MICROPROCESSING FUNDAMENTALS

SEMINAR WORKBOOK

A SHORT COURSE FOR

SCIENTISTS AND ENGINEERS

BY

RAYMOND N. BENNETT

AND

JOHN STOCKSDALE

AMERICAN INSTITUTE

FOR

PROFESSIONAL EDUCATION

CARNEGIE BUILDING
HILLCREST ROAD
MADISON, N.J. 07940

12/1/79

PREFACE

As in learning to drive a car, a microprocessor must be practiced. You cannot really learn how to use one just from reading books alone. This course includes a microcomputer and more information than can be covered in a three-day seminar; because it is the authors' purpose to give you sufficient background, written material, and hardware to be able to design a microcomputer system. BUT THIS CANNOT happen if the student does not study ALL the information given with the course and build up a system using the Sym-I.

ANALYZING SOFTWARE PROBLEMS (CONTINUED)

THE HARDWARE/SOFTWARE APPROACH TO MICROCOMPUTER DESIGN

APPENDICES

COURSE OUTLINE

COURSE OUTLINE

1. Introduction to Microprocessors and Microcomputers
   - A. Terminology
   - B. Architecture
   - C. Number Systems
   - D. Hardware and Software

2. Operating a Microcomputer: The SYM-I
   - A. Examining and Modifying Memory
   - B. Loading and Running Sample Programs
   - C. Using the Debug Mode

3. Microcomputer Architecture and Elementary Programming
   - A. Block Diagram of CPU
   - B. Data Bus, Address Bus, and Control Lines
   - C. Memory and I/O Addressing
   - D. Selected SYM-I Monitor Calls
   - E. A Selected Subset of Instructions

4. Programming Examples
   - A. Parallel Data Input and Output
   - B. Use of the SYM-I Keyboard and Display

5. First Night: Overnight Assignment

6. Interfacing Microcomputers to External Devices
   - A. Using Programmable I/O Lines for Device Control
   - B. Device Control Software Techniques
   - C. Common Interface Devices

7. Further Software
   - A. Flags and Conditional Branches
   - B. Counting and Timing Loops

8. Second Night: Overnight Assignment

9. Advanced Software
   - A. Binary and Decimal Arithmetic
   - B. Indexed Addressing
   - C. Indirect Addressing

10. Interval Timers and Interrupts
    - A. Using an Interval Timer for Time Delays
    - B. The 6502 Interrupt System
    - C. Interval Timer Triggered Interrupts
    - D. Interrupt Applications

11. Serial Data Input and Output
    A. The SYM-I Serial I/O System
    B. 20 mA Current Loop and RS-232 Interfaces
    C. The ASCll Code

12. Introduction to System Expansion

13. Introduction to Applications

READING ASSIGNMENTS

READING ASSIGNMENTS

The greatest gain from a short course such as this one can be realized if you are willing to participate fully by giving your undivided attention during the presentations and experiments as well as preparing yourself during the time between the sessions.

The readings recommended below are only a sample selected to point you in the correct direction to facilitate learning during these three days.

	SYM REFERENCE MANUAL	SYM HARDWARE MANUAL	SYM PROGRAM MANUAL	SEMINAR MANUAL
FIRST NIGHT	Chapters 1 and 2; Chapter 4, sections 4.1 to 4.3.3, 4.3 to 4.3.4, and tables 4.4 and 4.5		Pages 1-5 and 20-49	Basic Logic
SECOND NIGHT	Chapters 6 and 8, Appendix J	Pages 15-28	Chapters 5 and 7, Pages 129-143 (secondary) Chapter 6	

After the completion of the three day course, it is recommended that you pursue the study of your microcomputer system in the following order: First, read the SYM Reference Manual to gain an expert working knowledge of the machine that you are experimenting with. Second, read the SYM Hardware Manual to gain a knowledge of the strengths and weaknesses of the 6500 family and the monitor. Third, after you are familiar with all aspects of the system and how to use it, then you are ready to explore the programming of the Micro System via the SYM Programming Manual.

EXPERIMENTS

SYM EXPERIMENTS

INTRODUCTION

1. Electrical Connections:
   Connect the 44 pin edge connector-harness to the applications connector. It is labeled "A connector" on the board. The black and red power wires should face the "P connector" at the top center of the board.

2. Power Up:
   Apply power to the SYM by plugging in the power supply. The LED mounted on the upper edge of the board, between the A and P connectors, will come on. You will then hear a "beep" from the round transducer located at the lower center of the board. The seven-segment LEDs will remain dark.

3. CR ?
   The SYM command syntax has been designed to allow operation of the monitor from an ASCII keyboard based terminal or the on board keyboard with no change in syntax.
   Like many operating systems which use an ASCII keyboard as the input device the SYM allows entry of a command from the keyboard but witholds execution until a carriage return (CR) is input. After execution of a command the SYM will output a period (.) as a prompting character. This indicates the SYM is ready for a new command.
   The SYM follows this sequence whether it is using a terminal or the on board keyboard display. Note the presence of the CR key on the keyboard to facilitate this.

4. Initialization:
   Press the carriage return (CR) key. A "beep" will be heard to indicate a key has been depressed.
   The display will now output, "SY1.0..", to indicate the SYM is successfully communicating with the on board keyboard.
   If in the future one of your programs should run afoul or if you simply want the monitor to take control, you may do so by simply pressing the reset (RST) key followed by the (CR) key.

5. Examining Memory:

To examine the data contained at any given memory location use the following procedure as an example.

YOU KEY IN:	DISPLAY	COMMENTS
(RST)	blank	
(CR)	SY1.0..	*sym now ready for
(MEM)	1.0..m	command
(0),(0),(0),(0)	m 0000	
(CR)	0000.XX	*XX indicates the
(→)	0001.XX	data found as the
(→)	0002.XX	contents of the
(→)	0003.XX	memory location
(→)	0004.XX	displayed in the
(SHIFT),(←)	0003.XX	left four digits
(SHIFT),(←)	0002.XX	*may advance or
(SHIFT),(←)	0001.XX	retreat one location
(SHIFT),(←)	0000.XX	by (→) and (←) keys
(SHIFT),(+)	0008.XX	*may advance or
(SHIFT),(+)	0010.XX	retreat eight locs.
(-)	0008.XX	by use of (+) and
(-)	0000.XX	(-) keys

6. Entering Data:

To enter the data you desire to have stored at a particular location use the procedure demonstrated above to specify the location. Then press the two keys corresponding to the data you wish stored there.

Follow the example below:

YOU KEY IN:	DISPLAY	COMMENTS
from above	0000.XX	*let's change XX to $54
(5)	000.XX.5	
(4)	0001.XX	*loc. 0000 now contains $54
(SHIFT),(←)	0000.54	*examine loc. 0000
(→)	0001.XX	
(5)	001.XX.5	
(5)	0002.XX	*$0001 now contains $55
(5),(6)	0003.XX	*$0002 " " $56
(5),(7)	0004.XX	*$0003 " " $57
(5),(8)	0005.XX	*$0004 " " $58
(5),(9)	0006.XX	*$0005 " " $59

YOU KEY IN:	DISPLAY	COMMENTS
(5),(A)	0007.XX	*$0006 now contains $5A
(5),(B)	0008.XX	*$0007 " " $5B
(5),(C)	0009.XX	*$0008 " " $5C
(-)	0001.55	*check
(→)	0002.56	* "
(→)	0003.57	* "
(→)	0004.58	* "
(→)	0005.59	* "
(→)	0006.5A	* "
(→)	0007.5B	* "
(→)	0008.5C	* "

NOTE:

All addresses and data are represented with the hexadecimal numbering system.

EXPERIMENT 1

1. **Enter The Program Below:**

   Enter the machine code from left to right. Each line may contain one, two, or three bytes, depending on the machine instruction they represent.

2. Run The Program:

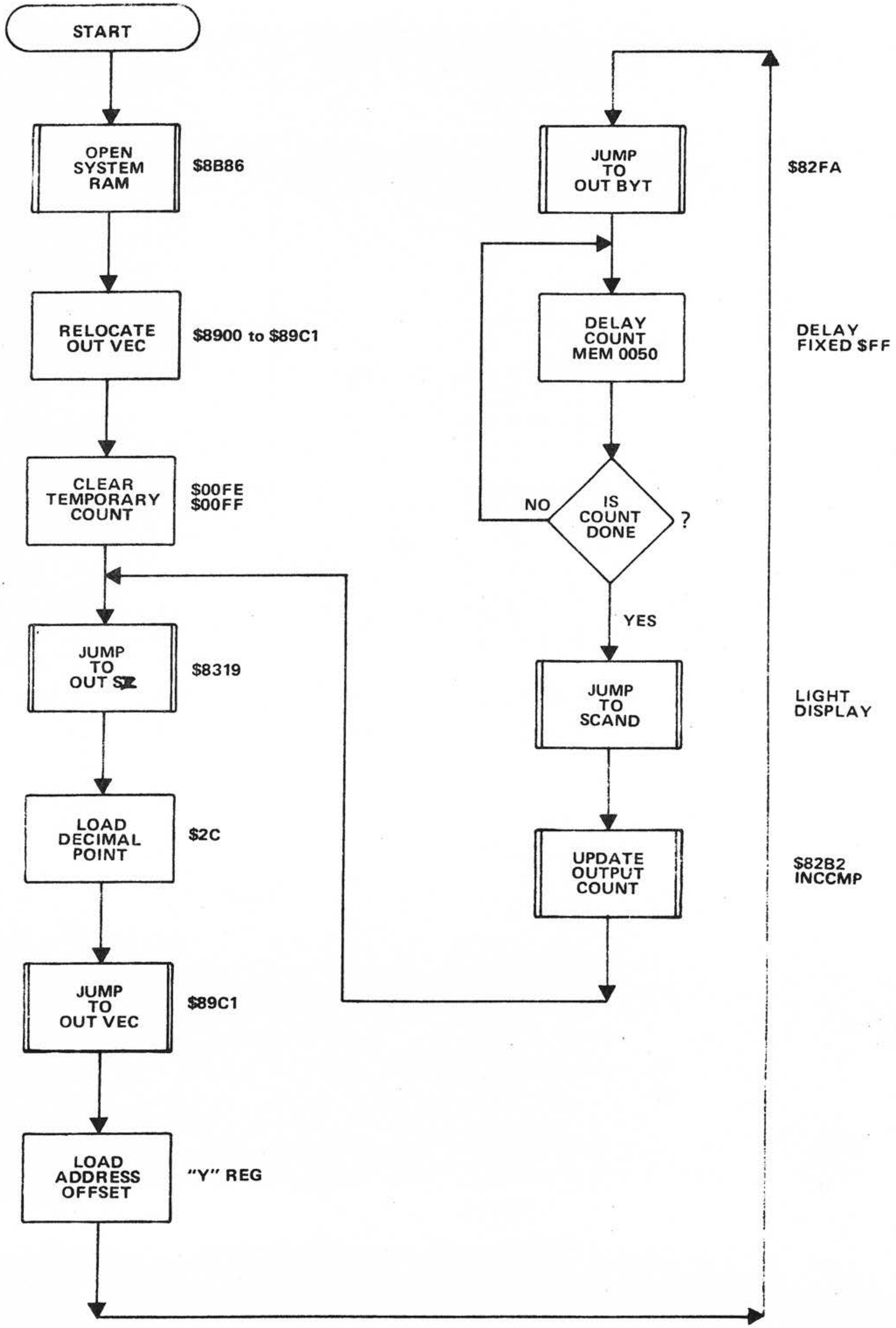
   To execute the program press (CR), (GO), **the starting address of your program,** in this case (0), (0), (0), (0), **then** (CR).

3. What Will It Do ?

   When operating properly, the program will sequentialy display memory locations and their contents starting at location 0000. It will display the information in the same format as we use to examine memory via the keyboard. It will display all locations from 0000 to $FFFF starting over again after finishing.

4. Stopping the Program:

   To stop a program and return machine control to the monitor press (RST).



EXPERIMENT #1

DISPLAY ROUTINE

```
MEMORY LOCATION             MACHINE CODE    COMMENTS

                                            ;EXPERIMENT 1
                                            ;DISPLAY ROUTINE
0000                        20 86 8B
0003                        A9 C1
0005                        8D 64 A6
0008                        A9 00
000A                        85 FE
000C                        85 FF
000E                        20 19 83
0011                        A9 2C
0013                        20 C1 89
0016                        A0 00
0018                        B1 FE
001A                        20 FA 82
001D                        C6 50
001F                        F0 06
0021                        20 06 89
0024                        18
0025                        90 F6
0027                        20 B2 82
002A                        18
002B                        90 E1
```

EXPERIMENT 2

Input and Output (I/O)

1. 6522 (VIA)

The SYM allows the user to input and output digital information from the applications connector. The SYM does the transfer through a SY6522 Versatile Interface Adapter (VIA). Within the SYM system it is referred to as device no. 1 and U25. The chip provides input or output through sixteen (16) seperate lines. These 16 lines are divided into two groups of eight (8) lines. Each group of 8 is referred to as a "port". The two (2) ports are designated "port A" and "port B". Each line, under control of the processor, can be configured as either an input or an output line, independent of all other lines.

Manipulation of an I/O port is accomplished via three (3) processor accessable registers within the 6522. These are:

(A) Data Direction Register (DDRA, DDRB)

This register specifies whether an I/O line will act as an input or output. A bit value of one (1) will cause the I/O line of corresponding bit position to act as an output line. A bit value of zero (0) will cause the I/O line of corresponding bit position to act as an input line.

There are two (2) data direction registers, one associated with each port. They are labeled DDRA and DDRB.

(B) Output Register (ORA, ORB)

When a bit of the Data Direction Register is a one (1) (indicating the line is to act as an output), the voltage on the corresponding I/O line is controlled by the corresponding bit in the output register. A one (1) in the output register would equate to a voltage of 2.4 or greater on the I/O line. This could be interpeted by TTL as a"high"or"true" condition. A zero (0) in the output register would equate to a voltage of .4 or less on the I/O line. This could be interpeted by TTL as a"low"or"false" condition.

(Note: For loading limitations see SY6522 datasheet)

(C) Input Register (IRA, IRB)

The data in this register reflects the voltage present on the port line if the line has been configured as an input line. A voltage of 2.4 or greater will be interpreted as a one (1) and a voltage of .4 or less will be interpreted as a zero (0).

(Note: The above statement regarding the operation of the input register will be true within the outlined experiments in this course, however the device can be caused to operate in exception to this. See the SY6522 data sheet for a complete discussion of operating modes.)

In the SYM the registers (of U25) described above are located at the following memory locations:

Port A		Port B	
DDRA=$A003		DDRB=$A002	
IRA =$A001	(R/W=high)	IRB =$A000	(R/W=high)
ORA =$A001	(R/W=low)	ORB =$A000	(R/W=low)

2. EIGHT BIT PARALLEL OUTPUT:

Configure all eight (8) lines of port A as output lines. This is done by writing %11111111 (% indicates the number following is represented in base 2) to the DDRA located at $A003. Use the SYM keyboard to do this.

Write various hexadecimal numbers into the ORA located at $A001.

You will observe the binary representation of these numbers on the eight (8) LEDs on the off board peripheral device. (See Fig. 1)

3. EIGHT BIT PARALLEL INPUT:

Configure port B as an input port. This is done by writing %00000000 to the DDRB located at $A002. This is done automatically by the monitor every time you press the (RST) key.

Note: NEVER configure ANY of the port B I/O lines as output lines. To do so may result in damage to the SYM. (This applies so long as the SYM is connected to the experimental device provided for use with the course.)

Examine the contents of the IRB which is located at $A000. The data in this register reflects the state of the I/O lines which are connected to the experimental device. (See Fig. 1)

4. INPUT FROM THE SYM KEYBOARD:

The SYM monitor contains a number of subroutines which it uses to control machine operation. These routines are available to the user. Most monitor subroutines make use of a small section of RAM contained within the SY6532 (U27). The machine however, protects itself from faulty user programs by "write-protecting" this block of RAM. When using a monitor subroutine which accesses this area of memory, and most do, it is necessary to disable this write-protect feature. This is most easily accomplished by calling the monitor subroutine which it itself uses to "un-write-protect" this special section of memory. This subroutine is labeled "ACCESS" and starts at memory location $8B86.

The SYM uses a monitor subroutine labeled "GETKEY" to poll the keyboard. While waiting for a key depression it scans the display, thus keeping it lit while the operator takes time to figure out which button to press next. When a key is finally depressed it will generate the ASCII code for the key, store this value in the accumulator, and return to the calling program.

Enter the following program. It uses the GETKEY subroutine to return the ASCII value of the key depressed to the accumulator. This value is then transfered to the eight (8) LEDs located on the peripheral device provided for use with the course.

5. WHAT WILL IT DO ?

When the processor begins executing the program the onboard display will remain lit. It will display the contents present just previous to pressing (CR), most likely "G 0000". The onboard display will remain unchanged until machine control is returned to the monitor program. The eight (8) LEDs on the external display will initially go dark.

When a key is pressed (except for (RST) and (DEBUG)) some of the off board LEDs will light. The binary code they present is the ASCII code for the last key depressed. Prove this to yourself by trying different keys and comparing the result output to the LEDs with the ASCII codes on the back of the SYM reference card.

KEYBOARD INPUT

```
LOC   CODE        LINE              COMMENTS

                                    ;EXPERIMENT 2
                                    ;KEYBOARD INPUT ROUTINE
0000  20 86 8B    JSR $8B86         ;UN-WRITE PROTECT SYSTEM RAM
0003  A9 FF       LDA #%11111111    ;CONFIGURE PORT A AS OUTPUT
0005  8D 03 A0    STA $A003
0008  20 AF 88    JSR $88AF         ;CALL GETKEY
000B  8D 01 A0    STA $A001         ;STORE RETURNED VALUE IN ORA
000E  4C 08 00    JMP $0008         ;LOOP BACK FOR ANOTHER KEY
```

EXPERIMENT 3

1. OUTPUT TO THE ONBOARD DISPLAY:

The SYM display is driven by the monitor. The monitor actually stores the data to be displayed in six reserved memory locations. These locations are referred to as the "display buffer". Each location within the buffer corresponds to a display digit. The programs and subroutines which appear to modify the display actually manipulate this display buffer.

One useful subroutine in the monitor that does just that is called "OUTBYT". OUTBYT takes the value stored in the accumulator, converts it to two (2) hexadecimal digits, and scrolls these digits onto the display (in reality the display buffer).

The display scrolls from right to left. Data scrolled off the display is lost.

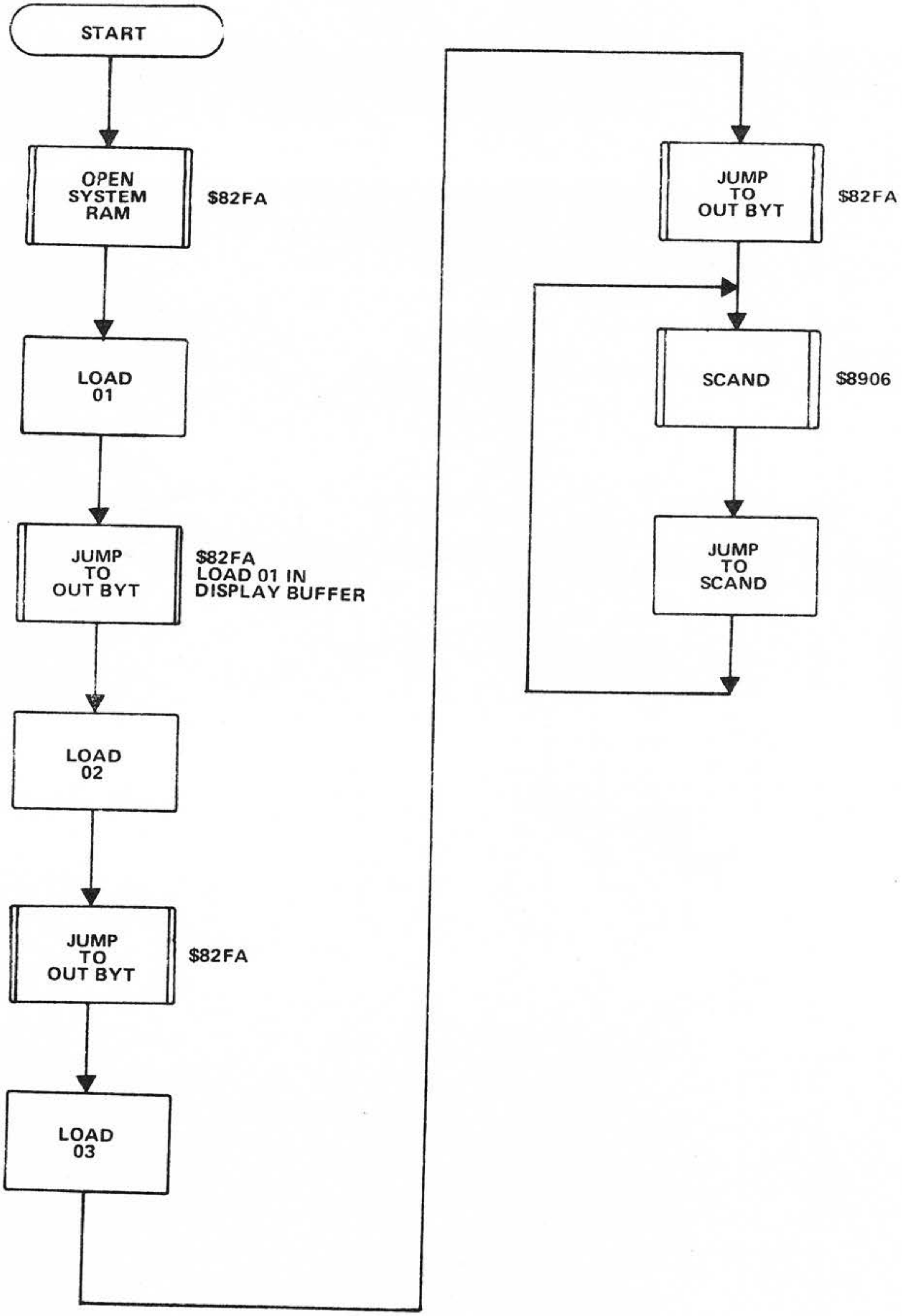
The monitor contains another subroutine which it uses to transform the data in the display buffer into signals which will light the LEDs in a readable form. This routine is labeled "SCAND" and starts at location $8906. It is of note that this routine will light each digit for only a brief instant. Therefore it must be called repeatedly to produce a readable display.

