3 filename,, and press the <Enter> key. To produce the binary file that can be downloaded, type EXE2BIN filename and press the <Enter> key.
3. Print a copy of your list file to help follow the execution of the program.

NOTE: If you execute the command MODE LPT1:132 before you execute the PRINT command, your list file should all fit on the page.

4. In the SDK-386 manual read the descriptions of the operations performed by each of the keyboard keys. Keep a copy of these descriptions handy for future reference.
5. Make sure the proper RS-232C cable is connected from the COM1 serial port on your computer to the J5 connector on the SDK-386 board. Power up the SDK-386 board and press the <DNLD> key. When prompted for the file type press the <B> key on the keypad and then the <+> key. When prompted for the starting address use the keypad to enter 300, then press the <+> key on the keypad. When prompted for the Ending address, enter 400 and press the <+> key. The SDK-386 is now waiting to receive data.
6. On your computer type

   dnload filename.bin/b

   and press the <Enter> key.
7. When the file transfer is complete, the display on the SDK-386 board will show the ending address for the code and data bytes that were transferred. Press the <ADDR> key on the SDK-386 keypad and when prompted for an address enter 00000300 and press the <+> key. The display should show the first byte of your program.

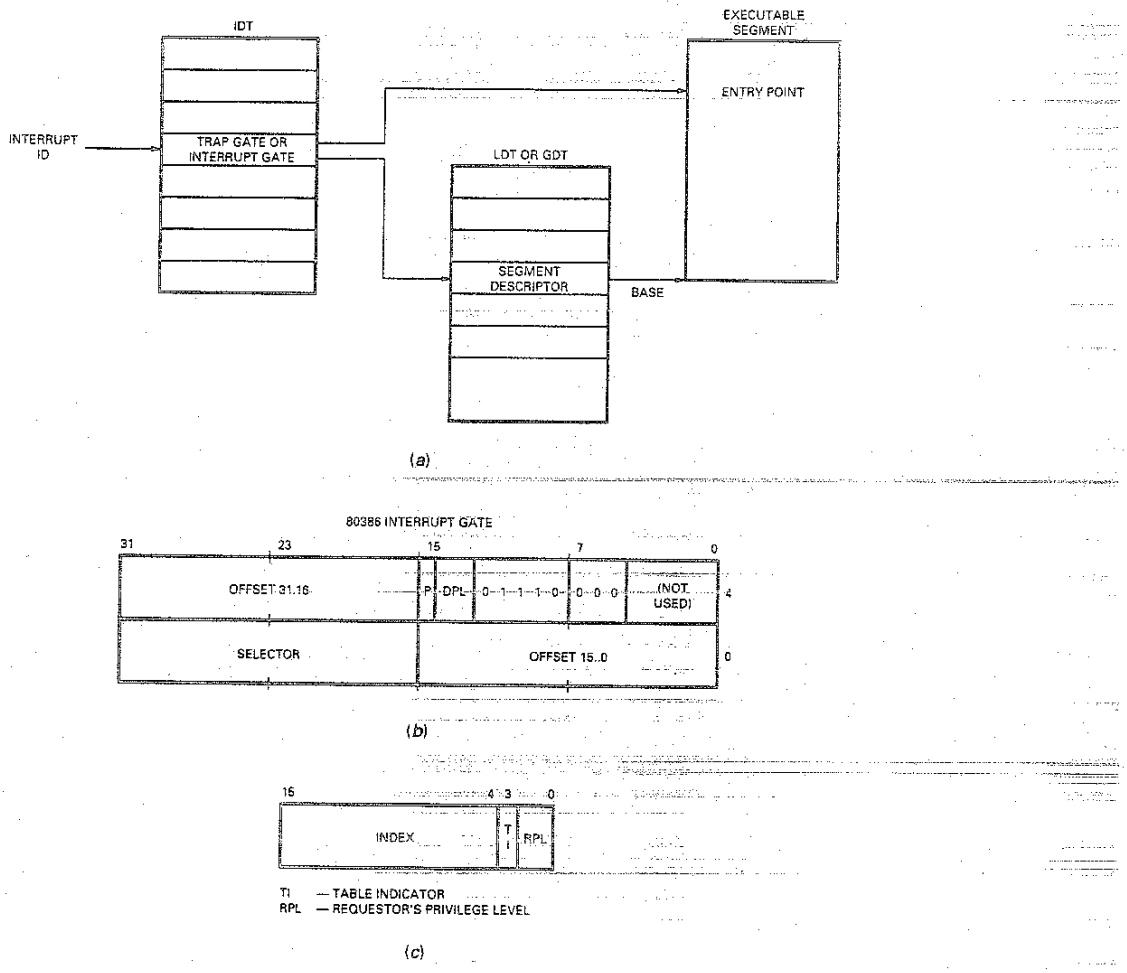


FIGURE 39-2 80386 protected-mode interrupts. (*a*) Protected-mode mechanism to access an interrupt-service procedure. (*b*) 80386 interrupt-gate-descriptor format. (*c*) Format for selector in descriptor.

8. Press the <+> key to step through more bytes of your program. When you tire of this, set the address back to 300 and press the <RUN> key to execute your program. The message should appear on the LCD. To terminate the program press the RESET pushbutton or the BREAK pushbutton to the left of the keypad.
9. Now that you have the basic operations down, rewrite the program in Figure 39-1 so that it outputs an incrementing hexadecimal count to the LCD. Assemble, link, convert, download, and test your program.

NOTE: The 8086 address, data, and control bus signals are available on the connectors along the left edge of the board, so if you want to look at the 80386 bus signals, you can connect a logic analyzer to these pins and observe them while this program runs.

10. As we said in the introduction, the 80386 operating in protected mode uses an interrupt descriptor table to send execution to an interrupt procedure. Figure 39-2*a* shows in diagram form how this works. It is very similar to the indirect method described for a CALL gate in Hall: Chapter 15.