The following program will use these subroutines to force the display to read "010203".

2. WHAT WILL IT DO ?

This program will force the display to read "010203". The keyboard will no longer function (except of course for the (RST) and (DEBUG) keys) as there are no instructions in the program to interrogate it.

(Remember: the program starts at location $0011.)



EXPERIMENT #3

DISPLAY OUTPUT

```
LOC     CODE              LINE                COMMENTS

                                              ;EXPERIMENT 3
                                              ;DISPLAY OUTPUT
0011    20 86 8B          JSR $8B86           ;UN-WRITE PROTECT SYSTEM RAM
0014    A9 01             LDA #$01            ;GET CONSTANT "01"
0016    20 FA 82          JSR $82FA           ;CALL OUTBYT
0019    A9 02             LDA #$02            ;GET CONSTANT "02"
001B    20 FA 82          JSR $82FA           ;CALL OUTBYT
001E    A9 03             LDA #$03            ;GET CONSTANT "03"
0020    20 FA 82          JSR $82FA           ;CALL OUTBYT
0023    20 06 89          JSR $8906           ;CALL SCAND
0026    4C 23 00          JMP $0023           ;JUMP BACK TO CALL SCAND
```

3. **OPTIONAL EXPERIMENT:**

Try to link this program and the previous experiment so that the ASCII value of the last key depressed will appear not only on the off board LEDs, but also on the onboard display.
(Hint: Use the "GETKEY" subroutine to light the display rather than the "SCAND" subroutine.)

EXPERIMENT 4

Controlling External Devices

1. **DEBUG:**

It is possible to execute a program one instruction at a time on the SYM using a feature called "debug". To use the feature enter a program as before. When you have finished press debug (ON), (GO), the starting address, and (CR). This will cause the SYM to execute the first instruction and stop. To execute the next instruction press (GO), then (CR). You may continue in this manner through the entire program.

After the completion of an instruction the display will read "XXXX.2 .", XXXX will be the current value of the program counter.

2. EXAMINING MACHINE REGISTERS:

Often when stepping through a program as described above one desires to examine the registers within the machine to determine whether the intended machine action has occured.

To examine the machines registers press (REG), (CR). The display will read "P XXXX", XXXX will be the current value of the program counter. By pressing (→) you may examine the remaining registers.

P	=Program Counter
r1	=Stack Pointer
r2	=Processor Status Register
r3	=Accumulator
r4	=Index Register X
r5	=Index Register Y

The value of the registers may be changed by examining the desired register and entering the new value.

3. SIMULATING A SIMPLE CONTROL SYSTEM:

This experiment will use the SYM and the external peripheral device to simulate the use of a microcomputer in a process control application.

The experiment uses three (3) of the switches on the outboard device as inputs. These are:

Switch 2 - Goes on when the solution in the vat passes a "pass - fail" test for clarity.

Switch 1 - Goes on when the vat runs empty.

Switch 0 - Goes on when the vat becomes full.

We will use the LEDs on the outboard device as outputs. They Indicate:

LED 0 - will go on to indicate that the pump is on.

LED 4 - will go on to indicate that the filter is on.

LED 7 - will go on to indicate that the drain valve is closed.

4. WHAT WILL IT DO ?

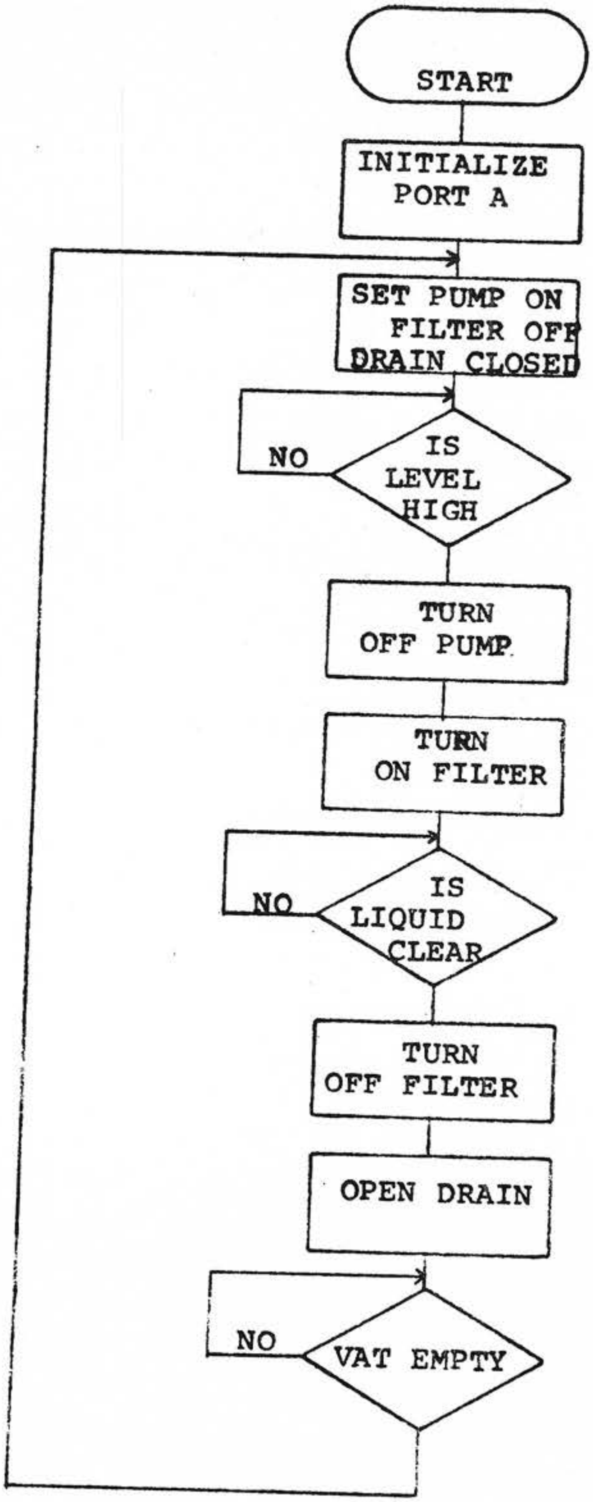
First run the program using the "debug" mode. Note particularly how the ROR, AND, and ORA instructions are used to force decisions on the basis of a single bit. Follow the program comments and flowchart as you step through the program.

Initially set the inputs to the following states:

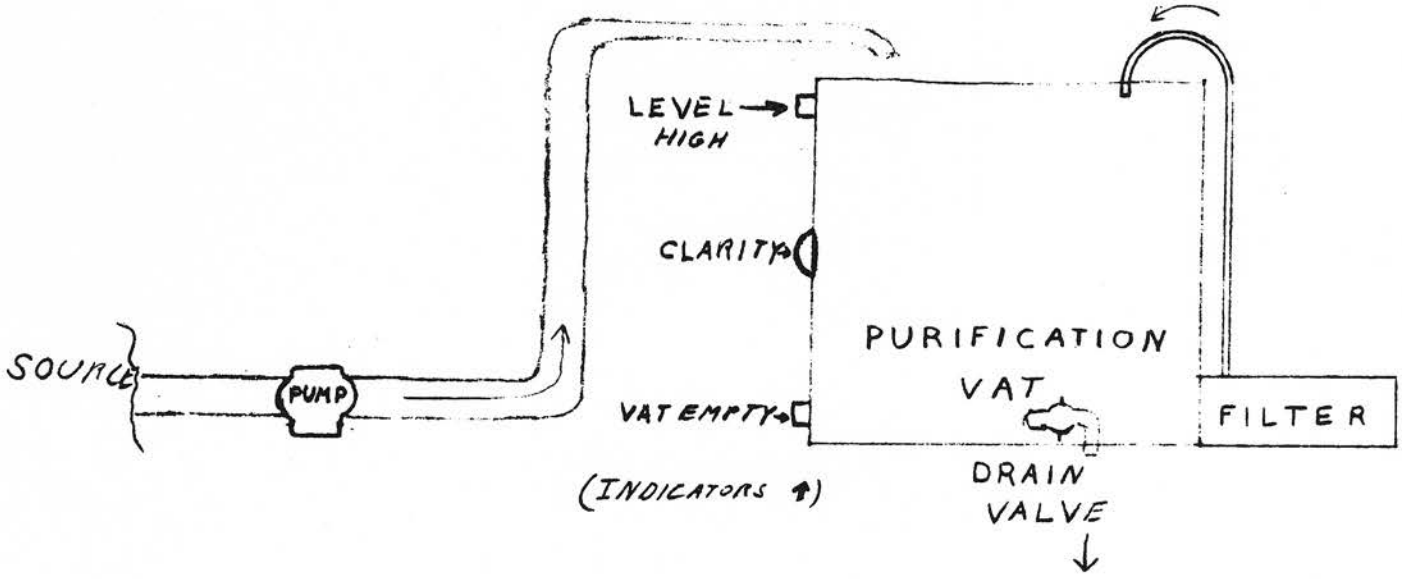
SW2 - off (not clear)
SW1 - off (not empty)
SW0 - off (not full)

When running the program at normal speed you will first observe the drain valve close and the pump go on. When the vat fills up (SW0=on) the pump will go off. In addition, the filter will go on. When the fluid becomes clear enough (SW2=on) due to the action of the filter, the filter will be turned off. The purified fluid will then be drained from the vat by opening the drain valve. As the level drops the level high indicator will go off (SW0=off) and finally the vat empty indicator will go on (SW1=on). When this occurs a new cycle is started.

```
LOC                                  COMMENTS
                                     ;EXPERIMENT 4
                                     ;PROCESS CONTROL
                                     ;PA0=PUMP
                                     ;PA4=FILTER
                                     ;PA7=DRAIN VALVE
                                     ;PB0=LEVEL HIGH
                                     ;PB1=VAT EMPTY
                                     ;PB2=CLARITY
                                     ;DDRB=$A002
                                     ;DDRA=$A003
                                     ;IRB=$A000
                                     ;IRA=$A001
                                     ;ORA=$A001
0000                   LDA #$FF      ;CONFIGURE PORT A AS OUTPUT
0002                   STA $A003     ;DDRA
0005                   LDA #$81      ;FORCE PUMP ON, DRAIN
                                     ;CLOSED, ALL OTHERS OFF
0007                   STA $A001     ;ORA
000A                   LDA $A000     ;CHECK FOR LEVEL HIGH
000D                   ROR           ;(ACCUMULATOR ADDRESS MODE)
000E                   BCC $FA       ;IF NOT CHECK AGAIN (WAIT)
0010                   LDA $A001     ;TURN OFF PUMP
0013                   AND #$FE
0015                   STA $A001
0018                   LDA $A001     ;TURN ON FILTER
001B                   ORA #$10
001D                   STA $A001
0020                   LDA $A000     ;CHECK CLARITY
0023                   AND #$04
0025                   BEQ $F9       ;NOT CLEAR? CHECK AGAIN
0027                   LDA $A001     ;IS CLEAR,TURN OFF FILTER
002A                   AND #$EF
002C                   STA $A001
002F                   LDA $A001     ;OPEN DRAIN
0032                   AND #$7F
0034                   STA $A001
0037                   LDA $A000     ;CHECK FOR VAT EMPTY
003A                   ROR           ;(ACCUM. ADDRESS MODE)
003B                   ROR           ;(ACCUM. ADDRESS MODE)
003C                   BCC $F9       ;NOT EMPTY? CHECK AGAIN
003E                   BCS $C5       ;EMPTY, RESTART CYCLE
```



EXPERIMENT 4
FLOWCHART



EXPERIMENT 4 PICTORIAL

EXPERIMENT 5

Counting Loops

1. **TRACE:**

   The SYM offers to the user a modified form of the debug mode called "trace".
   To use this feature first store some non-zero value at address $A656 ($09 is a good value). Then execute the program just as you would using the "debug" mode.
   The monitor will then display the program counter and accumulator, pause, execute the next instruction, display PC and A, pause, etc. .

2. **A COUNTING LOOP:**

   The following example shows how to set up a counter (here the X index register) to allow execution of a program segment some preselected number of times. We could have just as easily used the Y register, the accumulator, or any R/W memory location.
   Run this program using the "trace" mode and at full speed.

3. **WHAT WILL IT DO ?**

   At full speed the program will execute so quickly that the only indication that the program ran will be the outboard LEDs displaying the value 10. In trace mode the outboard LEDs will slowly count from 0 to 10.

COUNTING LOOP PROGRAM

```
                                   ;COUNTING LOOP PROGRAM
                                   ;EXPERIMENT 5
               LDA #$FF            ;CONFIGURE PORT A AS OUTPUT
               STA DDRA
               LDA #0              ;TURN OFF LEDS
               STA ORA
COUNT          LDX #10             ;LOAD COUNTER WITH 10
LOOP           INC ORA             ;INCREMENT OUTPUT PORT
               DEX                 ;DECREMENT COUNT
               BNE LOOP            ;IF COUNT NOT ZERO GO
                                   ;TO LOOP
DONE           JMP $8000           ;JUMP TO MONITOR
```

4. TIMING LOOPS:

All machine operations on the SYM are controlled with the crystal controlled clock oscillator. This oscillator operates at a nominal one megahertz. The frequency is very stable but may not be exactly the specified value. Each machine instruction requires a specific number of clock cycles. Thus program segments and loops can be used to produce very precise time delays which are as accurate as the clock. The number of cycles for each instruction can be found in the SYM reference card and the SY6500/MCS6500 Programing Manual.

The following program yields a delay of 501 clock cycles or if the clock really is a one megahertz clock, 501 microseconds.

TIME DELAY PROGRAM

```
              LDX #100     ;2 CYCLES
        LOOP  DEX          ;2 CYCLES
              BNE LOOP     ;3 CYCLES EXCEPT WHEN
                           ;BRANCH IS NOT TAKEN
                           ;THEN JUST 2
```

The loop is five (5) cycles long and is executed one hundred (100) times. Except on the last pass through the loop when the branch is not taken. In the 6500 series of processrs this shortens the execution time by one cycle. The calculation used in finding the cycle time of the loop is illustrated below.

```
   INITIAL COUNT VALUE       CYCLE TIMES OF INSTR. IN LOOP
((         X            *   ( 2          +        3          )-1)+2=
```

((X*(2+3)-1)+2=501

The value "1" was subtracted to account for the shortened branch instruction and the final value "2" was added to account for the initial "LDX" instruction which is not within the loop.

5. DELAY SUBROUTINE:

Now that you have the basic idea here is a more complicated program. We put the time delay in a subroutine so that it may be used more readily.

6. WHAT WILL IT DO ?

The main program continually increments the value present at the output port. However between each increment it calls the delay subroutine.
Start the main program at location $0000 and the subroutine at $0200.

EXPERIMENT5

```
                                              ;EXPERIMENT 5
                                              ;MAIN PROGRAM
                                              ;START AT $0000
            START   LDA #$FF                  ;CONFIGURE PORT A AS OUTPUT
                    STA DDRA
                    LDA #$00                  ;INIT A TO ZERO
            SHOW    STA ORA                   ;MOVE A TO OUTPUT REG
                    CLC                       ;CLEAR CARRY BEFORE ADD
                    ADC #$01                  ;ADD 1 TO A
                    JSR DELAY                 ;WAIT
                    JMP SHOW                  ;LOOP BACK TO SHOW
```

```
                                              ;EXPERIMENT 5
                                              ;DELAY SUBROUTINE
                                              ;START AT $0200
            DELAY   LDY #200                  ;LOAD Y WITH 200=Ty
            LOOPY   LDX #98                   ;LOAD X WITH 98=Tx
                    JMP LOOPX                 ;WASTE 3 CYCLES
            LOOPX   DEX                       ;DEC X COUNT
                    BNE LOOPX                 ;IF X NOT 0 DEC AGAIN
                    DEY                       ;DEC Y COUNT
                    BNE LOOPY                 ;IF Y NOT 0 CNTDWN 98 X AGAI
                    RTS
```

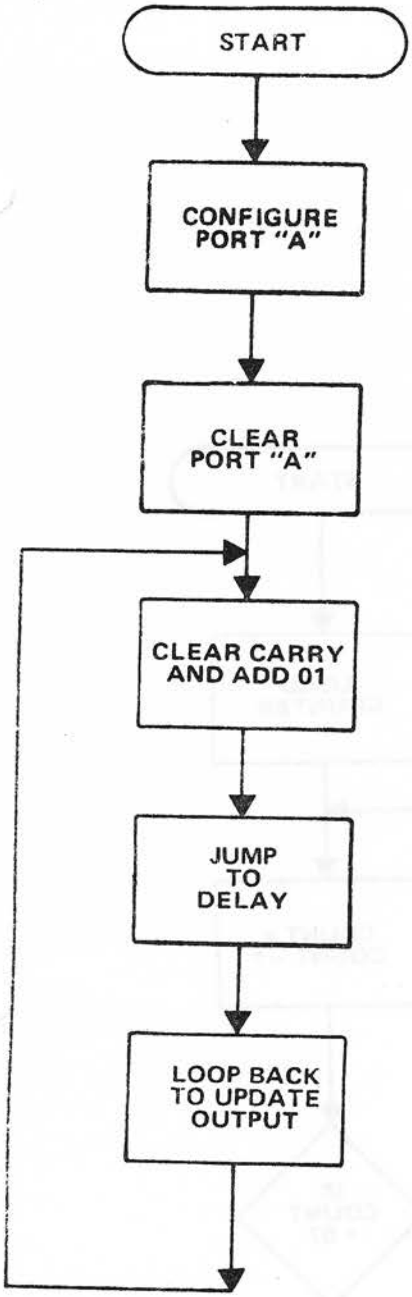
7. **OPTIONAL:**

The total time delay here is :

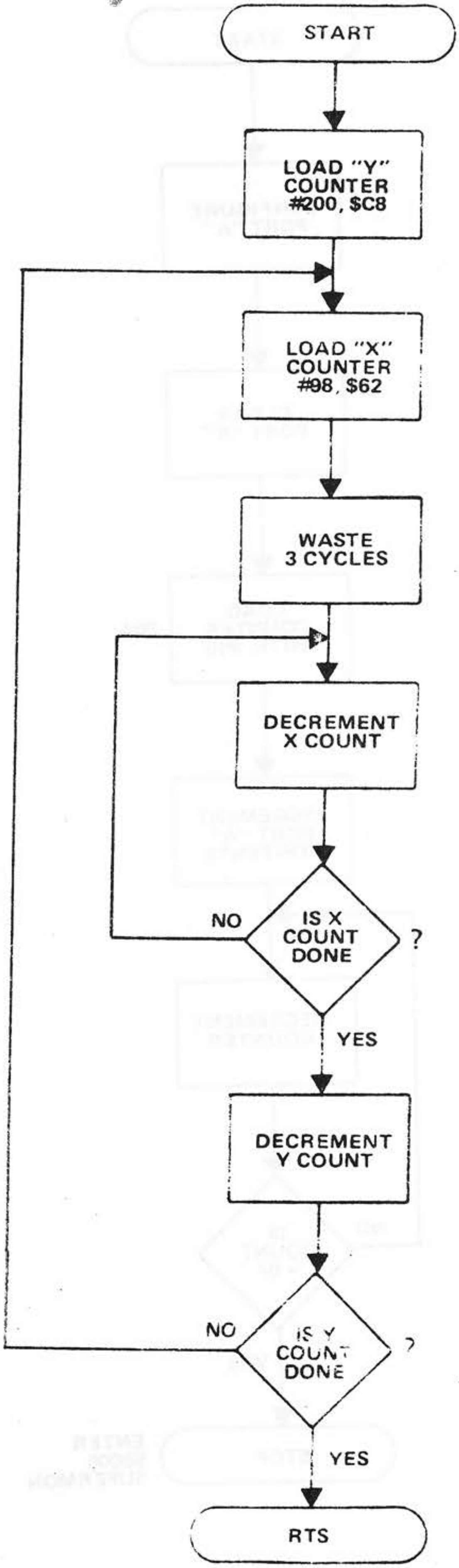
Ty*(5Tx+9)+13= Total Time Delay

Run this program with different values for Ty and Tx. You might try to write a program that would allow you to enter time constants from the keyboard in real time.

This is a good program to use to see the effects of some of the other accumulator insructions. Replace the CLC, ADC # sequence with SEC, SBC # or ROR or ROL or ASL or LSR. If you replace a two byte instruction with a one byte instruction be sure to add a NOP to fill the gap left in the program.

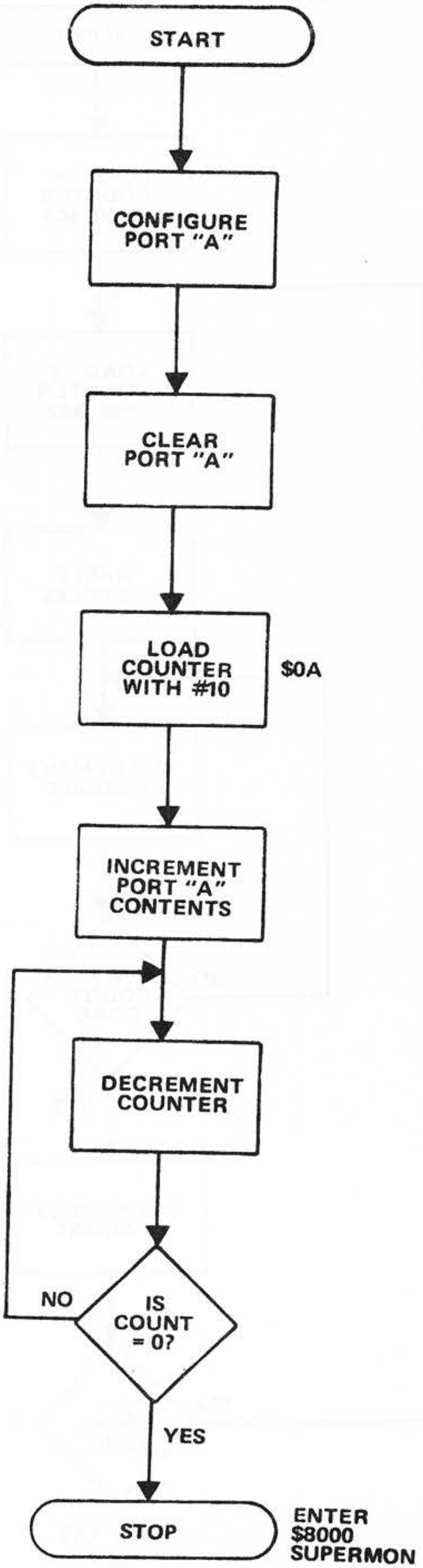


MAIN PROGRAM

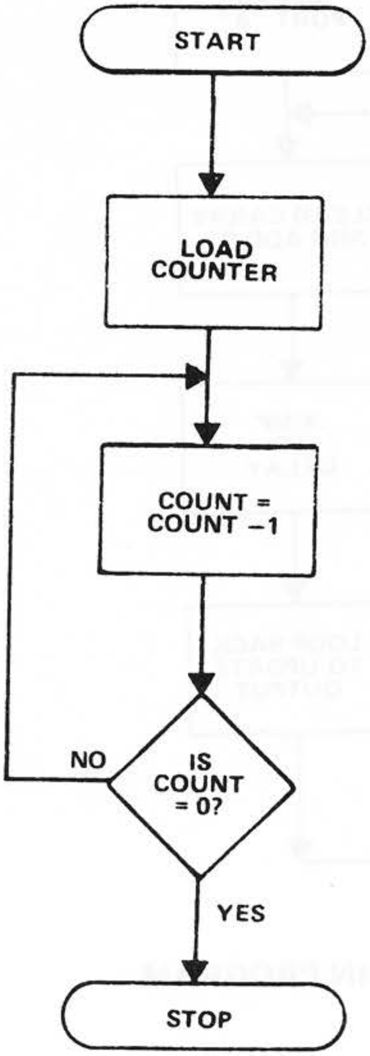


DELAY SUBROUTINE

EXPERIMENT #5



EXPERIMENT #5



EXERCISES

EXPERIMENT 6

INTERRUPTS & INTERVAL TIMERS

1. T1 & T2

The SY6522 versatile interface adapter contains two very useful interval timers.

The timers structure is that of one sixteen (16) bit counter divided into two eight (8) bit sections and two (2) eight (8) bit latches which are used to load the counter.

The processor may load the latches at anytime without affecting the countdown in progress. At the appropriate time the values stored in the latches are loaded into the counter simultaneously. The counter then begins to decrement at the system clock rate. When the count reaches zero (0) the chip may deliver an interrupt request signal (IRQ) to the processor to indicate the end of the time interval.

Timer 1 will operate in two modes:

(A)

ONE SHOT MODE:

In this mode the high order latch is not used. The low order latch may be loaded before the actual countdown begins. The countdown is initiated by a "write T1 (timer 1) counter high". This triggers the transfer of the value in the low order latch into the low order section of the counter, writes the value specified into the high order section of the counter, and starts the counter decrementing. When the counter reaches zero (0) the IRQ output on the SY6522 will go low (active) (see discussion below on control registers) and the T1 interrupt flag in the Interrupt Flag Register (IFR) will be set.

(B)

FREE RUNNING MODE:

This mode is available for T1 only. It operates in essentially the same manner as the one shot mode, except when the count reaches zero (0) the value of the high and low order latches will AUTOMATICALLY be transferred to the counter. This allows evenly spaced interrupts which have no dependence on service routine response time.

In either mode, after the timer times out it sets an interrupt flag in the IFR (a register within the 6522). Before this flag can be set again (to trigger another interrupt) it must first be cleared. One method of doing this is to write a one (1) into the bit position you wish cleared (that is into the IFR).

2. CONTROL REGISTERS:

The SY6522 provides several control registers which will need to be manipulated to control the operation of the chip. Here we will only cover three which are essential to a basic understanding of the operation of the chip. The others which are not discussed are explained at length in the SY6522 datasheet in your SYM Reference Manual.

ACR

Auxillary Control Register:

This register controls the timers, the shift register, and input latching. A 0 in bit position 6 enables the free running mode of timer 1. A 1 in bit position 6 forces timer 1 to operate in the one shot mode.

IFR

Interrupt Flag Register:

Each possible source of an interrupt from within the chip corresponds to a bit within this 8 bit register. There are seven possible sources within the chip. These are:

Bit 0 - (CA2) Port A Control Line 2
Bit 1 - (CA1) Port A Control Line 1
Bit 2 - (SR) Shift Register
Bit 3 - (CB2) Port B Control Line 2
Bit 4 - (CB1) Port B Control Line 1
Bit 5 - (T2) Timer 2
Bit 6 - (T1) Timer 1

The remaining bit indicates whether an interrupt request signal is being output from the chip.

IER

Interrupt Enable Register:

Each possible interrupting source described above has a corresponding bit within the IER. If the bit in the enable register is a 1,the source is allowed to cause an interrupt request signal to be generated. If however, the enable register bit asociated with the source is a 0 then no matter what the state of the Interrupt Flag Register bit, the source will not be allowed to cause an interrupt request signal to be issued from the chip. For example if T1 were to time out (and it was the only source within the chip to request an interrupt) bit 6 of the IFR would be a 1 , if in addition bit 6 of the IER were a 0 no IRQ signal would be generated by the chip. Bit 7 of the IFR would remain a 0 in this case also.

The bit assignments within the IER are the same as in the IFR.

The IER has an unusual procedure for letting the processor alter its contents.

If you wish to set a bit or bits within it write a one (1) to the positions you wish set and a one (1) to bit seven (7). The positions you write a zero (0) to will remain unchanged.

If you wish to clear a bit or bits within it write a one (1) to the positions you wish cleared and write a zero (0) to bit seven (7). The positions you write a zero (0) to will remain unchanged.

REGISTER	BIT							
	7	6	5	4	3	2	1	0
IFR	IRQ	T1	T2	CB1	CB2	SR	CA1	CA2
IER	SET/ CLEAR CONTROL	T1	T2	CB1	CB2	SR	CA1	CA2

3. EXPERIMENT 6

Note that the main program is the same program we used for experiment 1 with additional code preceding it.

These extra steps are used only once to initialize the various control registers within the SY6522, insert the proper interrupt vectors into system RAM, and initialize port A as an output port.

Once these functions have been performed Experiment 1 will operate just as before.

4. WHAT WILL IT DO ?

The LEDs on the outboard device will increment at a rate specified by the value loaded into the interval timer (T1) by the initialization section of the program.

Experiment 1 will operate as before with no observable performance degradation.

EXPERIMENT 6

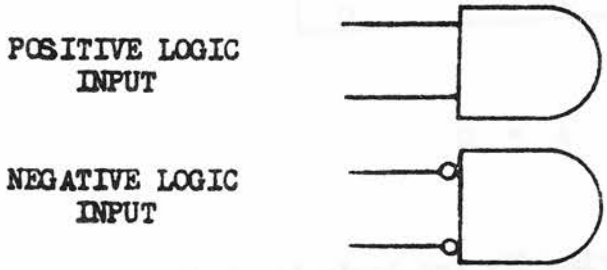
```
LOC     LINE               COMMENTS
                           ;EXPERIMENT 6
                           ;INITIALIZATION & MAIN (EXP. 1)
                           UIRQVCL=$A678
                           UIRQVCH=$A679
                           DDRA=$A003
                           ACR=$A00B
                           IER=$A00E
                           T1L-H=$A007
                           T1C-H=$A005
0000    JSR $8B86          ;UN-WRITE PROTECT SYSTEM RAM
0003    LDA #$00           ;STORE NEW INTERRUPT VECTOR
0005    STA $A678          ;         USER IRQ VECTOR (LOW)
0008    LDA #$02
000A    STA $A679          ;         USER IRQ VECTOR (HIGH)
000D    LDA #$FF           ;CONFIGURE PORT A AS OUTPUT
000F    STA $A003          ;         DDRA
0012    LDA #$40           ;SET T1 TO FREE RUNNING MODE
0014    STA $A00B          ;         ACR
0017    LDA #$C0           ;ENABLE INTERRUPTS FROM 6522
0019    STA $A00E          ;         IER
001C    STA $A007          ;SET INTERVAL TIMER
001F    STA $A005
0022    20 86 8B           ;EXPERIMENT 1
0025    A9 C1
0027    8D 64 A6
002A    A9 00
002C    85 FE
002E    85 FF
0030    20 19 83
0033    A9 2C
0035    20 C1 89
0038    A0 00
003A    B1 FE
003C    20 FA 82
003F    C6 50
0041    F0 06
0043    20 06 89
0046    18
0047    90 F6
0049    20 B2 82
004C    18
004D    90 E1


                        INTERRUPT SERVICE ROUTINE
                           ;START AT $0200
                           IFR=$A00D
                           ORA=$A001

0200    PHA
0201    LDA $A00D          ;CLEAR INTERRUPT FLAG IN IFR
0204    STA $A00D
0207    INC $A001          ;INCREMENT LEDS ON OUTBOARD
020A    PLA
020B    RTI
```

LOGIC AND INTERFACE DEVICES

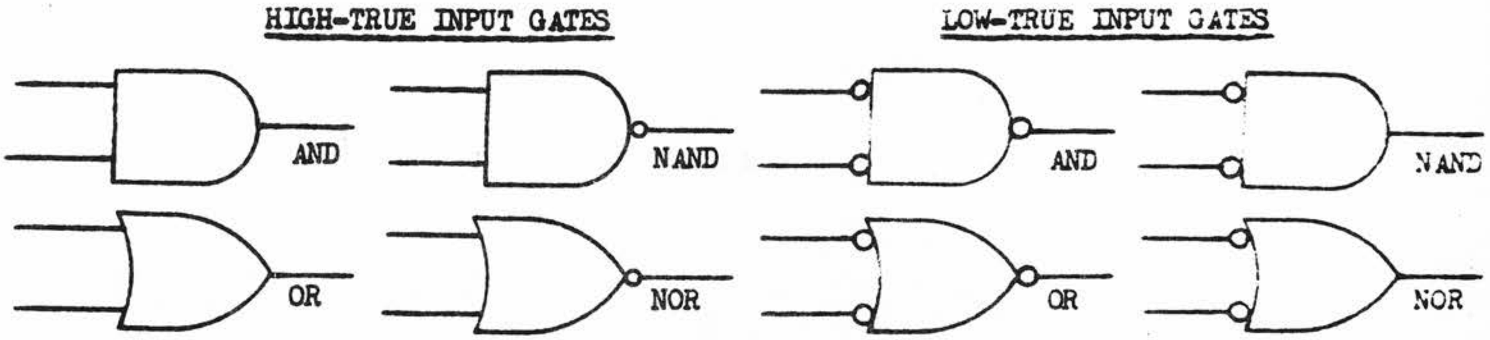
BASIC LOGIC DEVICES

Although microprocessors are called (and often are used as) logic replacements, basic logic gates are still needed in most microcomputer systems. They are used for Buffers, Latches, Address Decoders, and Signal Conditioners. Therefore, it is important to have a good understanding and working knowledge of basic logic gates.

Digital logic operates in the binary number system. Therefore, any one input or output can only be in one of two distinct states, either a "1" or a "Ø". Normally, references made in regard to a digital signal, a logical "1" is greater than 2.0 volts and a logical "Ø" is less than 0.8 volts; this is called HIGH-TRUE or POSITIVE-TRUE LOGIC. LOW-TRUE or NEGATIVE-TRUE LOGIC is the opposite, a logical "1" is less than 0.8 volts and logical "Ø" is greater than 2.0 volts. On logical diagrams, the type of logic (Positive or Negative) is shown by the use of a circle in the input/output lead touching the logic symbol for the gate to indicate a LOW-TRUE input/output. The absence of this circle indicates a HIGH-TRUE input/output.



When the circle is used in an output lead of a POSITIVE-TRUE input gate or the absence of it in a LOW-TRUE input gate, it changes the name of the gate by adding the letter "N" in front of the gate's name, such as:

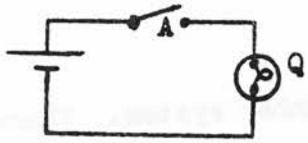


An explanation of the basic logic gates follows:

I. NON-INVERTING BUFFER

This device is used primarily to increase the load handling capabilities of another device. The output of this device will always be the same logic level as its input.

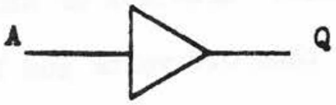
ANALOGY:



The switch closed represents a high input
The switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

Closing the switch turns the lamp ON. Opening the switch turns the lamp OFF.

LOGIC SYMBOL:



A - Input
Q - Output

TRUTH TABLE:

A	Q
1	1
0	0

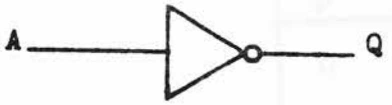
BOOLEAN EQUATION:

A = Q

II. INVERTING BUFFER

This device is used primarily for logic level inversion. The

LOGIC SYMBOL:



A - Input
Q - Output

The small circle at the end of the gate indicates output inversion.

TRUTH TABLE:

A	Q
1	0
0	1

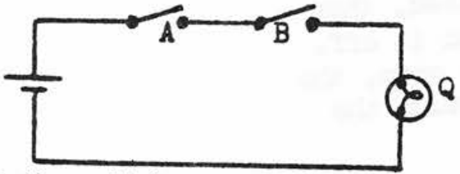
BOOLEAN EQUATION:

$$A = \overline{Q}$$

III. AND

This device used primarily to indicate whether or not <u>all</u> of its inputs are high at the same time. The output is HIGH-TRUE.

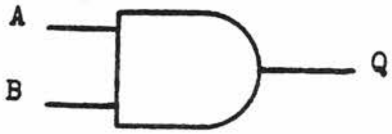
ANALOGY:



A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

Both switches must be closed to turn the lamp on. If either or both switches are open, the lamp will be off.

LOGIC SYMBOL:



A & B - INPUTS

Q - OUTPUT

TRUTH TABLE:

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

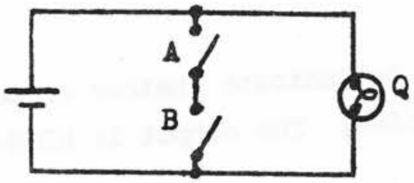
BOOLEAN EQUATION:

$$A \cdot B = Q$$

IV. NAND

This device is used the same as the AND, except the output is LOW-TRUE.

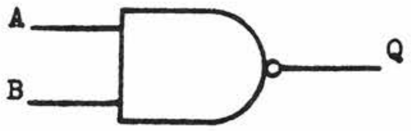
ANALOGY:



A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

When both switches are closed, they short out the lamp and turn it off. If either or both switches open, the short will be removed and turn the light on.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

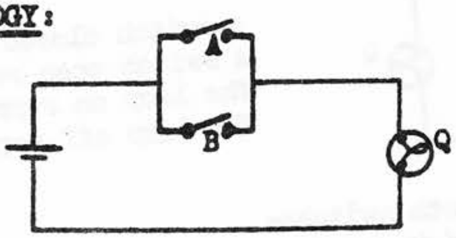
BOOLEAN EQUATION:

$$A \bullet B = \overline{Q}$$

V. OR

This device is used to indicate when at least one of its inputs is high. The output is HIGH-TRUE.

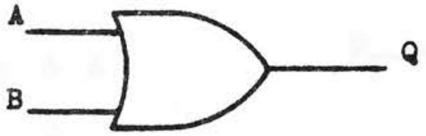
ANALOGY:



A switch closed represents a high in
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

Closing of either or both switches turns the light on. All the switches must be open to turn the lamp off.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

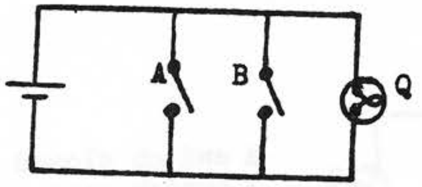
BOOLEAN EQUATION:

$$A + B = Q$$

VI. NOR

This device is used for the same purpose as the OR gate, except the output is LOW-TRUE.

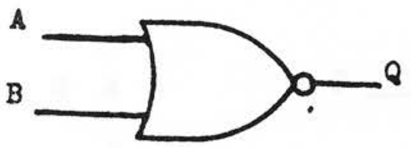
ANALOGY:



A switch closed represents a high input
A switch open represents a low input
The lamp on represents a high output
The lamp off represents a low output

Closing of either or both switches shorts out the lamp and turns it off. Both switches must be open to turn on the lamp.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

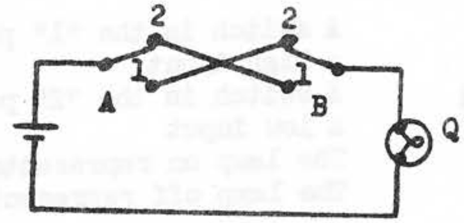
BOOLEAN EQUATION:

$$A + B = \overline{Q}$$

VII. EXCLUSIVE - OR

This device is used to indicate when one, and only one, input is high. The output is HIGH-TRUE.

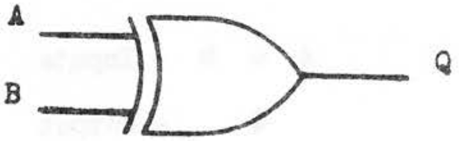
ANALOGY:



A switch in the "1" position represents a high input
A switch in the "2" position represents a low input
The lamp on represents a high output
The lamp off represents a low output

For the lamp to be on, one switch must be in the "1" position and one must be in the "2" position. Otherwise, the lamp will be off.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

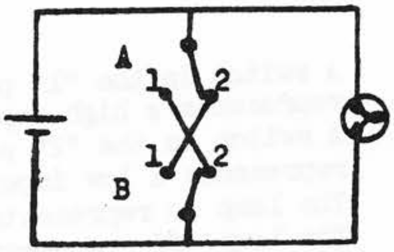
BOOLEAN EQUATION:

$$A \oplus B = A\bar{B} + \bar{A}B = Q$$

VIII. EXCLUSIVE-NOR

This device is the same as the EXCLUSIVE-OR gate, except the output is LOW-TRUE.

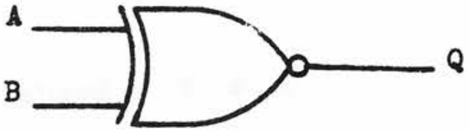
ANALOGY:



A switch in the "1" position represents a high input
A switch in the "2" position represents a low input
The lamp on represents a high output
The lamp off represents a low output

The only way to short-out the lamp and turn it off, is to have one switch in the "1" position and one switch in the "2" position. Otherwise, the lamp will be on.