The interrupt type is multiplied by 8 and used to point to a descriptor in the interrupt descriptor table. Part of the data in this descriptor is a selector which points to a code segment descriptor in the global descriptor table (GDT) or the local descriptor table (LDT). The descriptor in the GDT or LDT contains the base address for the desired segment. Another part of the information in the interrupt descriptor in the IDT is the offset of the desired instruction in the destination segment. The 80386 adds the base address from the GDT or LDT descriptor to the offset from the IDT descriptor to produce the physical address of the interrupt procedure. Figure 39-2*b* shows the format for the descriptor that we need to load into the interrupt descriptor table and Figure 39-2*c* shows the format for the selector in this descriptor. This sounds terrible,

```
%PAGESIZE 66,132
;EXPERIMENT 39 FIGURE 3
; PROGRAM TO DEMONSTRATE USE OF INTERRUPTS ON SDK-386 BOARD

.386
CODE SEGMENT USE32
        ASSUME CS:CODE, DS:CODE, ES:CODE

        ORG 300H

        MOV EBX, OFFSET INT32COUNT ; Get PROC offset
        MOV LOWOFFSET, BX          ; Low word to IDT frame
        ROR EBX, 16                ; High word to BX
        MOV HIGHOFFSET, BX         ; High word to IDT frame

; Copy IDT frame built here to Interrupt Descriptor Table

        MOV ESI, OFFSET IDTFRAME ; Point at IDT frame
        MOV EDI, OFFSET INT32DESCRIPTOR ;
        MOV ECX, 04H               ; Number of words to copy
        CLD
        REP MOVSW                  ; Copy to IDT
        MOV EDX, 0                 ; Initialize register counter
        MOV COUNT, 0               ; Initialize memory counter

HERE:   INT 32
        NOP
        NOP
        NOP
        JMP HERE

        PROC INT32COUNT
        INC EDX                    ; Increment reg counter
        INC COUNT                  ; Increment mem counter
        MOV ECX, 000FFFFFH
AGAIN:  LOOPD AGAIN                ; Wait for a while
                                   ; (With INTR input, give
                                   ;   time for signal to go low)
        IRETD                      ; 32 bit return
        ENDP

        COUNT DD 0                 ; Variable count in memory

IDTFRAME:
        LOWOFFSET DW ?             ; Space for low 16 bits of
                                   ;  Procedure offset
        SELECTOR DW 08H            ; Selector which points to
                                   ;  flat code descriptor in GDT
        CONTROL DW 1000111000000000B
                                   ; Seg present, DPL=0, fixed bits
        HIGHOFFSET DW ?            ; Space for high 16 bits of
                                   ;  Procedure offset

        ORG 000001A0H
            INT32DESCRIPTOR LABEL WORD

        CODE ENDS
        END
```