LOGIC SYMBOL:



A & B - Inputs

Q - Output

TRUTH TABLE:

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

BOOLEAN EQUATION:

$$A \oplus B = A\overline{B} + \overline{A}B = \overline{Q}$$

IX. DISSCUSSION OF LOW-TRUE LOGIC

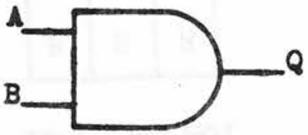
The preceeding discussion on the basic logic gates has not discussed gates with LOW or NEGATIVE-TRUE inputs. This is because there are no I.C.'s specifically designated for LOW-TRUE inputs, but a close examination

of the truth tables shows the following relationships:

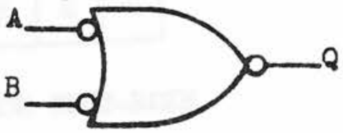
NOTE: In the following truth tables, "L" & "H" are used instead of "1" & "0" to reduce the confusion of what is a LOGICAL "1" or "0" between HIGH-TRUE and LOW-TRUE input LOGIC. An L ≤ 0.8 volts and an H ≥ 2.0 volts.

HIGH-TRUE INPUT = **LOW-TRUE INPUT**

HIGH-TRUE AND = LOW-TRUE OR

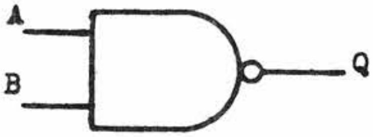


A	B	Q
L	L	L
L	H	L
H	L	L
H	H	H

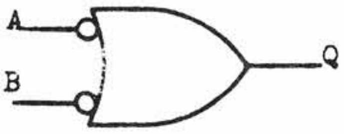


A	B	Q
L	L	L
L	H	L
H	L	L
H	H	H

HIGH-TRUE NAND = LOW-TRUE NOR

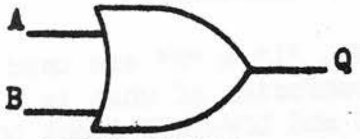


A	B	Q
L	L	H
L	H	H
H	L	H
H	H	L

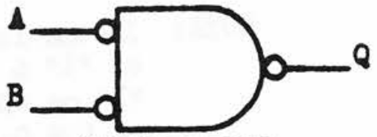


A	B	Q
L	L	H
L	H	H
H	L	H
H	H	L

HIGH-TRUE OR = LOW-TRUE AND

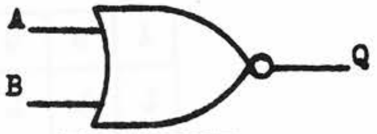


A	B	Q
L	L	L
L	H	H
H	L	H
H	H	H

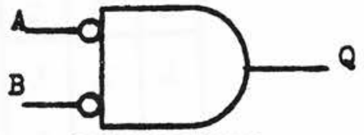


A	B	Q
L	L	L
L	H	H
H	L	H
H	H	H

HIGH-TRUE NOR = LOW-TRUE NAND



A	B	Q
L	L	H
L	H	L
H	L	L
H	H	L

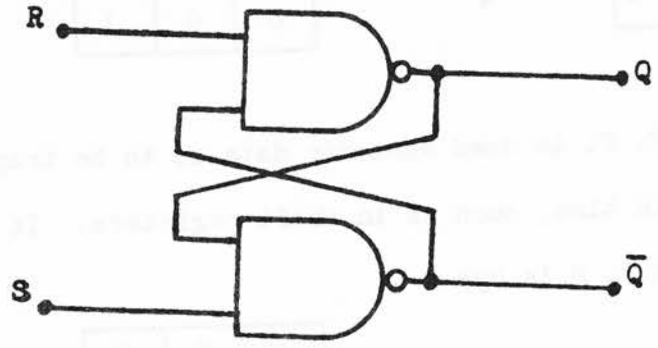


A	B	Q
L	L	H
L	H	L
H	L	L
H	H	L

FLIP-FLOPS

I. R-S LATCH:

The R-S latch was probably the first type of flip-flop ever built, (R = Reset & S = Set).



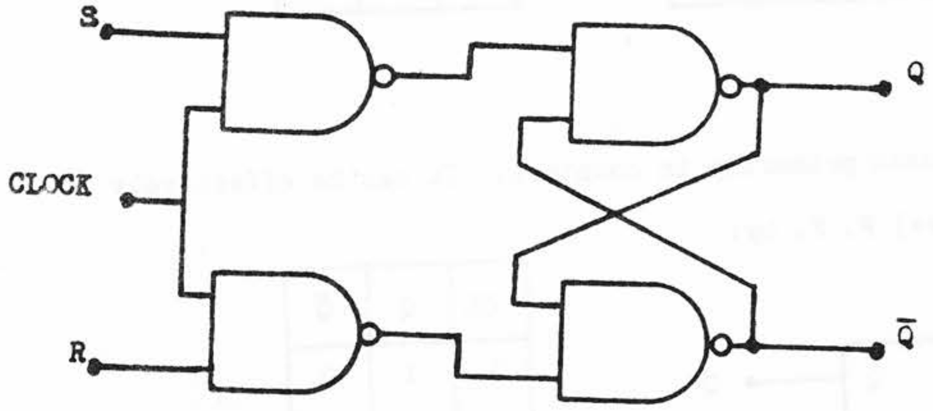
R	S	Q	$\overline{Q}$
0	0	1*	1*
0	1	1	0
1	0	0	1
1	1	NO CHANGE	

*Not allowed

To make the R-S latch into a clocked flip-flop, a clock input must be added.

II. R-S FLIP FLOP:

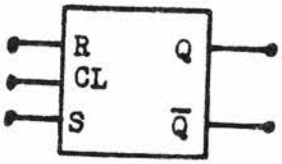
The R-S flip-flop is the simplest of the flip-flops.



R	S	Q	$\overline{Q}$
0	0	NO CHANGE	
0	1	1	0
1	0	0	1
1	1	1*	1*

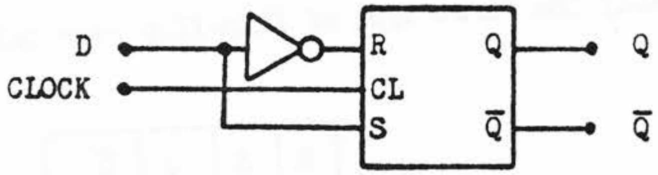
*Not allowed

The addition of the two NAND gates with the clock input changes it into a clocked R-S flip-flop. The inputs (R & S) can only change the outputs (Q & $\overline{Q}$) during a high input clock pulse. The R-S flip-flop is usually drawn in this manner:



III. DATA OR D-TYPE FLIP-FLOP:

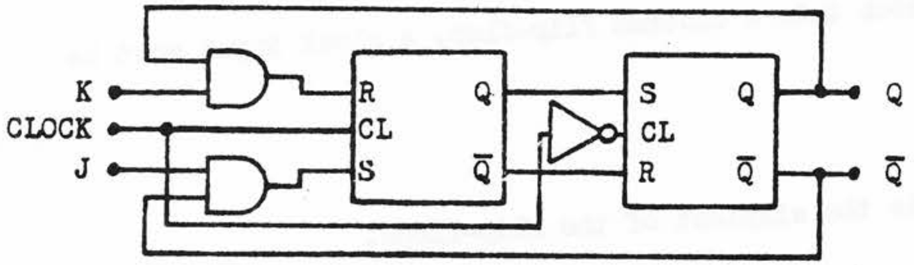
The D-Type F. F. is used primarily for a data latch. It can be made effectively from an R-S F. F. by:



D	Q	$\overline{Q}$
1	1	0
0	0	1

IV. J-K TYPE FLIP-FLOP:

The J-K or Master-Slave F. F. is used whenever data is to be trapped and latched at a given instant in time, such as in shift registers. It can be effectively made from two R-S F. F.'s by:

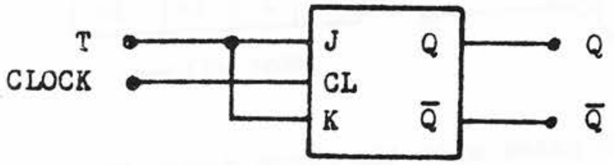


J	K	Q
0	0	Qn
0	1	0
1	0	1
1	1	$\overline{Q}$n

WHERE:
Qn = value of Q during previous clock cycle.

V. TOGGLE OR T-TYPE FLIP-FLOP:

The T-Type F. F. is used primarily in counters. It can be effectively made from a J-K (Master-Slave) F. F. by:



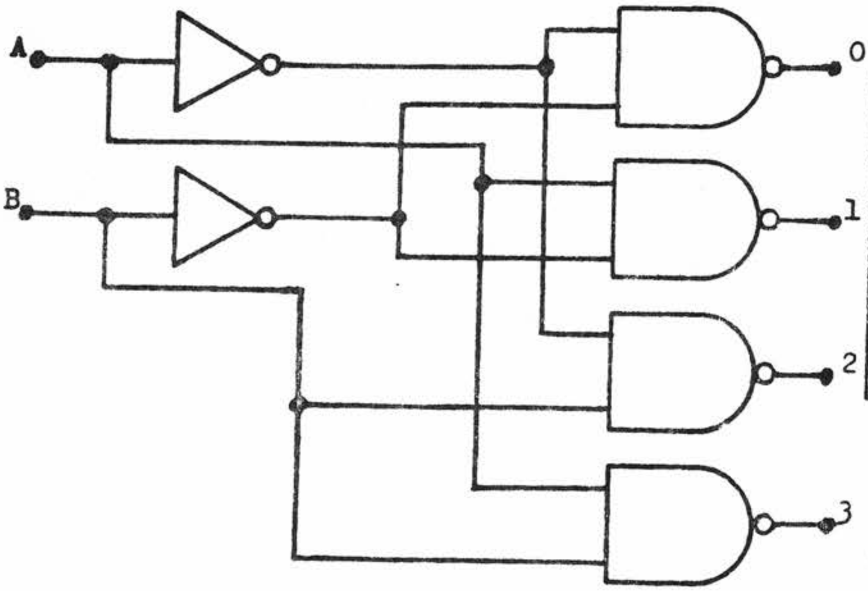
CL	Q	$\overline{Q}$
1	1	0
0	1	0
1	0	1
0	0	1

For every complete clock cycle (‾|_), Q and $\overline{Q}$ go through ½ of their cycle. Therefore, the T-Type F. F. divides the clock frequency by 2 as long as the "T" input is held high.

DECODERS/DEMULTIPLEXERS

DECODERS:

These devices are most commonly used for address decoding. They are available in 2-line to 4-line, 3-line to 8-line, 4-line to 10-line, 4-line to 16-line configuration. For simplicity, a 2-line to 4-line decoder is shown below:



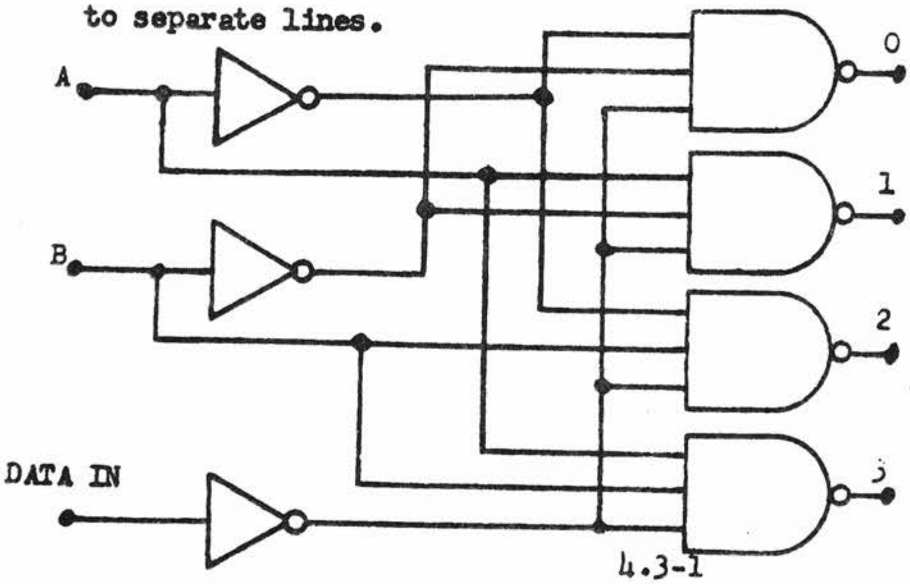
B	A	0	1	2	3
0	0	L	H	H	H
0	1	H	L	H	H
1	0	H	H	L	H
1	1	H	H	H	L

Where "A" is the least-significant bit and "B" is the most-significant bit.

With this device, it only takes 2 lines to specify or enable 4 different devices. The output is low-true.

DEMULTIPLEXERS:

These gates are the same as a decoder, except the NAND gates have an additional input for data. This device separates serial data on one line to separate lines.



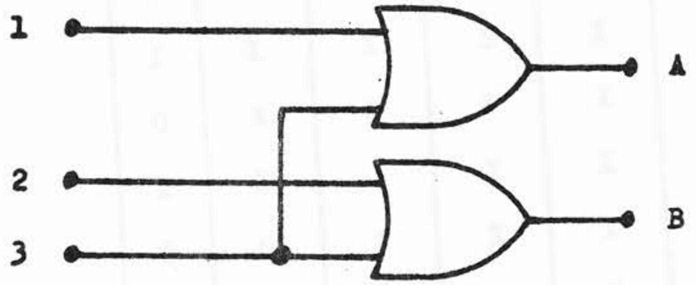
B	A	0	1	2	3
0	0	D	H	H	H
0	1	H	D	H	H
1	0	H	H	D	H
1	1	H	H	H	D

Where "D" is the DATA presented to the data input.

ENCODER/MULTIPLEXERS

ENCODERS:

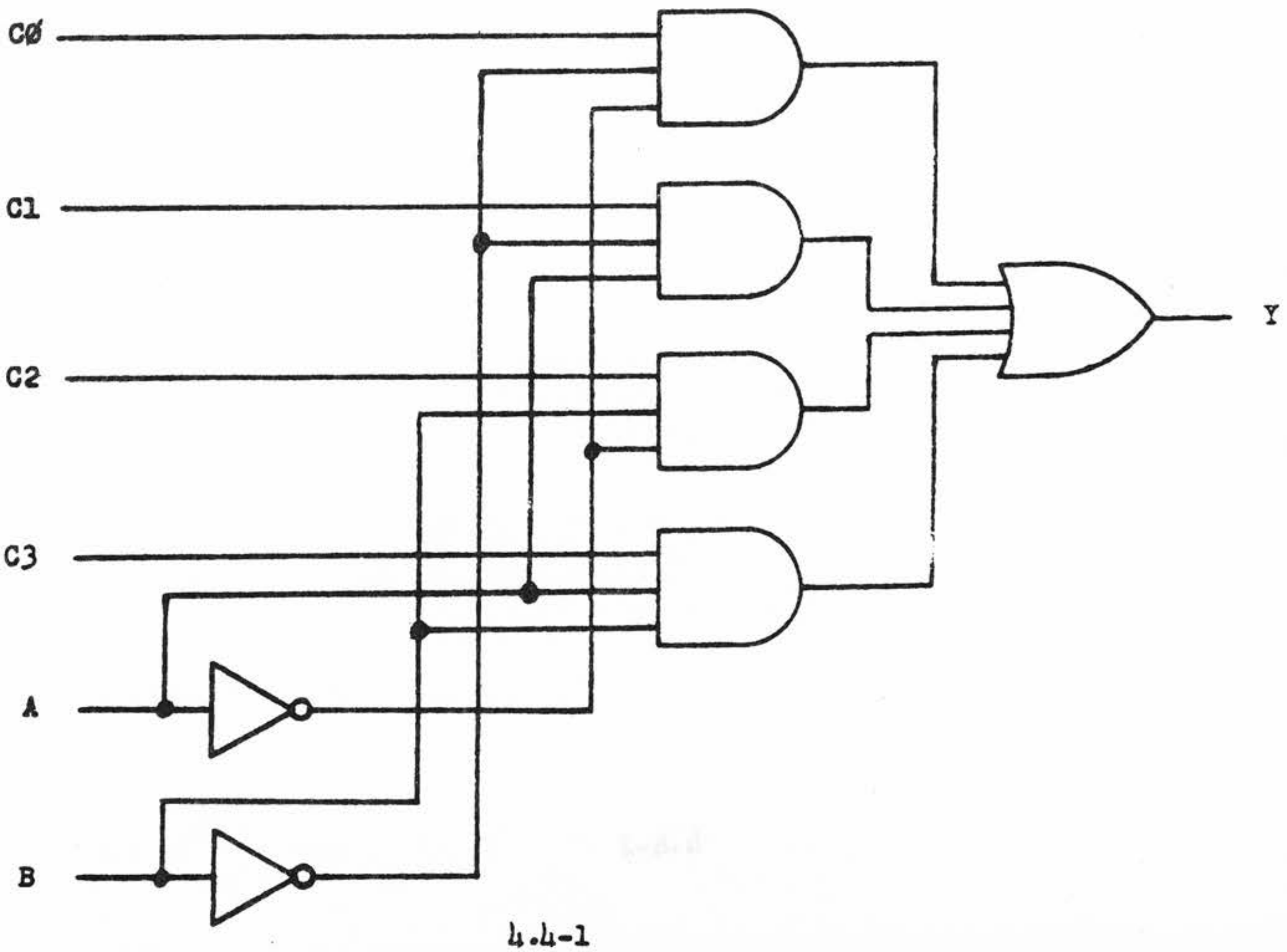
These devices are used to convert several inputs into a few encoded lines. These are used on keyboards and multi-position switches.



1	2	3	B	A
L	L	L	0	0
H	L	L	0	1
L	H	L	1	0
L	L	H	1	1

MULTIPLEXERS:

Multiplexers or Data Selectors are used to select one of several data sources and place the data from that source onto a single output line. These are available in 4 to 1, 8 to 1, and 16 to 1 configurations.



B	A	CØ	C1	C2	C3	Y
0	0	0	X	X	X	0
0	0	1	X	X	X	1
0	1	X	0	X	X	0
0	1	X	1	X	X	1
1	0	X	X	0	X	0
1	0	X	X	1	X	1
1	1	X	X	X	0	0
1	1	X	X	X	1	1

X = DON'T CARE

INTERFACE DEVICES

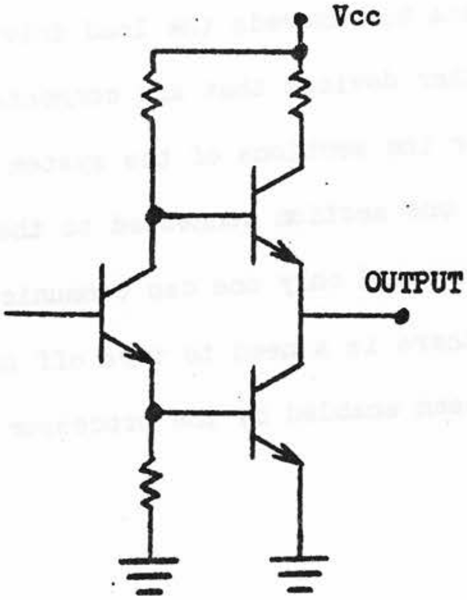
OPEN-COLLECTOR LOGIC

There are several instances where a large multiple input OR gate is needed. In certain cases the common practice is to create a WIRED-OR. This is done by wiring two or more gate outputs together to create a single input into another gate. The WIRED-OR is a LOW-TRUE output. The WIRED-OR is used frequently to OR several interrupts together.

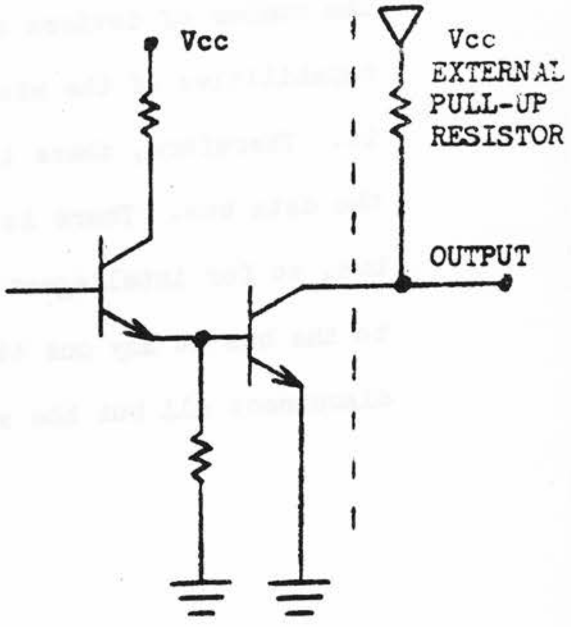
BUT, this procedure CANNOT be done with just any logic gate. The standard TTL logic gate has both active pull-up and active pull-down. Therefore, if two standard TTL outputs were tied together and one output was high, while the other was low; each gate would try to make it's output prevail until finally one of the output transistors of one of the gates burned out.

The only type of logic that can be WIRED-OR together is OPEN-COLLECTOR logic. The internal differences of the output driving circuits is shown below:

STANDARD TTL
OUTPUT DRIVER



OPEN-COLLECTOR
OUTPUT DRIVER



NOTE: The open-collector logic has no internal pull-up device (neither active nor passive). Therefore, the gate can ONLY pull what is attached to its output to ground. Since there is no pull-up device in the gate, several of these types of outputs can be wired together with no ill effects. But their combined outputs must have two states (high and low) to be of any use as an input to another gate. Therefore, an external pull-up resistor must be added to the junction of the WIRED-OR. The value of the resistor is calculated by:

$$\frac{2.6}{I_{TL}} > R > \frac{4.6}{I_S - I_{TL}}$$

WHERE:

I_{TL} = Total of the leakage currents of all the gates of the WIRED-OR when their outputs are all high.

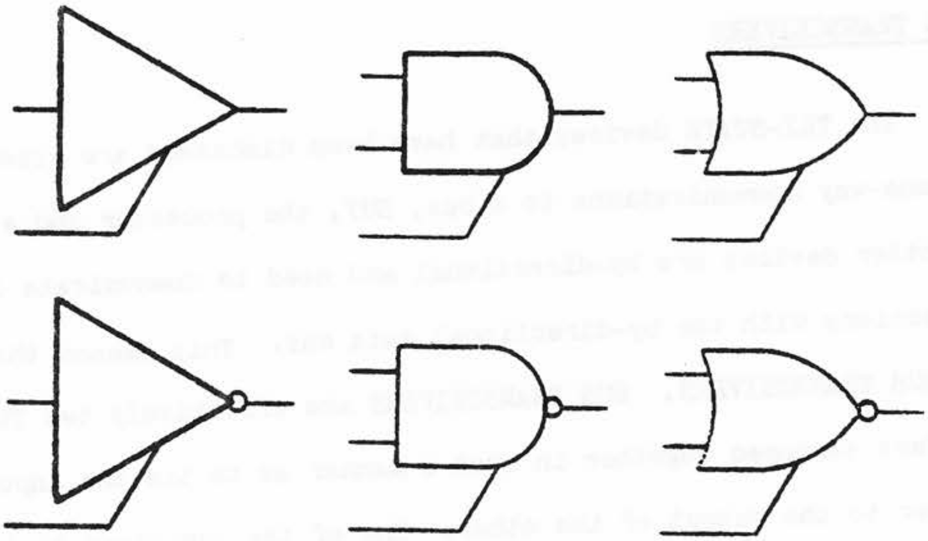
I_S = The lowest maximum current sinking capability of any of the gates forming the WIRED-OR when its output is low.

TRI-STATE LOGIC

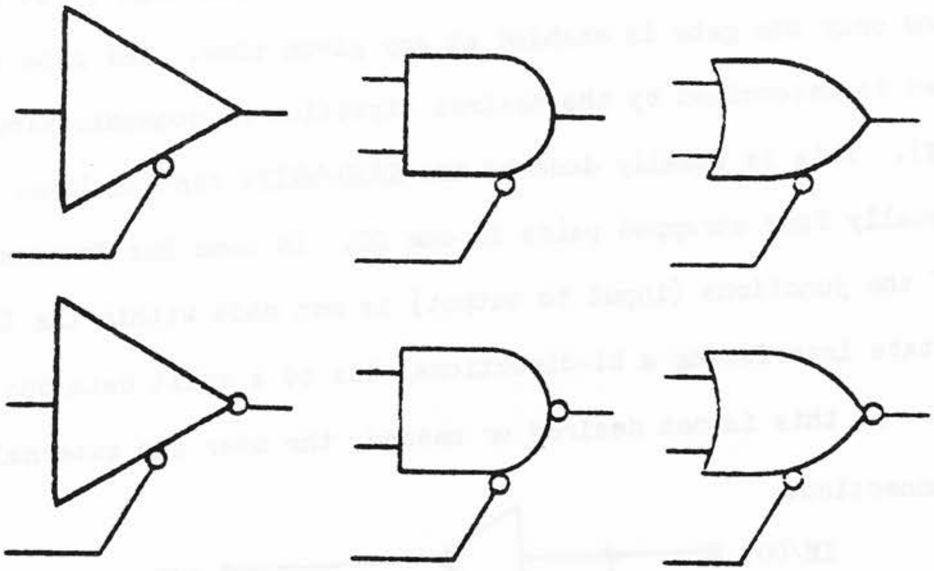
In a micro-computer system transferring of data from one part of the system to another is done via the DATA BUS. In a large number of systems, the number of devices attached to the data bus exceeds the load driving capabilities of the microprocessor or other devices that are connected to it. Therefore, there is a need to buffer the sections of the system to the data bus. There is always more than one section connected to the data bus, so for intellegent communications, one and only one can communicate to the bus at any one time. Therefore, there is a need to turn off or disconnect all but the section that has been enabled by the processor.

But a large number of devices only have two output states (high or low). So, there is a need for a special output that has three states (high, low, or off). This is referenced to as three-state or TRI-STATE logic. The logic symbols for these devices are below:

HIGH-TRUE ENABLES



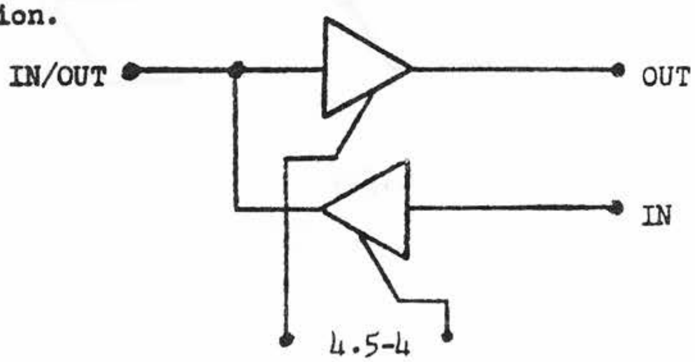
LOW-TRUE ENABLES



These gates when enabled, through the separate enable input, will function like the standard gates that we have already discussed. But, when they are disabled their outputs go to a high-impedance or off-state. Therefore, many 3-state devices can be attached to a common line without unwanted interaction as long as one and only one is enabled to output to that line at any given time.

BUS TRANSCEIVERS

The TRI-STATE devices that have been discussed are essential to one-way communications to a bus, BUT, the processor and a number of other devices are by-directional and need to communicate in both directions with the by-directional data bus. This caused the creation of BUS TRANSCEIVERS. BUS TRANSCEIVERS are effectively two TRI-STATE buffers strapped together in such a manner as to tie the input of each buffer to the output of the other. One of the junctions is to be attached to one of the data bus lines while the other junction is attached to the same respective line of a device or section that is to be buffered. One and only one gate is enabled at any given time. The gate that is enabled is determined by the desired direction of communications (IN or OUT). This is usually done by the READ/WRITE control line. There are usually four strapped pairs in one IC. In some Bus Transceivers, one of the junctions (input to output) is not made within the IC to facilitate interfacing a bi-directional bus to a split data bus or device. If this is not desired or needed, the user can externally make the connection.



ANALYZING SOFTWARE PROBLEMS

ANALYZING SOFTWARE PROBLEMS

INTRODUCTION

The object of this chapter is to present a general procedure used to design software to solve a problem. This procedure is completely machine independent, and it can be applied to any software problems you are likely to encounter. The most important thing to remember about this procedure is that you do not concern yourself with the programming language details until well into the solution. This is true of even the seemingly "trivial" programs. There is no way more certain to result in a program that is sloppy, ill-designed, and hard to debug than to try to write the program directly from the problem definition. To be effective software must be designed first and then implemented using the correct techniques.

5.0 The Software Design Procedure

The systematic approach to developing a programmed system is a logical extension of the normal problem solving cycle engineers and scientists have employed for years. It consists of seven basic steps:

1. problem definition,
2. problem partitioning,
3. algorithm development for each partition,
4. writing the program for each partition,
5. debugging each program,
6. integrating the programs back into the system, and
7. final system debug.

Using this technique, the problem is broken down into smaller and smaller sub-problems until they are a size which you can deal with conveniently and effectively. This is because it is much easier to focus your attention on one small section of the system at a time. You develop each of these blocks and sub-blocks into a group of detailed flowcharts and programs, each of which is tested and debugged. They are then interfaced and the whole system tested. This systematic approach is intended to help you minimize errors, since the small highly localized programs are much easier to thoroughly check out than a single large, spread out program.

Graphically, the procedure is illustrated in Figure 5.1. You start with a central problem and partition it into logical blocks, solve and debug each of the blocks, and finally integrate and refine the blocks into the final system. There may be one or many levels of blocking, depending on the complexity of the problem. With experience, you will find this general approach to be the most direct and consistent way to implement a working software system, regardless of size. Less organized approaches may work for smaller systems, but you will become hopelessly tangled as the systems grow in size. It is best to learn the general procedure and use it on all problems, small or large. The greatest disasters usually occur when the whole design procedure is dispensed with because the problem is too "trivial" to warrant the general approach. Conversely, dogged application of this approach can make many formidable problems turn out far better and faster than anticipated.

In the remainder of this lesson we will initiate our study of the general software solution procedure. Lessons Three through Ten will then expand and refine the techniques used during the solution process.

5.1 Step 1: Define the Problem

As with any procedure for solving any problem, the first step is always the same (and the hardest): define the problem. For the case of software problems, you must decide exactly what the finished software system is to do. This definition of the operational characteristics you want the final system to have is called the functional specification. Naturally, it is easier to define and specify solutions for some type of problems than for others. Problems which are concerned with the implementation of specific features are generally easiest. Problems which require both judgement and implementation are the hardest. In the first case, the task is to figure out how to do something. In the later case, it is often a question of whether or not the job can be done, and if it can, what is the best way to do it. For example, a program to write single data bytes onto a magnetic tape unit is a fairly specific problem with a similarly straightforward functional specification. There is little conceptual design work to be done. It is mainly a question of using a program to control the

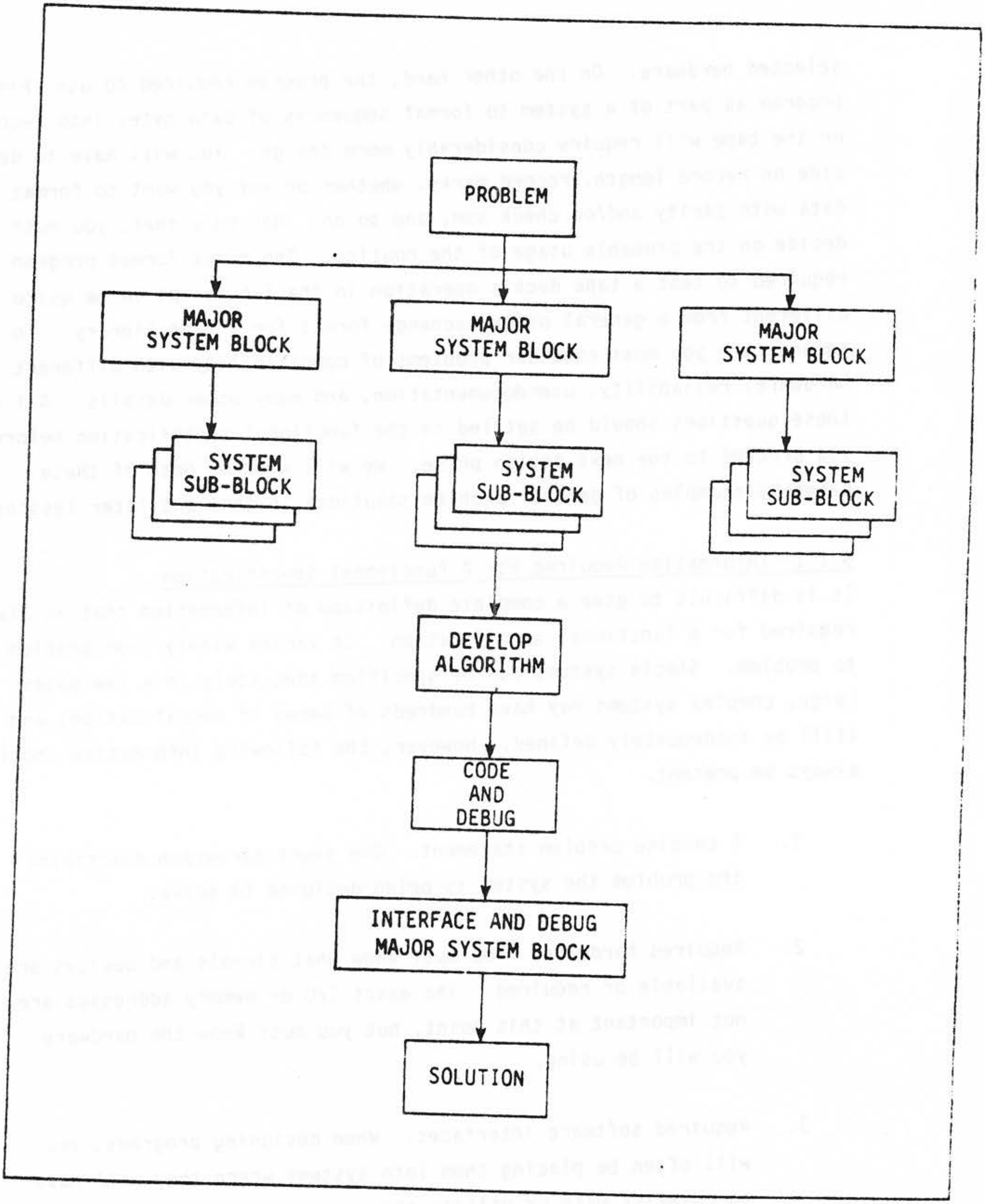


FIGURE 5.1 GENERAL PROBLEM SOLUTION PROCESS

selected hardware. On the other hand, the program required to use this program as part of a system to format sequences of data bytes into records on the tape will require considerably more design. You will have to decide on record length, record marks, whether or not you want to format the data with parity and/or check sum, and so on. Not only that, you must decide on the probable usage of the routine. The quick format program required to test a tape deck's operation in the lab is apt to be quite different from a general usage exchange format for a tape library. In the second case you must consider problems of compatibility with different hardware, reliability, userdocumentation, and many other details. All of these questions should be settled in the functional specification before you proceed to the next design phase. We will examine both of these cases as examples of general problem solutions in this and later lessons.

5.1.1 Information Required For A Functional Specification

It is difficult to give a complete definition of information that is always required for a functional specification. It varies widely from problem to problem. Simple systems can be specified adequately in a few pages. Large, complex systems may have hundreds of pages of specifications and still be inadequately defined. However, the following information should always be present.

1. A concise problem statement. One short paragraph describing the problem the system is being designed to solve.

2. Required hardware. You must know what signals and devices are available or required. The exact I/O or memory addresses are not important at this point, but you must know the hardware you will be using.

3. Required software interfaces. When designing programs, you will often be placing them into systems where they will have to co-exist with or utilize other programs. If this is to be the case, it should be noted in the specification. In this case, exact details are necessary; you should mention the

relevant system standard or format (i.e., all output must conform to system I/O standard 1-13) for all routines to be interfaced. These requirements will often have a significant affect on your design.

4. A complete description of how the system is supposed to function when complete. This is usually the longest part of the functional specification. This section should include a description of user interaction (if any), data required, output produced, special features, error condition handling, etc. In other words, a complete description of how the system will look to the world from the outside with no consideration for how it will look from the inside.

This problem makes writing the software specification sound like a rather formidable task. It is. A good, well thought out specification is the key to a good (i.e. successful!) project. It is well worth the time required to think the problem through completely. If you know what you have to do, it becomes much easier to proceed directly to a solution than when you must constantly stop and start to fill in the blanks in the problem definition. Few specifications are ever totally complete, but you should strive to get as close as possible before you start the actual design. Once you become immersed in the details of the solution, it becomes much more difficult to separate the normal implementation problems from those caused by a fundamental design logic error.

Example 5.1

Consider the design of a program to interface a magnetic tape recorder to microcomputer. This program will control the transfer of parallel data between the tape deck and the microcomputer. It will control all tape deck hardware functions which are required to perform these data transfers. The following is a possible functional specification.

Scope: This specification covers the program required to interface a Magbyte Model 1010 digital magnetic tape drive to an everyday microcomputer.

Required Hardware: The interface will require the tape drive to be connected to the computer through two input ports and two output ports: one data input port with parallel data from the tape deck, one data output port with parallel data out to the tape deck, one status input port and one control output port. Status input signals available are End of Tape, Write Protect and Ready. Control signals required are Tape Advance, Read/Write and Transfer Data. The timing waveforms are shown in Figure 5.2.

Software Requirements: The I/O routines must conform to the normal system requirements: output data to be passed via the C register (or the appropriate register or memory location for your system) and input data is to be returned in the A register (or the appropriate register or memory location for your system) upon exit. The routines must restore any registers or memory locations used.

Operational Description:

Input Operation: Upon call, the routine will generate all timing and control signals required to transfer one data byte from the tape in the tape drive into the processor. It will then return to the calling program with that data byte. If the tape drive status indicates End of Tape, an error indicator should be set on return. Otherwise it should be reset.

Output Operation: Upon call, the routine will generate all timing and control signals required to transfer the data byte passed from the calling program onto the magnetic tape in the tape drive. If the tape drive status indicates End of Tape or Write Protect, an error indicator should be set on return. Otherwise it should be reset.

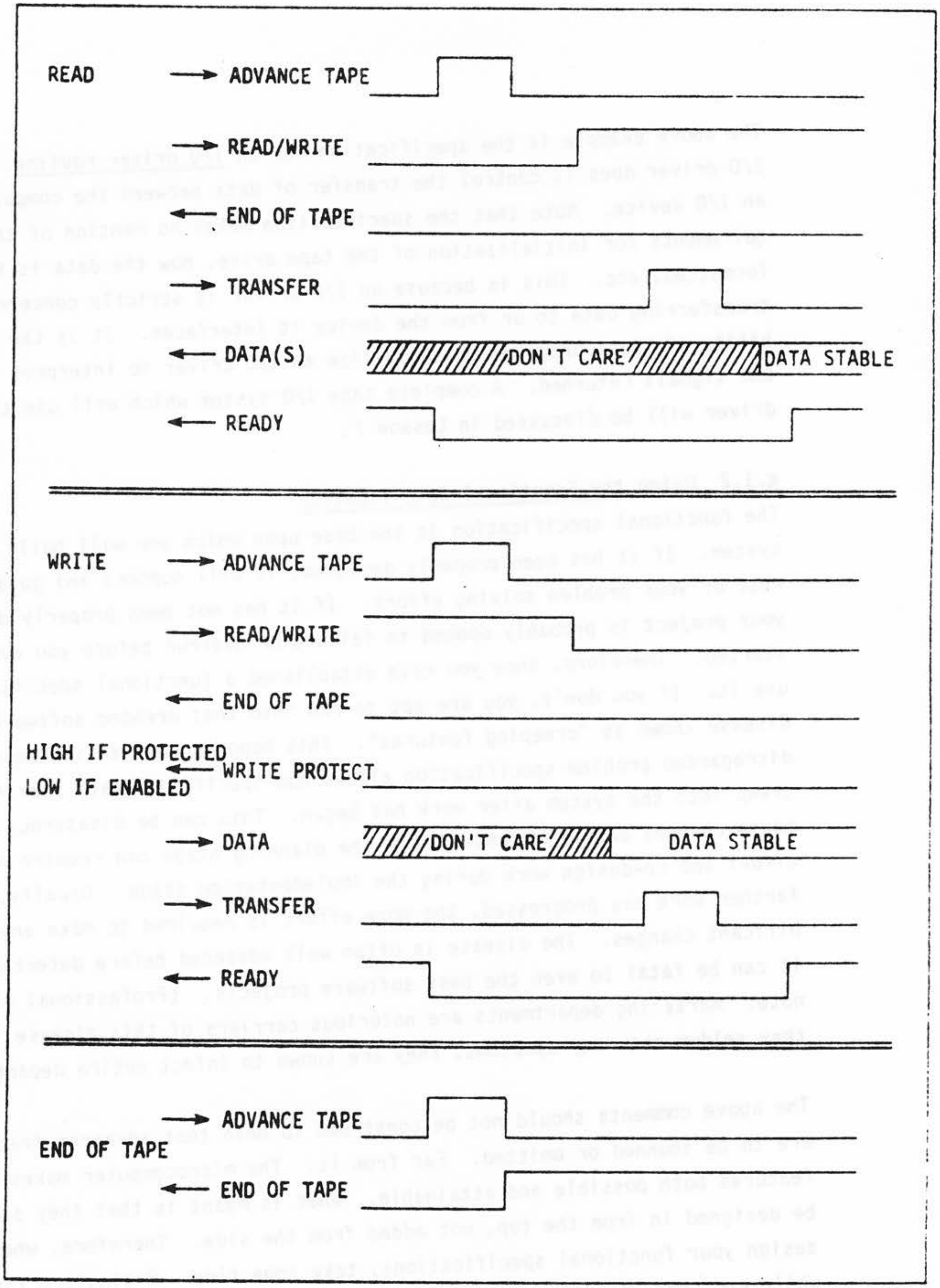


FIGURE 5.2 TAPE DECK TIMING WAVEFORMS

The above example is the specification for an I/O driver routine. All an I/O driver does is control the transfer of data between the computer and an I/O device. Note that the specification makes no mention of the requirements for initialization of the tape drive, how the data is to be formatted, etc. This is because an I/O driver is strictly concerned with transferring data to or from the device it interfaces. It is the responsibility of those programs which utilize an I/O driver to interpret the data and signals returned. A complete tape I/O system which will use this driver will be discussed in Lesson 7.