FIGURE 39-3 386 program that demonstrates protected-mode interrupt vector table initialization and interrupt procedures.

but it is really not hard to implement on the SDK-386 board.

Figure 39-3 shows a simple program which initializes the interrupt descriptor table as needed to call an INT 32 interrupt procedure. The procedure increments a count in memory and a count in a register each time the INT 32 instruction executes. Here's how it works.

In our program we declare and initialize a data structure called IDTFRAME. These locations are used to assemble the four words needed for the descriptor. The first four instructions in the program load the 32-bit offset of the interrupt service procedure into the appropriate locations in IDTFRAME. Next we use a string move instruction to copy the contents of IDTFRAME to the actual interrupt descriptor table at offset 000001A0H in RAM.

Finally, we execute the INT 32 instruction to call the procedure. The interrupt procedure increments a couple of counts, delays for a while, and then returns to the mainline. Once in the mainline it executes the INT 32 instruction again.

NOTE: An INT instruction is a good way to test an interrupt procedure that will eventually be used to service a hardware interrupt. When the procedure works correctly with the INT instruction, you can remove the INT instruction, insert instructions to enable the hardware interrupt input, and see if the program works correctly with an external interrupt signal.

Carefully study the program in Figure 39-3 until you understand the function of each directive and instruction.

11. Create a source file containing the program in Figure 39-3, then assemble, link, convert, download, run, and test the program.

NOTE: To check the count press the <BREAK> key and look at the EDX register contents and the COUNT memory location.

12. Pressing the <USER INT> key on the SDK-386 board generates a type-32 interrupt. Modify the program from Figure 39-3 so that it counts the number of times you press the <USER INT> key and displays the count on the LCDs.

EXPERIMENT 40

Putting It All Together—386 Programming on a PC

REFERENCES

Hall: Chapters 5, 11–15

EQUIPMENT AND MATERIALS

386-based PC

DOS 2.0 or later

Borland Turbo Assembler or other assembler that can assemble 80386 code

Borland Turbo C++ Professional Package

INTRODUCTION

As mentioned in Hall: Chapter 15, writing programs for a 386-based PC operating in protected mode requires special tools and is beyond the goals of this manual. With the tools you have learned to use in this manual, however, you can write real-mode programs which take advantage of the advanced instructions and addressing modes of the 386 described in Hall: Chapter 15. One way of doing this open-ended experiment gives you a chance to work with some of these advanced instructions.

PROCEDURE

A section at the end of Hall: Chapter 5 shows you how to write a simple recursive procedure to calculate the factorial of a number, and a problem at the end of Hall: Chapter 5 discusses how to do it without recursion. For this experiment you will write a program which prompts the user to enter a positive integer, calculates the factorial of the number, and outputs the computed factorial to the screen. The challenge here is write the program so that it produces a correct result as quickly as possible for the largest-possible input number.

You can implement this program on a PC in any way you want. As indicated by the title, you might write an assembly language program which takes advantage of the large registers and the large memory addressing range of the 80386. As another approach you might decide to use an 8087 math coprocessor to perform the calculations. As still another approach, you might decide to write the program in C and let the compiler do much of the work. Whichever method you choose, you should try to be creative and to use as much as possible of what you have learned from the previous experiments. For example, a little graphics might be a nice touch. If the result becomes too large to hold in memory at one time, you might have to write it to a disk file. Also, if the result is too large to display on the screen, you might have to format it in pages and send it to a printer. We hope that this project will give you some fun and show you how much you have gained.