5.1.2 Using the Functional Specification

The functional specification is the base upon which you will build your system. If it has been properly designed, it will support and guide the rest of your problem solving effort. If it has not been properly designed, your project is probably doomed to failure or overrun before you even get started. Therefore, once you have established a functional specification, use it. If you don't, you are apt to run into that dreaded software disease known as "creeping features". This happens when an inadequate or disregarded problem specification allows non-specified "neat" features to creep into the system after work has begun. This can be disastrous, because changes easily accommodated in the planning stage can require massive effort and re-design work during the implementation stage. Usually, the farther work has progressed, the more effort is required to make any significant changes. The disease is often well advanced before detected and it can be fatal to even the best software projects. (Professional engineers note: marketing departments are notorious carriers of this disease. While they seldom show any symptoms, they are known to infect entire departments.)

The above comments should not be construed to mean that advanced features are to be shunned or omitted. Far from it. The microcomputer makes these features both possible and attainable. What is meant is that they should be designed in from the top, not added from the side. Therefore, when you design your functional specifications, take some time. Brainstorm for a while and come up with a list of features that the system can really accomplish. Try trading off some hardware and software to lower cost or increase system performance. Microcomputers make whole new fields of features possible, and it is worth your time to see if you can find some for your project. But once that functional specification is done, stick with it. If really drastic changes are needed, you will probably be better off starting over than trying to patch an inadequate specification.

5.2 Step 2: Partition The Problem Into Function Blocks

Once you have completed the functional specification for your system, you can begin to partition it into operational blocks. An operational block is a section of the system which is responsible for performing some specific system function. Operational blocks can be as complex as a complete floating point arithmetic package or as simple as a few instruction data conversions. In system operations, control passes from one functional block to another as the program executes. In this respect the block diagram can actually be considered as a type of overall system flowchart. It differs from the flowchart in that it does not specify the actual algorithms used to implement the functions (see Section 5.3). You first design the structure of the program as a series of successively more detailed operational blocks until you reach a level of complexity that you can deal with effectively. You then proceed to algorithm development for each block.

Blocking and partitioning are the cornerstone of converting a functional specification into a functional program. You can have as many levels of blocks and sub-blocks as the problem requires. When you are first learning, you should not hesitate to block down to sections which seem almost trivial. As you gain experience you will be able to judge more accurately what size blocks you can comfortably handle. Also, extremely involved or complex sections of a system may require much more detailed blocking than the more straightforward sections. The flexibility of blocking is that it allows you to easily adjust the level of detail to match the complexity of the problem.

5.2.1 Deciding on the System Blocks

The decision of what blocks to divide the system into initially is usually made by referring to the characteristics defined in the functional specification. Some common initial blocks are:

a. input operations,

b. program functions (transfer data, search memory, do arithmetic, etc.),

c. system control and timing,

d. output operation, and

e. major data structures (tables, lists, etc.).

These blocks are then drawn and interconnected to form the system block diagram. It is important to remember that at this initial point you are concerned with identification of the major system structures. You are not yet concerned with their actual operation. The design of how the operational blocks will implement their functions will commence once the overall system structure has been established. In theory, it should be possible to implement the system in either hardware, software, or some combination of hardware and software at the end of the blocking operation. This leaves you with the maximum flexibility for actual system implementation.

Example 5.2

Let's take the specification for the magnetic tape I/O driver we wrote in Example 5.1 and do the block diagrams for that system. We can see from the functional specification that we will require blocks to input data, output data, and control the data transfers. Figure 5.3a shows an initial block diagram for this simple system. Note that it shows all data and control signals that are passed through the system. Since the data is transferred to and from the tape deck in parallel form, no further blocking is needed for the Input and Output blocks. However, the control block is required to perform several operations. It must detect end of tape, control the read/write line, sense a write protect condition, and advance the tape. This block is sufficiently complex to warrant sub-blocking. It is shown sub-blocked in Figure 5.3b. Note how all inputs and outputs of the sub-block diagram match those on the main block diagram. It is the same interface expanded to show more detail.

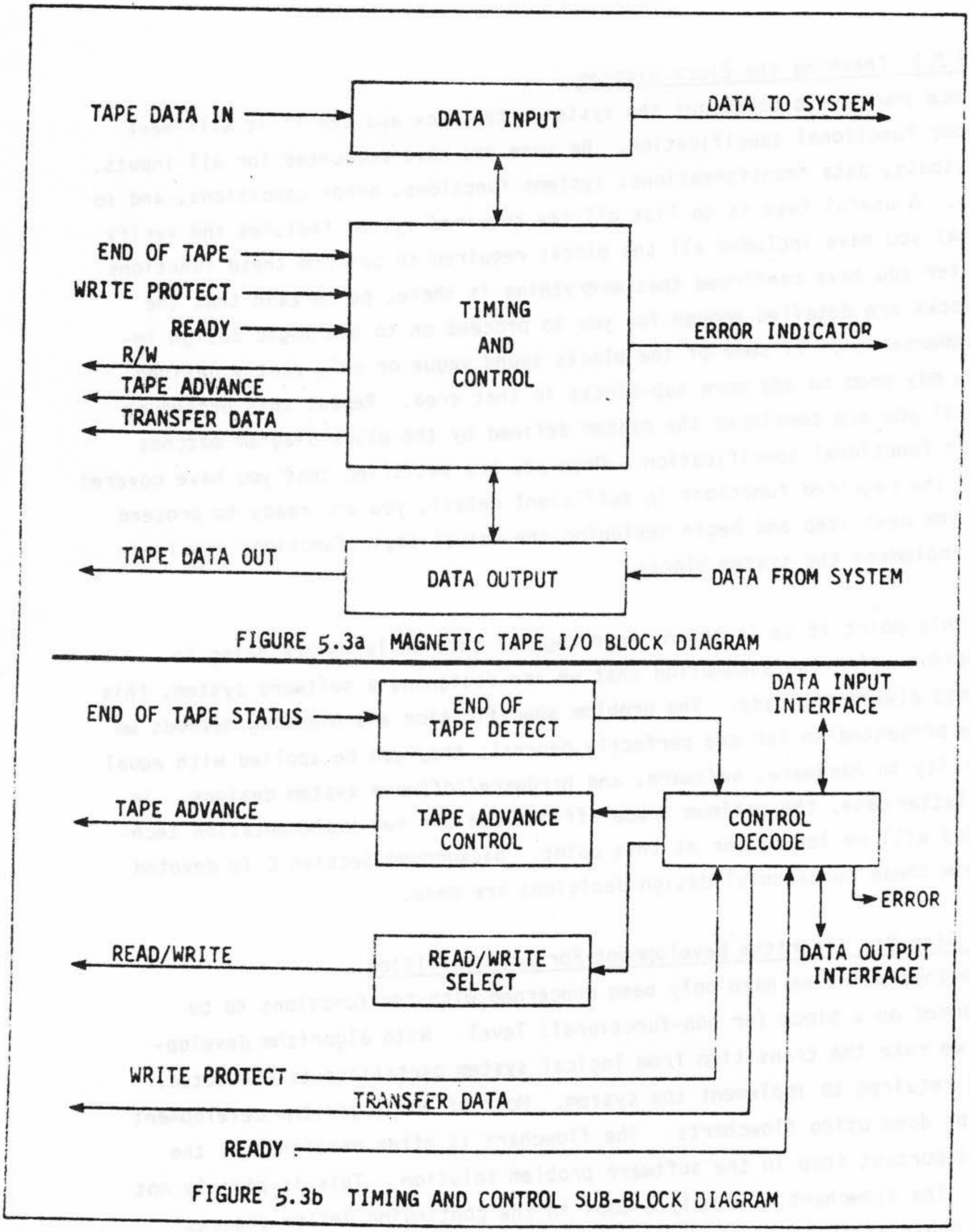


FIGURE 5.3a MAGNETIC TAPE I/O BLOCK DIAGRAM

FIGURE 5.3b TIMING AND CONTROL SUB-BLOCK DIAGRAM

FIGURE 5.3

5.2.2 Checking the Block Diagram

Once you have blocked out the system, step back and see if it will meet your functional specification. Be sure you have accounted for all inputs, outputs, data transformations, systems functions, error conditions, and so on. A useful test is to list all the required system features and verify that you have included all the blocks required to perform these functions. After you have confirmed that everything is there, be certain that the blocks are detailed enough for you to proceed on to the logic design implementation. If some of the blocks sound vague or only partly defined, you may need to add more sub-blocks in that area. Repeat this procedure until you are convinced the system defined by the block diagram matches your functional specification. Once you are satisfied that you have covered all the required functions in sufficient detail, you are ready to proceed to the next step and begin designing the actual logic functions required to implement the system blocks.

At this point it is important to recognize that while we are going to continue using the assumption that we are designing a software system, this is not always the case. The problem specification and blocking methods we have presented so far are perfectly general; they can be applied with equal facility to hardware, software, and hardware/software system designs. In the latter case, the optimum trade off between the two implementation techniques will be looked for at this point. Background Section C is devoted to how these fundamental design decisions are made.

5.3 Step 3: Algorithm Development For Each Partition

Up to this point we have only been concerned with the functions to be performed on a block (or non-functional) level. With algorithm development we make the transition from logical system partitions to the actual logic required to implement the system. Most of our algorithm development will be done using flowcharts. The flowchart is often mentioned as the most important step in the software problem solution. This is plainly not true. The flowchart is simply a tool in the continuing design process which began with the problem specification. It is no more correct to sit down and start drawing flowcharts than it is to sit down and start writing machine code. Both operations have their place in the problem solution procedure. Neither is satisfactory alone. Flowcharts are one possible way to conveniently develop and check the logic of the problem blocks for correct operation. Using flowcharts it is possible to develop program logic independent of any specific computer. It is also much easier to find logic errors in the symbolic flowchart than to try and hunt them down once they are committed to program implementation. (This is particularly true with the relatively primitive debug facilities currently provided by microprocessor manufacturers.)

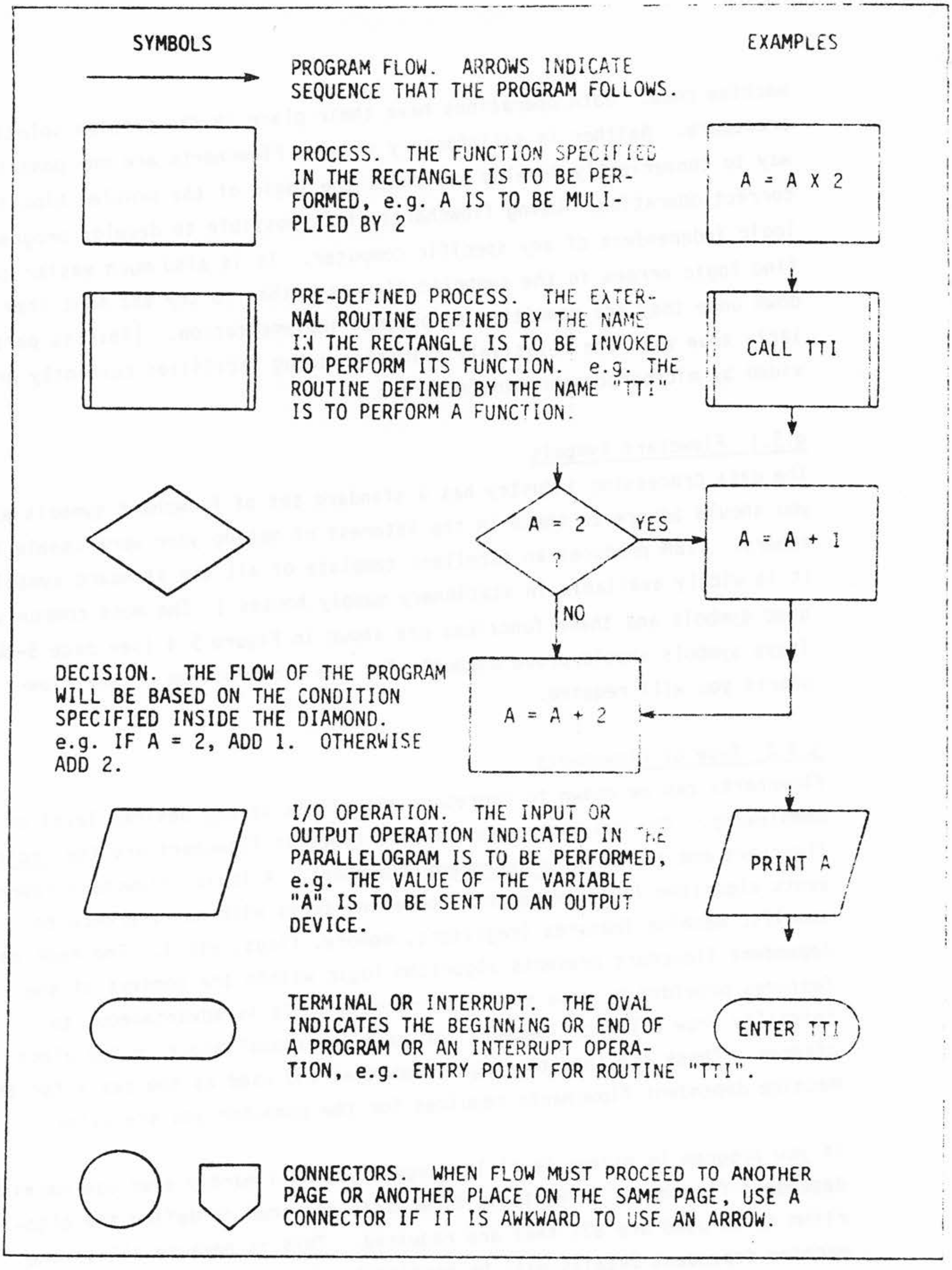
5.3.1 Flowchart Symbols

The data processing industry has a standard set of flowchart symbols and you should adhere to these in the interest of making your work usable to others. (IBM produces an excellent template of all the standard symbols; it is widely available in stationery supply houses.) The most commonly used symbols and their functions are shown in Figure 5.4 (see page 5-14). These symbols should prove adequate for the construction of any flowcharts you will require.

5.3.2 Type of Flowcharts

Flowcharts can be drawn to represent algorithms at any desired level of complexity. The two most commonly used types of flowchart are the logic flowchart and the machine dependent flowchart. A logic flowchart represents algorithm logic in general operating terms with no reference to specific machine features (registers, memory, flags, etc.). The machine dependent flowchart presents algorithm logic within the context of the features provided by some specific processor. It is advantageous to initially draw a logic flowchart for each functional block in the block diagram. These are then thoroughly debugged and used as the basis for the machine dependent flowcharts required for the computer you are using.

If you program in higher level languages, you will hardly ever use machine dependent flowcharts. The logic flowcharts required to define the algorithm to be used are all that are required. This is because all of the machine dependent details will be handled by the language processor.



PROGRAM FLOW. ARROWS INDICATE SEQUENCE THAT THE PROGRAM FOLLOWS.

PROCESS. THE FUNCTION SPECIFIED IN THE RECTANGLE IS TO BE PERFORMED, e.g. A IS TO BE MULIPLIED BY 2

PRE-DEFINED PROCESS. THE EXTERNAL ROUTINE DEFINED BY THE NAME IN THE RECTANGLE IS TO BE INVOKED TO PERFORM ITS FUNCTION. e.g. THE ROUTINE DEFINED BY THE NAME "TTI" IS TO PERFORM A FUNCTION.

DECISION. THE FLOW OF THE PROGRAM WILL BE BASED ON THE CONDITION SPECIFIED INSIDE THE DIAMOND. e.g. IF A = 2, ADD 1. OTHERWISE ADD 2.

I/O OPERATION. THE INPUT OR OUTPUT OPERATION INDICATED IN THE PARALLELOGRAM IS TO BE PERFORMED, e.g. THE VALUE OF THE VARIABLE "A" IS TO BE SENT TO AN OUTPUT DEVICE.

TERMINAL OR INTERRUPT. THE OVAL INDICATES THE BEGINNING OR END OF A PROGRAM OR AN INTERRUPT OPERATION, e.g. ENTRY POINT FOR ROUTINE "TTI".

CONNECTORS. WHEN FLOW MUST PROCEED TO ANOTHER PAGE OR ANOTHER PLACE ON THE SAME PAGE, USE A CONNECTOR IF IT IS AWKWARD TO USE AN ARROW.

FIGURE 5.4 FLOW CHART SYMBOLS

Similarly, general algorithms and problem solutions which are to be implemented on a variety of computers are best presented using logic flowcharts. Any user can then take the general logic flowchart and use it as the basis for the implementation of a solution on any computer or in any language. As you gain experience with your particular installation, you will be able to go directly from the block diagrams to flow charts that are a cross between purely logical and purely machine dependent flowcharts. However, if you intend to save the algorithm or solution for documentation or possible use on some other system, it would be a good idea to draw a good logic flowchart after the system is completed.

5.3.3 How to Design Algorithms

The design of program algorithms is actually the design of software, a vast subject indeed. We will be covering a portion of that subject in the next eight lessons. However, we can discuss some of the general procedures used when translating a logical system block to an algorithm.

1. Decide what the block is to do. This is the same step as when we initially specified the problem. The only difference is that it is now being done for a small, local program rather than for the whole system. Naturally, the label on the block will provide a good starting place for this description. Usually a one or two sentence description of the operation to be performed is all that is required.

2. Decide where the data to be operated upon is located. Is it read in, passed from another block, looked up in a table, or what? You will need operation blocks to input the required data. While you decide where to get the data, decide if you need to do anything special to it before you use it. Does it have to be complemented? Rotated? Masked? Scaled? If so, you know you will need some data transformation blocks in the flowchart.

3. Figure out how to perform the required operation. This is the real meat of the algorithm development. This will be where you combine process blocks, data and decisions to convert the data from the input

format to the output format. This part of the process will usually account for the largest portion of your work. Remember, develcping the algorithm is an iterative process.

It will usually take several tries before you get the algorithm correct. Start out by writing down the sequence of operations to be performed in the order they must be performed, like "read in data, then test for control characters, then test for lower case characters", and so on. This will give you that all important feel for the sequences of actions which are to be performed. After you have the general flow, add the process and decision blocks you need to actually perform the operations.

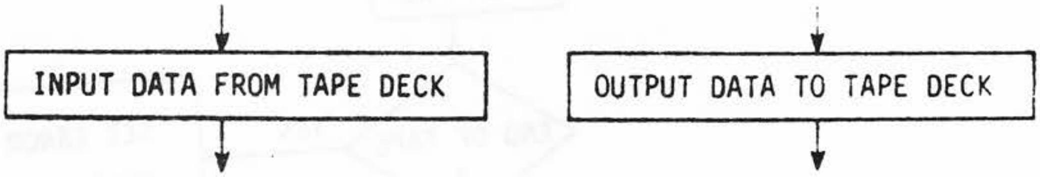
After you have an algorithm that should work, try it out with data to see if it does work (all on paper, of course). Try to imagine every possible data condition that could occur and then be sure your algorithm can process it correctly. You must be certain your logic is correct in the algorithm before you proceed to coding. Be particularly careful that your algorithm can handle error conditions. This is an area which is particularly susceptible to errors which will be very hard to detect. Be patient and thorough. Time spent getting the logic correct in the algorithm will be time saved during system debugging. Think first, program later.

4. Decide what to do with the finished data. Does it have to be specially formatted? Do you save it? Pass it back to a calling routine? Output it? Add the blocks required to get the output data ready for the receiving routine or device.

5. Keep the structure simple. Make it a goal to keep the flow straightforward, logical and clear. Be particularly careful about how you enter and exit from the routines. There are really only a few simple structures you should ever need to use in construction of any algorithm. We will examine these structures in the next few lessons.

Example 5.3

Let's develop the algorithms required for our magnetic tape drive interface system used in Examples 5.1 and 5.2. The first thing that becomes apparent is that the data input and output blocks are very large blocks and very small programs. The data is to pass through the routine in parallel without being modified. Thus the flow charts for those blocks would be simply one block each:



The obvious conclusion is that the majority of these flowcharts will be concerned with when to read and write the data, namely the timing and control blocks. Let's take the read block first. From the timing diagram we can see that for this tape deck the sequence of control for reading a data byte from the tape is advance the tape (from the manufacturer's specification we find that it automatically advances in one byte increments), test for End of Tape, set the Read/Write line to Read, wait for data ready, read the data, then exit. The algorithm for this function is shown in the logic flowchart in Figure 5.5. Note how the flowchart defines a logical solution to the problem without reference to any specific hardware.

A similar procedure is used to design the algorithm for writing data. For Write operation the timing waveform specifies that we advance the tape, test for End of Tape, test for Write Protect, set the Read/Write line to Write, set up the output data, strobe the data transfer line, wait for Data Ready and exit. This flowchart is shown in Figure 5.6. Using these two logic flowcharts we could then draw the machine dependent flowcharts or proceed directly on to the actual program.