APPENDIX A

Directions for Configuring an SDK-86 Board for Downloading Programs

HARDWARE

1. Install SDK-86 jumpers W1, W2, W3, W4, and W5.
2. Install jumper W25 to change the baud rate to 4800 baud.
3. Set the wait-state jumper W28 for one wait state.
4. Make up the RS-232C cable as shown in Hall: Figure 13-10*b*.
5. Plug the RS-232C cable into the IBM PC serial board and into the J7 plug on the SDK-86.
6. Make sure that –12 V as well as +5 V and ground are connected to the board.

SOFTWARE

We assume that you are using an IBM PC or PS/2-type computer with a hard disk. Make sure the hard disk contains the following files and that your path command makes them accessible.

MODE.COM	from DIS Utilities disk
PRINT.COM	from DOS disk
DEBUG.COM	from Dos Utilities disk
EXE2BIN.EXE	from DOS Utilities disk
TASM, TLINK	from Borland
or	
MASM, LINK	from Microsoft
SDKCOM1.EXE	from available Hall: Disk-2 or from Hall: Figure 14-27

An editor of your choice

HINTS

Experiment 8 describes in detail the use of SDKCOM1 to download and run programs on an SDK-86 board. The only major problems we have seen people have with this process were caused by failing to note the following.

1. You must press the RESET key on the SDK-86 and enter the SDK-86 command G FE00:0 to run the serial monitor program so the board will communicate with the PC. You have to do this only when you turn on the power or after you press RESET to get control if a program locks up the SDK-86.
2. When you write a program that you want to download to an SDK-86 board, you must leave out the START label after the final END statement in the program. (Remember that this start label is required if you are going to run the program in the PC.)

APPENDIX D

IBM DOS (INT 21H) Function Calls

If you require further explanation of any of these function calls, refer to the *IBM DOS Technical Reference Manual*, Chapter 5 (DOS Interrupts and Function Calls).

NOTES: An ASCIIZ string is a string with the last byte equal to zero. File handle for printer = 0004. No need to open printer file. Place function number in AH, other data in registers as required, and then execute the INT 21H instruction. Registers used to pass parameters are altered; other registers are not changed.

KEYBOARD INPUT AH = 1H
Waits for one character to be typed, displays character on screen, and returns character in AL.

DISPLAY OUTPUT AH = 2H
The character in DL is output to the screen.

AUXILIARY INPUT AH = 3H
Waits for character from Asynchronous Communications adapter, returns character in AL.

AUXILIARY OUTPUT AH = 4H
The character in DL is output to the Asynchronous Communications adapter.

PRINTER OUTPUT AH = 5H
The character in DL is output to the printer.

DIRECT CONSOLE I/O AH = 6H
If DL = FF, AL returns with ZF = 0 and character input from keyboard. If the character is not ready, then ZF = 1. If DL<>FF, then DL is assumed to have valid character to output to screen.

DIRECT CONSOLE INPUT AH = 7H
Waits for character from keyboard. Returns with character in AL. No display on screen.

CONSOLE INPUT NO ECHO AH = 8H
Identical to AH = 1 except the key is not displayed. Returns with character input in AL.

PRINT STRING ON CRT AH = 9H
Requires DS:DX to point to character string in memory whose last character is 24H.

BUFFERED KEYBOARD INPUT AH = 0AH
Requires DS:DX to point to input buffer. Byte 1 of buffer specifies number of characters buffer can hold (not 0). Characters read from keyboard until <RET>. Characters placed in buffer beginning at byte 3. Byte 2 in buffer set to number of characters read, excluding the <RET>. The <RET> (ODH) is always last character in string. Bell will ring if too many characters are entered.

GET DATE AH = 2AH
Returns date in CX:DX. CX = year in binary, DH = month (1 = JAN), DL = day. See DOS reference manual.

SET DATE AH = 2BH
Requires date in CX:DX. If date valid, returns AH = 0; otherwise AH = FF. See DOS reference manual.

CREATE A FILE AH = 3CH
Requires DS:DX to contain address of an ASCIIZ string with the drive, path, and filename. CX = file attribute = 0 for read/write. Creates a new file or truncates an old file to zero length in preparation for writing. File is opened for read/write and file handle is returned in AX.