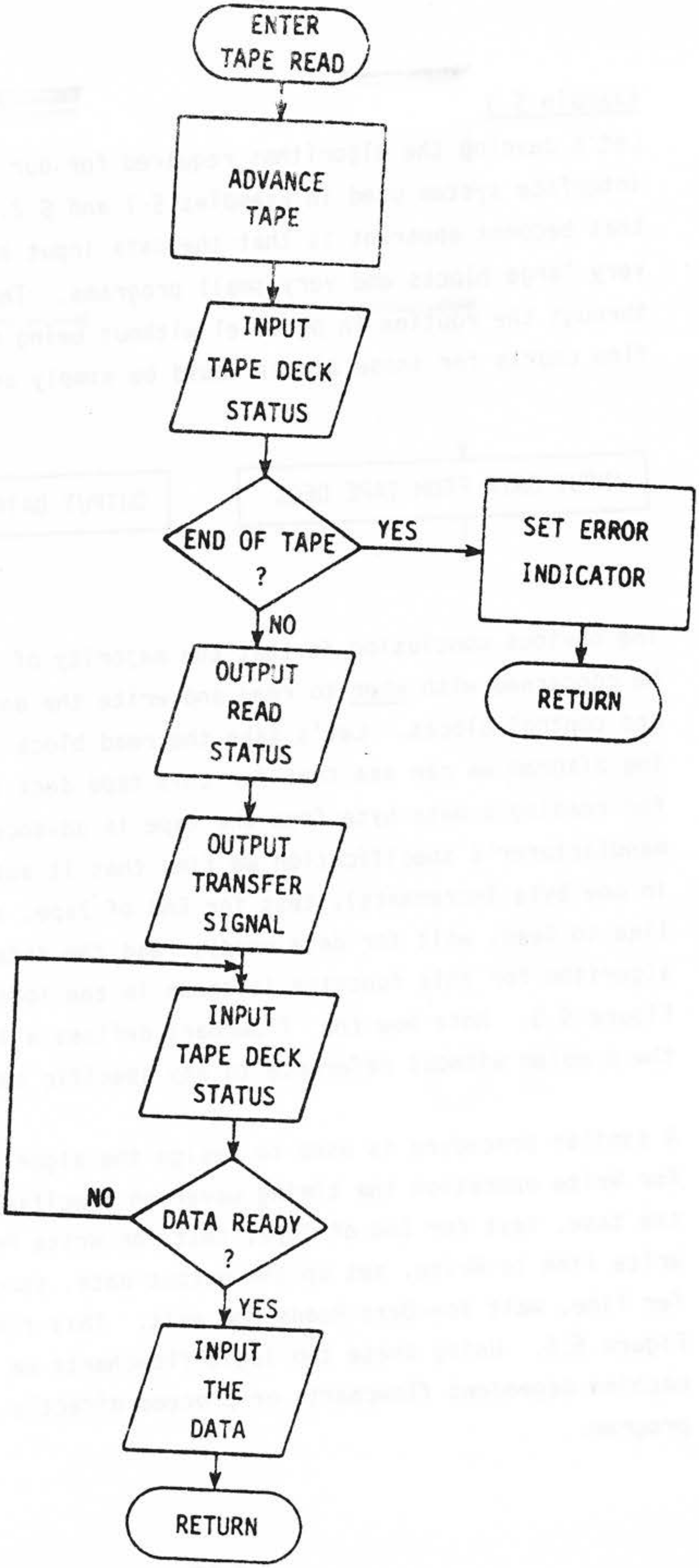


FIGURE 5.5 TAPE DECK READ LOGIC FLOW CHART

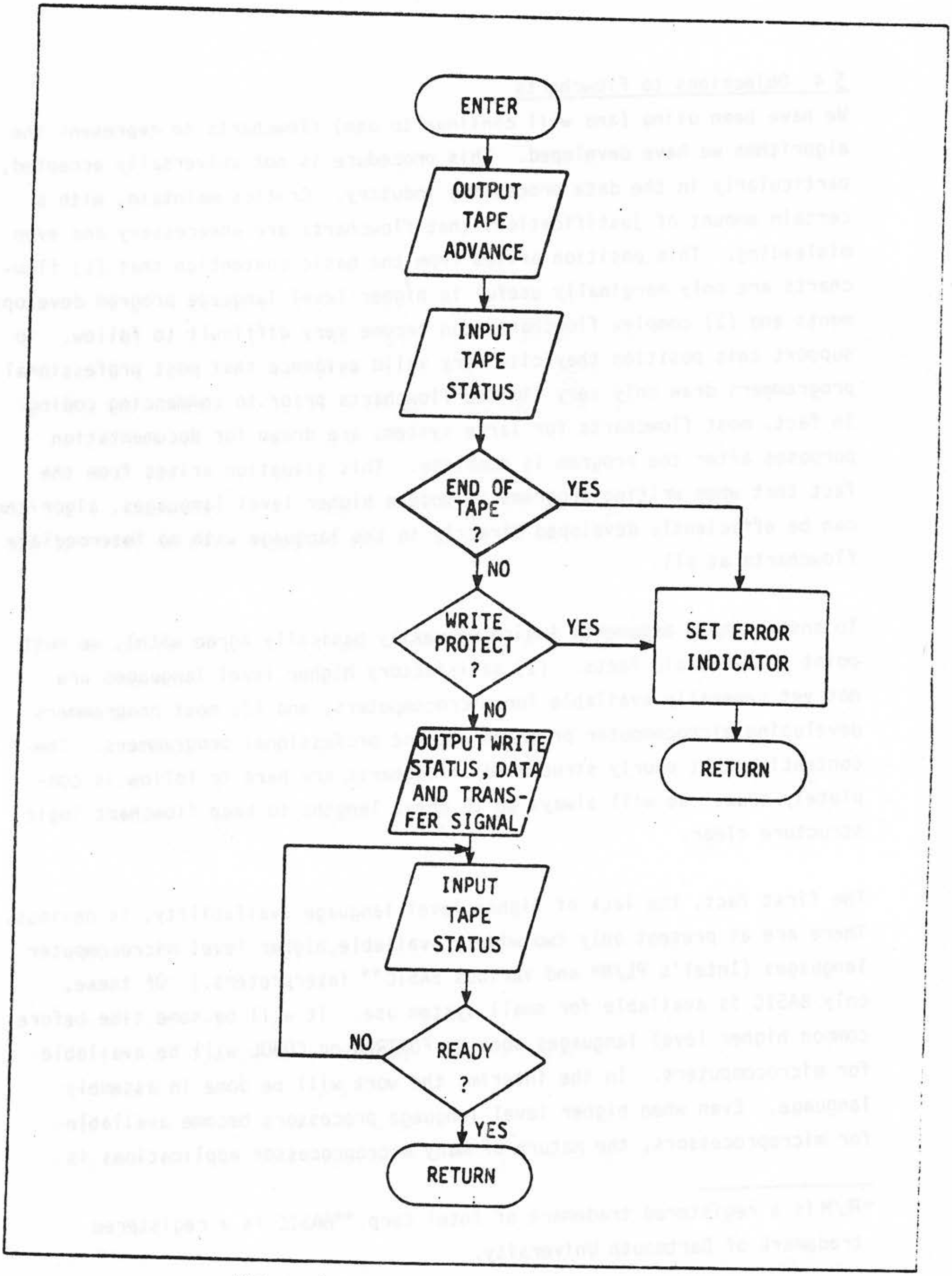


FIGURE 5.6 TAPE WRITE LOGIC FLOW CHART

5.4 Objections to Flowcharts

We have been using (and will continue to use) flowcharts to represent the algorithms we have developed. This procedure is not universally accepted, particularly in the data processing industry. Critics maintain, with a certain amount of justification, that flowcharts are unnecessary and even misleading. This position arises from the basic contention that (1) flowcharts are only marginally useful in higher level language program developments and (2) complex flowcharts can become very difficult to follow. To support this position they cite very valid evidence that most professional programmers draw only very limited flowcharts prior to commencing coding. In fact, most flowcharts for large systems are drawn for documentation purposes after the program is complete. This situation arises from the fact that when writing programs in modern higher level languages, algorithms can be efficiently developed directly in the language with no intermediate flowcharts at all.

To answer these arguments (which we really basically agree with), we must point to two basic facts: (1) satisfactory higher level languages are not yet generally available for microcomputers, and (2) most programmers developing microcomputer programs are not professional programmers. The contention that poorly structured flowcharts are hard to follow is completely true. We will always go to great lengths to keep flowchart logic structure clear.

The first fact, the lack of higher level language availability, is obvious. There are at present only two widely available higher level microcomputer languages (Intel's PL/M* and various BASIC** interpreters.) Of these, only BASIC is available for small system use. It will be some time before common higher level languages such as FORTRAN or COBOL will be available for microcomputers. In the interim, the work will be done in assembly language. Even when higher level language processors become available for microprocessors, the nature of many microprocessor applications is

*PL/M is a registered trademark of Intel Corp.**BASIC is a registered trademark of Dartmouth University.

such that a knowledge of assembly language will still be required. Higher level languages are only marginally effective in developing programs for use in control or real time applications. Programs of this type require the complete control of the computer's hardware that assembly language provides. For assembly language, use of the flowchart provides a pseudo higher level language for algorithm development that can be either dependent or independent of the computer to be used. (We will have much more on the higher level-assembly language tradeoffs in Lesson 9.)

That most microcomputer programmers are not professional programmers is also fairly obvious. Most current microcomputer programmers are logic designers and hobbyists, many programming for the first time. Since they will probably be forced to use assembly language, these users will be learning programming, algorithm development, and machine structure all at the same time. The use of assembly language programming and flowcharts will enable us to separate these learning activities. In particular, the initial process of teaching general algorithm development is better presented with general flowcharts than with some specific language. The techniques presented using some specific language may reflect the compromises made by the language rather than those required to solve the problem. After some initial algorithm development training, the user may be able to proceed to flowchart free higher level language programming. For that initial training, however, the logic flowchart is an important teaching tool.

To make maximum use of flowcharts without becoming unduly attached to them we will adopt a carefully structured approach. All algorithms will be presented in general logic flowcharts. We will not use machine dependent flowcharts except in the context of specific examples. All flowchart structures will be chosen from a small group of simple, logically sufficient structures whose use can be directly transferred to most higher level languages. In this way we will make maximum use of flowcharts while avoiding the major objections.

5.5 Procedures After Algorithm Development

After you have completed the problem definition, block diagrams, and algorithms, you can begin to think about writing the program required to implement the logic system you have defined. However, it should be apparent by now that if you have followed the first three steps correctly, this step should present you with very little trouble. The blocking and algorithm steps combined with the flow charts will have supplied the system structure and control logic. All you will need to do is implement these features using the programming language you have available. Naturally, that is easier said than done, but if the logic is correct, the problem will have been reduced to finding combinations of machine instructions or higher level language statements to perform the desired operations. We will spend the next eight lessons refining and expanding your problem solving skills, augmenting these skills with useful programming techniques.

5.6 Summary

This lesson has presented the general approach required to solve software problems. All software problems can be solved by dividing them into blocks and sub-blocks, developing algorithms for those blocks, writing programs to implement the algorithms and interfacing the blocks back into a system which solves the problem. The general approach to problem definition, blocking and algorithm development was then presented and illustrated using the example of a digital Read/Write magnetic tape deck.

QUESTIONS

1. Describe the general software problem solution process. Is this the way you normally approach problems? Do you think the general procedure can be applied to other, non-software problems?

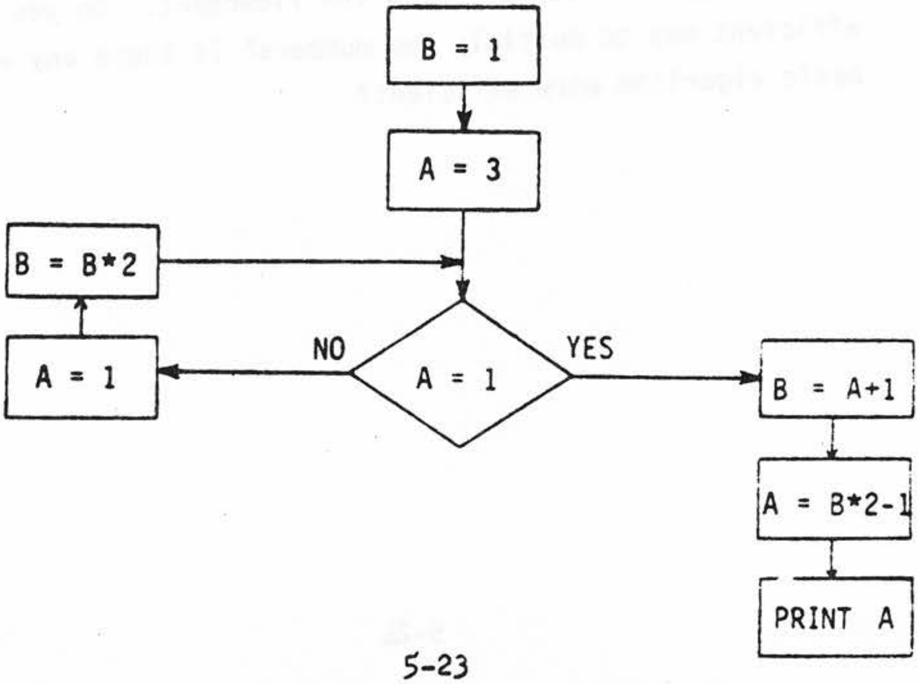
2. Why is it important to establish and follow a functional specification at the outset of the solution to a problem?

3. Describe "creeping features". Have you ever seen it in action? What was the cause? What was the result?

4. Describe the difference between a logic flowchart and a machine dependent flowchart. Which do you usually use? If you usually use a machine dependent flowchart, do you usually draw a logic flowchart of the solution for future use?

PROBLEMS

1. What value of A will be printed in the example flow chart below:



2. The Fibonacci series F(N) is a mathematical number sequence which is defined for all integer values of N by the following algorithm

$$F(0) = 0$$
$$F(1) = 1$$
$$F(N) = F(N - 1) + F(N - 2) \text{ for all } N > 1$$

For example, $F(2) = F(2 - 1) + F(2 - 2)$
$= F(1) + F(0)$
$= 1 + 0$
$= 1.$

Thus the Fibonacci series can be represented as follows

N	0	1	2	3	4	5	6	7	. . .	N
F(N)	0	1	1	2	3	5	8	13	. . .	F(N - 1) + F(N - 2)

Draw the flowchart to compute F(N) for any value of N.

3. Draw a flowchart which incorporates the flowchart developed in Problem 2, to compute and print the first 100 values of N and F(N). (Assume that the command "Print" is sufficient to print a value.)

4. One simple method often used to multiply two numbers together is to repeatedly add one number to itself. For example, 3 * 4 can be thought of as 3 + 3 + 3 + 3 = 12. Develop the algorithm to multiply two numbers using this method. Draw the flowchart. Do you feel this is an efficient way to multiply two numbers? Is there any way to make this basic algorithm more efficient?

THE HARDWARE/SOFTWARE APPROACH TO MICROCOMPUTER DESIGN

THE HARDWARE/SOFTWARE APPROACH TO MICROCOMPUTER DESIGN

INTRODUCTION

In the course of designing a system there are a series of crucial decisions which must be made regarding the ultimate system implementation. Throughout the software course we are concerned primarily with the implementation of the software portions of these systems and how they interact with available hardware. To be sure, this area is vital to the designer. However, the thorniest problem initially confronting most designers of microprocessor based systems is how to partition the system functions between hardware and software implementations. This is understandable since most users are far more experienced with hardware design than software design. However, the plain truth is this: within the speed limits imposed by any computer, anything that can be done with hardware can be done with software. In fact, only a small percentage of applications will present speed problems. Usually even these applications are only speed sensitive in areas which can be readily identified and processed with discrete logic to make them adaptable to a software solution. We thus have a sliding scale of implementation possibilities from applications with no software (i.e. no microprocessor) to applications where 95% of the system cost will be in the software. Given this wide range of possibilities, how do we decide where to draw the line? Where indeed. Assuming that the objective is to do the job and make some money, the answer is obvious: we draw the line at the point where we find the lowest cost hardware/software system that does the job.

Before we can discuss how to trade hardware cost for software cost, we must first identify the areas that affect cost in both of these areas. For the purposes of discussion we shall consider cost to be localized in three areas: hardware cost, software cost, and system cost. After we have discussed the various cost areas we will be able to discuss tradeoffs required to modify system cost and performance.

The CPU chosen for the system will have the central effect on the hardware cost of the system. This is not because of the cost of the processor itself. For most systems the actual CPU cost will be an insignificant portion of the total system cost. It is a result of the effect of the CPU on all other aspects of the system design, both hardware and software. It therefore makes the most sense to discuss these costs within the context of the CPU itself.

6.1 Hardware Cost

Hardware cost will be considered to be all of the hardware which must be designed to implement the required system functions. This would include the microprocessor, memories, interfaces, clocks, power supplies, terminals, printers, or other pre-configured peripherals.

6.1.1 System Speed

To paraphrase an old police traffic slogan, "Speed kills microcomputer projects". This is due to the sad fact that of all the great things microprocessors do, doing them fast isn't their best attribute. Most commonly available microprocessors have maximum cycle speeds in the 2MHz range. Execution of an instruction generally requires from 4 to 10 machine cycles. Moreover, doing anything useful will require several instructions. What all this means is that a microprocessor operates considerably slower than conventional sequential and combinational logic. As a rough rule of thumb, if your system requires the processor to do anything faster than 10μsec (100 kHz) you will need to be very careful in your design.

There are a limited number of high speed microprocessors available, but these are sets of devices, not single package microprocessors. They are somewhat harder to use and considerably more expensive. If you begin to use these you may discover your cost rapidly exceeding the cost of some other form of logic implementation. Also, high speed for the CPU generally requires high speed memories, interface logic, and peripheral devices, further raising costs.

As we mentioned earlier, few projects have overall speed requirements that are so severe as to preclude microprocessor use. However, they do exist, and if you think you have one, be very careful to be certain from the start that a microprocessor will be able to do the job. Conversely, there is no point in paying for system speed you don't need. Speed is expensive. You generally get a certain level of speed with the microprocessor. If you're not using it, see if you can trade it for some interface simplicity. No use buying a fast processor and fast interfaces if a fast processor and some slower, dumber, and cheaper interfaces will do. We'll talk more about this later.

6.1.2 Memory Requirements

The system memory is where you will store the programs and data required for system operation. With most microcomputer systems this memory will consist of a combination of read/write memory (RAM) and read only memory (ROM). (With some processors the CPU itself contains a small read/write memory, thus making it possible to implement simple systems with just the CPU and ROM's. Larger systems will require additional read/write memory.) The object of the game here is, as usual, to minimize cost. This is done by getting as much of the software into ROM as possible. This is because ROM can be left with power off and the program will still be there when power is restored. Alas, such is not the case for RAM. Thus when you hear people say that programs should be in ROM because ROM is cheaper then RAM, it isn't really true. Bit for bit the costs are becoming quite comparable, with many types of ROM considerably more expensive than RAM. The fact is that RAM is not practical in dedicated systems which must maintain the program without re-loading memory every time the power is turned ON.

Read/write memory can be broadly divided into static RAM and dynamic RAM. A static RAM will maintain its data as long as power is applied. A dynamic RAM must be "refreshed" periodically. This refresh operation is accomplished by pulsing some of the address lines (usually the most significant bits) periodically. To do this requires the addition of special circuitry to the system. In general, the integrated circuit constraints are such that a static memory requires more area on the semiconductor chip than a dynamic memory of similar size. Static memories also dissipate more power per bit. The largest RAM memories are, therefore, usually dynamic, at least initially. As the device technology improves these larger memories usually then become available in static form.

The cost of both static and dynamic memories has declined and will continue to decline. This cost is based on the absolute cost per bit for a given amount of storage. However, the device organization and not this absolute cost per bit is often more important in practical applications. In terms of cost per bit, a 4096 x 1 dynamic memory may be much cheaper than a 256 x 8 static memory. However, you will need eight of the dynamic memories to do any good. They will require refresh circuitry, and they will take up eight times more P.C. board space in production. If you only need a 128 byte buffer and some miscellaneous program storage, the bigger "cheaper" memory may cost far more. For cost effective design it is imperative that you avoid memory overkill. Design in what you need, allow some extra for unforeseen difficulties and reasonable future expansion and stop.

The advances in memory technology are impressive and they receive lots of publicity. But the fact remains that few systems for mass production will require vast amounts of RAM. Often minimum package count and ease of system interface will be far more important than sheer volume. Buy one development system with lots of RAM. Use it to develop lots of systems with only the RAM required to do the job.

With ROM's, the situation is considerably different. Read only storage is really only useful organized in multiples of the computer's basic data word. It doesn't make much sense to mask program two 1023 X 4 ROMs for use in an 1024 X 8 system. As a result, ROMs are widely available for eight-bit processors in sizes from 8 x 8 to 2048 x 8. ROMs are available in three types, each suitable for certain areas of application: EROMs, PROMs and masked programmed ROMs.