OPEN A FILE AH = 3DH
Requires DS:DX to contain address of ASCIIZ string with drive, path, and filenames. AL = access code (00 = read only; 01 = write only; 02 = read/write). On return, AX = file handle for the file; or, if carry flag set, AX = error code.

CLOSE FILE AH = 3EH
Requires BX = file handle. On return, the file will be closed and all internal buffers flushed.

READ FROM FILE OR DEVICE AH = 3FH
Requires BX = file handle. CX = number of bytes to read. DS:DX = buffer address. On return AX = number of bytes read, if AX = 0, then program has tried to read from the end of the file. See DOS reference manual.

WRITE TO A FILE OR DEVICE AH = 40H

Requires BX = file handle, CX = number of bytes to write, DS:DX points to buffer holding data to be written to file. See DOS reference manual.

TERMINATE A PROCESS (EXIT) AH = 4CH

If AL = 00, control is returned to DOS.

ERROR MESSAGES FOR FUNCTION CALLS

Many of the function calls return with the carry flag = 0 if the operation is successful and return with the carry flag = 1 if an error occurred in the call. If carry = 1, then AX will contain one of the following hex codes:

Code	Condition
1	Invalid function number
2	File not found
3	Path not found
4	Too many open files (no handles left)
5	Access denied
6	Invalid handle
7	Memory-control blocks destroyed
8	Insufficient memory
9	Invalid memory block address
10	Invalid environment
11	Invalid format
12	Invalid access code
13	Invalid data
15	Invalid drive was specified
16	Attempted to remove current directory
17	Not same device
18	No more files

APPENDIX C

Summary of C Graphics Functions

(Function headers in graphics.h)

```
arc(int x, int y, int startangle, int endangle, int radius)
    /* Angles in degrees, going counterclockwise from 3 o'clock. */

bar(int left, int top, int right, int bottom)
    /* Draws filled-in bar. */

bar3d(int left, int top, int right, int bottom, int depth, int topflag)
    /* Three-dimensional bar, top drawn if topflag = 1. */

circle(int x, int y, int radius)

drawpoly(int numpoints, int far *polypoints)
    /* Draw polygon, pass number of vertices +1, and pointer to array containing coordinates of
        vertices. Coordinates of starting vertex must be included twice to close figure. */

ellipse(int x, int y, int startangle, int endangle, int xradius, int yradius)
    /* Draw an elliptical arc; ellipse if startangle = endangle. */

fillellipse(int x, int y, int xradius, int yradius)
    /* Draw and fill ellipse with current fill color and patterns. */

fillpoly(int numpoints, int far *polypoints)
    /* Draw and fill polygon. See drawpoly for explanation. */

floodfill(int x, int y, int border)
    /* x and y are coordinates of a point in the area to be filled; border is the color of border
        surrounding area. */

getimage(int left, int top, int right, int bottom, void far *bitmap)
  /* Copies an image from screen to a buffer in memory. */

imagesize(int left, int top, int right, int bottom)
  /* Returns the number of bytes required to store image. */

initgraph(int far *graphdriver, int far *graphmode, char far *pathodriver)
  /* Note: must pass pointers to driver, mode, and path string.

        int driver = CGA, mode = CGACO; /* See accompanying figures for options. */
        char pathtodriver[] = "C:\\TC\\BGI";

        initgraph(&driver, &mode, pathtodriver);
```

graphics_drivers

constant	Numeric value
DETECT	0 (requests autodetection)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Graphics driver	*graphics_modes*	Value	Column × row	Palette	Pages
CGA	CGAC0	0	320 × 200	C0	1
	CGAC1	1	320 × 200	C1	1
	CGAC2	2	320 × 200	C2	1
	CGAC3	3	320 × 200	C3	1
	CGAHI	4	640 × 200	2 color	1
MCGA	MCGAC0	0	320 × 200	C0	1
	MCGAC1	1	320 × 200	C1	1
	MCGAC2	2	320 × 200	C2	1
	MCGAC3	3	320 × 200	C3	1
	MCGAMED	4	640 × 200	2 color	1
	MCGAHI	5	640 × 200	2 color	1
EGA	EGALO	0	640 × 200	16 color	4
	EGAHI	1	640 × 350	16 color	2
EGA64	EGA64LO	0	640 × 200	16 color	1
	EGA64HI	1	640 × 350	4 color	1
EGA-MONO	EGAMONOHI	3	640 × 350	2 color	1*
	EGAMONIHI	3	640 × 350	2 color	2**
HERC	HERCMONOHI	0	720 × 348	2 color	2
ATT400	ATT400C0	0	320 × 200	C0	1
	ATT400C1	1	320 × 200	C1	1
	ATT400C2	2	320 × 200	C2	1
	ATT400C3	3	320 × 200	C3	1
	ATT400MED	4	640 × 200	2 color	1
	ATT400HI	5	640 × 200	2 color	1
VGA	VGALO	0	640 × 200	16 color	2
	VGAMED	1	640 × 350	16 color	2
	VGAHI	2	640 × 480	16 color	1
PC3270	PC3270HI	0	720 × 350	2 color	1
IBM8514	IBM8514HI	1	1024 × 768	256 color	
	IBM8514LO	0	640 × 480	256 color	

* 64K on EGAMONO card
** 256K on EGAMONO card

```
line(int x1, int y1, int x2, int y2)
  /* Draws a line between (x1,y1) and (x2,y2) */
```

setfillstyle(int pattern, int color)
/* Pattern from following chart, color from color chart. */
setfillstyle sets the current fill pattern and fill color. To set a user-defined fill pattern, do *not* give a *pattern* of 12 (USER_FILL) to setfillstyle; instead, call setfillpattern. The enumeration *fill_patterns*, defined in graphics.h, gives names for the predefined fill patterns, plus an indicator for a user-defined pattern.

Name	Value	Description
EMPTY_FILL	0	fill with background color
SOLID_FILL	1	solid fill
LINE_FILL	2	fill with—
LTSLASH_FILL	3	fill with ///
SLASH_FILL	4	fill with ///, thick lines
BKSLASH_FILL	5	fill with \\\, thick lines
LTBKSLASH_FILL	6	fill with \\\
HATCH_FILL	7	light hatch fill
XHATCH_FILL	8	heavy cross-hatch fill
INTERLEAVE_FILL	9	interleaving line fill
WIDE_DOT_FILL	10	widely spaced dot fill
CLOSE_DOT_FILL	11	closely spaced dot fill
USER_FILL	12	user-defined fill pattern

setlinestyle(int linestyle, unsigned upattern, int thickness)
/* Linestyle from following chart; upattern = 0, width from following chart. */

linestyle specifies in which of several styles subsequent lines will be drawn (such as solid, dotted, centered, dashed). The enumeration *line_styles*, defined in graphics.h, gives names to these operators:

Name	Value	Description
SOLID_LINE	0	solid line
DOTTED_LINE	1	dotted line
CENTER_LINE	2	centered line
DASHED_LINE	3	dashed line
USERBIT_LINE	4	user-defined line style

thickness specifies whether the width of subsequent lines drawn will be normal or thick.

Name	Value	Description
NORM_WIDTH	1	1 pixel wide
THICK_WIDTH	3	3 pixels wide

upattern is a 16-bit pattern that applies only if *linestyle* is USERBIT_LINE (4). In that case, whenever a bit in the pattern word is 1, the corresponding pixel in the line is drawn in the current drawing color. For example, a solid line corresponds to a upattern of 0xFFFF (all pixels drawn), while a dashed line can correspond to a *upattern* of 0x3333 or 0x0F0F. If the *linestyle* parameter to setlinestyle is not USERBIT_LINE (!=4), the *upattern* parameter must still be supplied, but it is ignored.

NOTE: The *linestyle* parameter does not affect arcs, circles, ellipses, or pieslices. Only the *thickness* parameter is used.

setviewport (int left, int top, int right, int botton, int clip)
/* Set size of active graphics window. See Hall: Figure 13-23.

GLENCOE
Macmillan/McGraw-Hill

ISBN 0-07-025743-4
90001>
9 780070 257436