An EROM is a ROM which can be erased and re-used. An EROM can be programmed and, if errors are found, erased and reprogrammed. Erasure is accomplished by exposing the EROM to intense ultraviolet light for a half hour or so. In this way the EROMs can be re-used indefinitely. EROMs are the most expensive type of ROM. They are best used in development work or low volume production equipment which require frequent changes to the operating program.

A PROM is a ROM which comes from the manufacturer with all locations as one's or zero's. It can then be programmed by the user. Unlike an EROM, however, once programmed a PROM cannot be erased. PROMs are somewhat lower in cost than EROMs. However, frequent program changes can quickly make them more expensive. They are best used in production systems which will require few changes but whose production volume does not justify a mask programmed ROM.

A mask programmed ROM is fabricated by the manufacturer to contain the desired program. It is neither field programmable nor erasable. A ROM is ordered by sending the semiconductor manufacturer your program. They then generate a custom ROM from your specification. The cost of ROMs produced this way is the lowest available. However, the semiconductor manufacturers charge a flat fee for the generation of the required mask. This cost makes mask programmed ROMs cost effective only for those high volume products whose program will never (hopefully) require change.

6.1.3 I/O Requirements

It is rapidly becoming apparent that I/O is the soft underbelly of most microprocessor based systems. Interfacing the microprocessor to the rest of the system is always a requirement. The microprocessors currently available generally provide only enough interface capability to directly interface one normal TTL device. This means that all signals in and out of the microprocessor must be buffered. Further, control signals must be decoded, interrupts must be processed, data must be latched and held until the processor or peripheral is ready to accept it, and many other system requirements must be met. All this falls within the realm of I/O.

The fundamental element of microprocessor I/O operations is the I/O port. An I/O port is the point where the signals to and from the various I/O devices meet their respective signals from the microprocessor. I/O ports provide both buffering and some control decoding. The I/O addresses sent out by the CPU are decoded to provide an enable signal to a specific I/O port, thereby gating the information from that port onto the system data bus for a read operation or gating the information on the system data bus into the port for a write operation. The mechanics of how the port works are not as important as the realization that all data into and out of the microprocessor is going to have to pass through I/O ports. This means that you will want to get your money's worth out of every port. To help you do this, some processors provide a small number (usually two or four) of I/O ports right on the CPU chip itself. If you only need one or two ports for a simple system, this can be a significant cost saving factor.

After you've got the I/O ports, you then must design the special logic rrquired to control the devices or circuits you are interfacinq. For most microprocessor applications this is where you will do the majority of your hardware design. If you do lots of microprocessor systems, you will eventually arrive at some standard I/O port design, but there will almost always be some detailed interface design work to be done.

When making the decisions about how to implement your I/O ports and control logic, you may be able to obtain some cost advantage by using a specialized interface device. Some microprocessor manufacturers have designed special families of devices to ease I/O design. These devices usually consist of several I/O ports, some defined logic functions, and all required control logic required to interface some device directly to the microprocessor with little or no external logic. For example, the data ports, control logic, and interface circuitry required to input and output parallel data directly to a serial interface is one popular example. Others include interrupt handlers, real time clocks, bi-directional data ports, and so on, with more becoming available as the industry defines what functions are commonly useful. If you can find someof these to fit your needs, they can save you money.

I/O design is the area where you can often achieve significant savings by trading hardware for software. It is also the area where you may be able to trade some cost for enough added speed to make a usable system. I/O design is an area where creative use of software and hardware will result in optimum system performance at lowest system cost.

6.1.4 Peripheral Devices

In terms of production cost the most expensive portions of your system can easily turn out to be those assemblies you have to buy pre-assembled. All types of computer keyboards, displays, printers, tape equipment, A/D and D/A converters, and similar peripherals are very expensive relative to the cost of the microprocessor hardware. In the normal microprocessor system these devices account for over 50% of the hardware cost. If you must include these components in your system, it is very important to make a very careful analysis of whether or not your product is still cost effective. It can be devastating to have to add a $75 keyboard to a microprocessor system where the total component cost is only $50. In this type of situation you might see if you can use a less expensive device and add the other features with software. All these types of decisions must be weighed carefully before you start the actual design.

6.1.5 Device Support

Into this category we toss all those microprocessor system details that drive your system cost up. These are particularly obnoxious because they are often overlooked until it is too late. The three most common offenders in this category are clocks, power supplies and interface requirements.

The system clock is used to provide the timing signals required to run the CPU and some of the other system logic. From a cost standpoint, there are two areas of interest: who generates the clock and how good does it have to be. In the first case the answer is either the CPU or the system. If the CPU generates its own clock (it may need an external resistor and capacitor), you don't have to worry about the second question. If you have to generate the clock, you definitely have to worry about it. Some microprocessors are very finicky about their clocks. This means special driver chips, crystals, logic, power supplies (i.e. money). If you are in a very cost sensitive operation, this can make a significant difference.

In addition to the main CPU clock, certain interfaces will require their own clocks. This includes serial interfaces, real time clocks, and many special interfaces. In some cases you may be able to derive the required clock(s) from the main system clock. If not, you will have to plan on the added cost of the required additional clock(s).

Power supplies are another area where requirements differ widely from microprocessor to microprocessor. Some microprocessors will run off the same +5V power supply that is used for all the logic. Some require up to three different power supplies. Power supplies are not cheap and you can quickly add a large cost to the system that you may be able to avoid entirely by chosing a different processor. (Note: after you go to the trouble of picking a microprocessor, be sure the rest of the system runs on the same voltages. It doesn't make much sense to cut corners to get a single supply microprocessor and discover the memories chosen need three supplies anyway.)

Besides paying attention to the number of system power supplies, you must be aware of the overall system current requirements. These requirements can vary widely, depending upon the CPU, memories, and interface logic used. You must be certain that your power supplies can supply enough current to meet peak system usage. Conversely, you don't want to pay for more capability than you need. To solve this problem, you usually don't settle on the final production power supply ratings until the system is complete and its power requirements are characterized. This is in contrast to the selection of the system hardware, where the number of supplies to be used in the system is determined before beginning the design.

Interface requirements relate to support circuitry required to use the microprocessor with other devices in the system. A microprocessor that is very easy to use among the members of its own family of devices may turn out to be a horror to interface to the rest of the world. This is particularly true of P-channel devices to be used in N-channel or TTL systems. Incompatibilities among system components can lead to problems and increased costs all over the system, including the previously mentioned clock and power supply areas.

6.1.6 Microprocessor Hardware Selection Summary

It should be obvious from the preceding brief discussion that picking microprocessor hardware is a tricky business. Even ignoring the software criteria, you must be very certain you get a device which will meet your system requirements at the lowest cost. It is important to remember at this point that lowest system cost may not always be the same as lowest possible hardware cost. Modification ease, maintenance and other factors may enter into the picture. There are times when you may want to knowingly allow some extra hardware cost to lower the costs in some other area. We will point out these areas as we go along.

6.2 Software Costs

Software costs are insidious. You can't see it, or feel it, or hear it, but software can break your microcomputer project faster than almost anything. As hardware systems and peripheral devices become more and more standard, more and more of the design-to-price burden is going to fall on the designer who has to design the software to hold these hardware blocks together.

Software is characterized by a very high development cost and a very low duplication cost. By way of example, IBM's software development of OS 360 (a very large and complex software project, to put it mildly) is estimated to have taken over 5000 man years of development time. However, the entire system can easily be duplicated and stored on $1000 worth of magnetic tape. As we said, duplication is cheap, development is expensive. This characteristic brings with it the following generalization: software for use in high volume products must be fixed. It is absolutely not possible to produce low cost custom software. Once you commit a program to ROM, don't consider changing the program unless you are prepared to change every other identical ROM in every other system. (Not to mention updating all reference documentation.) The cost of custom software (unless you are in that business) is so high as to completely preclude it from volume systems. The software development cost very quickly completely overshadows the hardware cost.

Software exerts cost pressure on projects in two basic ways. The first is when poor technique and analysis lead to systems with inefficient use of expensive hardware resources. This causes the system to end up with more memory than it really needed, high speed interfaces that could have been eliminated with good software, extra I/O ports that some software multiplexing could have eliminated, and so on. The second way software raises cost is in the development/support cycle. This results in late projects due to inadequate time requirement forecasting, program bugs that turn up just after you take delivery on 10,000 mask programmed ROMs, documentation that requires a complete software system redesign when the program has to be changed a year after release, and other gory, expensive examples. Of the two areas software causes problems, the second is far more serious than the first. The first set of problems will naturally become less severe as you become more familiar with hardware/software system designs. (After all, that's what this course is here to teach you.) The second set belong to that group of problems that the entire computer community suffers along with year after year. Some progress is being made, but it is still a thorny problem. Good engineering practice is your best defense. Remember this basic rule: hardware and software design are equally complicated. The only difference is the rules.

Let's look at those areas where software can raise (or lower) your hardware costs. Remember we are considering a sliding cost scale from all hardware to virtually all software.

6.2.1 Processor Organization

The architecture of the processor you choose for your system can have a significant effect on your software costs. This is felt primarily in two areas: memory and I/O. A processor which is deficient in memory addressing modes will require larger programs to accomplish the same job as a processor with more flexible addressing. More program means more memories, and more memories means more cost. A lack of on-chip registers may require you to use memory for temporary data storage. These memory references take more time during program execution and may make the difference between a simple (i.e. cheap) interface and a more complex (i.e. expensive) one. A versatile interrupt system may enable you to do most of the interrupt decoding with logic built into the CPU. Otherwise, you will have to add more service routines, I/O devices and money. A processor with a versatile instruction set may enable you to implement your programs much more efficiently, thereby saving memory space. The list goes on and on. Any area of the microprocessor's architecture can become a cost sensitive point in certain applications. The ultimate goal is to find the cost sensitive areas in your application and pick a processor that is strong in those areas.

6.2.2 Program Structure

The program structure, just as with the processor architecture, exerts its primary effect on the system memory requirements and I/O structure. Poorly designed programs will often take twice the memory of more carefully designed programs. You must balance the time and cost required to optimize programs against the cost of memory saved. Ideally, you will become skilled enough to design near optimum code the first time, thereby avoiding the expensive refinement procedure. Also, different program structures can be used to get maximum speed of program execution in speed sensitive areas. Failure to take advantage of these structures can result in the use of more expensive I/O interface hardware than is actually needed. The different program structures and their tradeoffs in speed and memory usage are discussed throughout the software lessons.

6.2.3 Implementation Language

The level at which you develop your programs has its primary effect on system memory size and overall system speed. Programs developed in higher level languages will generally be faster to develop, but they will take more time to execute and occupy from two to ten times more memory than the same program done in assembly language. Assembly language programs can be designed for optimum memory usage and system speed but they take more time to develop. A data processing industry estimate is that assembly language programs take from two to five times longer to develop than comparable higher level language programs. This is particularly true of large, complex systems. You must balance the cost of development against the cost of the additional hardware resources. As a general rule, higher level languages will be lower in cost for small quantities of systems with assembly language becoming more cost effective as production quantity increases. (This assumes the higher level language programs can meet all system speed requirements without extra work.) The higher level language/ assembly language tradeoffs are discussed in Lessons 9 and 10.

6.3 Systems Cost

Beyond the costs associated with producing the hardware are those costs associated with developing and maintaining the product. Unlike production costs, which are incurred as a function of how many units are produced, these costs are largely independent of production. Indeed, it is possible to incur very large costs in this area and never produce a single unit.

6.3.1 Development Costs

System Development Costs include all of the expenses you incur during the design of the product. Since these costs will be incurred prior to production, they will usually have to be met from your available resources. The areas of cost in this phase are all well known. However, the addition of software development adds a few extra categories.

Hardware Selection

All time and money spent evaluating various microprocessors and system components prior to commencing the actual system design. This would also include all analysis of crucial timing and interfaces and the initial partitioning of the system into hardware and software blocks.

Hardware Design

All time and money spent designing and debugging the hardware required to implement the system hardware.

Software Design

All time and money spent designing and debugging the programs required for use in the system. This may include a significant amount of expense for timesharing computer usage if you do not have the required program translation facilities available in house.

Development Tools

This includes any special hardware (such as a microcomputer development system or special test hardware) you have to buy for debugging and checking out the system design. Some of this cost will actually be spread out over all developments which end up based on the same microprocessor.

Documentation

All cost spent in developing the user manuals, production documents, reference specifications, and other documents essential to converting a working lab project into a viable product. This cost should not be underestimated. Thorough documentation will probably consume 20-25 percent of your development budget. However, it will be money well spent as your product matures and requires changes.

Marketing

This is the cost incurred in taking your finished product from the lab and presenting it to the world. This is not usually an engineering activity.

6.3.2 Modification Costs

Once you have a working product, there is always the possibility that you will want to issue a new, improved version. This is one area where a microprocessor based system can really save you time and money. In a total hardware system, a design extension or re-design will usually mean an almost total re-investment of the initial development costs. However, with a microprocessor based system you may be able to make substantial functional changes with little or no changes to the hardware. This is because a software system can be re-configured by changing the program. Bearing in mind that all the software cost rules still apply, this is still usually a very effective technique. Expanding or changing an existing system is one area where you will find that the money spent on documentation was well spent. It can often make the difference between a successful and cost effective design modification or a complete re-design.

Program changes will often not be effective in products which were optimized so completely initially that there is not much extra hardware left to work with. The program can, after all, only perform functions which use available hardware. No matter how clever your programmer, if there isn't enough memory or I/O ports, some things just won't be feasible. If you have a product which looks like it is a candidate for later expansion, you may wish to incur a little higher production cost initially by adding some hardware for later use.

6.3.3 Maintenance Costs

Any cost you incur when your product fails in the field comes under this heading. All those field servicemen, return clerks, rework lines, and other support are expensive. Here too, the microprocessor can save you money. Almost by definition, the microprocessor must communicate with the entire system. This means that with the addition of some programming, memory, and some small amounts of hardware you can convert your microprocessor based system into its own diagnostic tester. You may not need to provide thorough tests, but even some simple tests can make troubleshooting a lot easier. Anything you can do to make testing and servicing easier will lower your maintenance costs.

Naturally, you must weigh the benefits of self-testing against the cost it will add. Often, however, you will discover at the end of the project that you have some extra I/O lines or a partially full ROM. Since these are going to be there anyhow, you may as well use them if you can. Since this type of thing is not usually discovered until well into the project, the addition of self test features at that point is one of the few times when it may be desirable to add features after the design has started. However, if you want to be sure you have self-testing you should never wait to see what is left over. In that case, the self-testing features should be designed in like any other system feature.

6.4 A Perspective On Costs

Now that we have examined the various component costs, let's see how they relate to the total cost per unit of our proposed product. Over the total life of a product, the cost can be represented by the following general equation:

$$TC = \frac{FC}{N} + VC$$

where TC is the total cost per unit,
FC is the fixed cost required to develop and maintain the product,
VC is the variable cost associated with producing each unit, and
N is the number of units.

The terms in this equation can now be further broken down into those cost areas we discussed in the previous sections. Thus the fixed cost portions of the equation would turn out to be the development costs of the hardware and software, the documentation, the modification costs to the line of products, marketing, and all other cost which is incurred regardless of the volume of product produced. These costs are amortized over the number of units produced; the larger the number of units produced, the lower the fixed cost per unit.

The variable costs would be the cost of all the hardware components, production labor, field service for the percentage of units which prove defective, and all those other costs which vary based upon the number of units produced.

It is clear from this equation that the area where we will want to direct our cost reduction effort is dependent upon the quantity of units produced. For small quantities of units, we will want to minimize the fixed costs.

In practical terms this means using higher level languages (when available), hardware that is designed for ease of debugging and high reliability, and a general emphasis on development speed rather than low cost production. Conversely, for high volume production we will want to absolutely minimize production costs. This means highly optimized programs to minimize memory use, maximum use of program controlled interfaces to eliminate unneeded hardware, mechanical designs for easy production and any other techniques which can be used to hold the cost down.

The exact point at which the emphasis shifts from fixed cost reduction to variable cost reduction naturally changes for every product. In general, the more expensive the final product, the lower the emphasis on the variable costs.

6.5 Trading Off Software and Hardware

Now that we have discussed the main factors affecting system performance and cost, we can discuss the areas where system problems will force us tc trade off hardware and software to modify system performance and cost. As we mentioned earlier, high speed (programmed, hardware, or whatever), large numbers of parts, and complex software are all expensive. We will be trying to implement all required system functions using the minimum cost combination of these items.

6.5.1 Conditions Which Lead to Design Trade Offs

In the course of the design we will be faced with several possible project conditions, some of which will require us to consider the various possible system tradeoffs. These conditions can be summarized as follows:

1. system speed too low, system cost too high,
2. system speed too low, system cost acceptable,
3. system speed acceptable, system cost too high,
4. system speed acceptable, system cost acceptable,
5. system speed excessive, system cost too high,
6. system speed excessive, system cost acceptable.

Clearly, each of these conditions requires different remedial action. Condition one is an obvious crisis situation. Unless some major breakthrough can be discovered, the project is probably doomed. Condition two is also fairly critical. It can be worked on only if the necessary speed can be acquired without driving cost into the unacceptable range. Very careful analysis will be required. Conditions three and five are probably both solvable by application of some hardware/software trade offs. Conditions four and six can be left alone. They may also be examined to see if extra features might be added to utilize the excess system speed without increasing the cost to an unacceptable level. If you elect to try this, be very careful not to go overboard. Any additions are best made in very small controlled increments. Avoid "creeping features" (see Lesson 2). If you aren't sure what to add, don't. Be happy you brought this one in under budget and save your money for next time.

After you figure out which condition your project is in, you have three alternatives: built it, change it, or cancel it. Building it or canceling it are decisions that you have to make on a situation by situation basis. Changing it may help you postpone that decision for awhile, but ultimately you will still have to decide. We can now examine how to change it so that hopefully you can decide to build it.

6.5.2 System Speed Problems

As we have emphasized all along, speed usually costs money. There are very few situations where increasing system speed lowers the cost. If you have a project which has to have increased speed, you might consider the title of this section to be "Trade Offs that Increase Cost". With that in mind, we can examine where to look to increase system speed.

System speed problems can be broadly divided into data transfer rate problems and data manipulation rate problems. In system operation these two types of problems will require distinctly different solutions. However, the same general techniques will apply to correcting both.

6.5.2.1 Data Transfer Rate Problems

Data transfer problems are encountered when transferring data between the computer and system I/O devices. This class of speed problem can be further subdivided into processor rate limited problems and peripheral rate limited problems. Processor rate limited problems arise when the computer is transferring data to a device which must have a high, non-varying transfer rate. This is characteristic of many real time interfaces, disk drives, and high speed buffered I/O devices. In the case of the disk drive, for example, it is not practical for the computer to vary the speed of disk rotation. Therefore, the processor must be able to read the data as fast as the rotating disk presents it to the read head. Data transfer rate problems of this type will result in lost or erroneous data. They represent the most serious system speed problems and they must be detected and corrected before the system will function properly.

Curing processor rate limited problems where the speed differential is excessive requires the addition of hardware to transfer some of the speed burden from the CPU. If the speed differential is close, restructuring the program sections which perform the actual data transfers may provide the speed margins you need. However, since instructions execute in fixed multiples of system cycle times, it will be impossible to adjust the system speed any more accurately than the execution time of the fastest instructions. For this type of problem, adjusting system speed by varying the program structure will only be effective over a fairly narrow range of timing.

Unlike processor rate problems, peripheral rate limited problems turn up when the computer is able to process the data at a much higher rate than the I/O devices can supply or accept it. This problem is most commonly encountered when the microcomputer is communicating with peripherals which are mechanical or which require user interaction, i.e. printers, tape readers, teletypewriters, etc. For example, many small microcomputer systems rely on the Teletype Corporation's model ASR 33 teletype as the main system peripheral. It serves as the keyboard, display, punch and reader for all program I/O operations. Now the teletype can only transfer data at the rate of ten characters per second, or one data byte every 100 milliseconds. Printing 2500 characters (a small program listing) will take over four minutes. In this case, the computer will be spending most of its time waiting for the teletype to finish printing.

Peripheral rate problems are probably the most commonly encountered system speed problems. Fortunately, they seldom present a critical design problem. The cure is usually to add a faster I/O device. Even this solution has limitations. Most computer peripherals involve mechanical devices, and these will almost always be slower than the computer. You must trade off the cost of the faster peripheral against the time saved. If you discover you have a system which spends most of its time waiting for I/O transfers (a condition referred to as I/O bound), you may want to see if you can come up with some features to utilize what is essentially free processor time. Even better, you may be able to use some of that time to replace some hardware and further lower system cost. On the other hand, if the system can do everything it needs to at a cost you can afford, who cares if it spends 95% of its time waiting for the user to press a key? Microprocessor hardware is going to become so inexpensive that it will probably become far more economical to underutilize several microprocessors than to spend the development cost to optimize the use of one.

6.5.2.2 Data Manipulation Rate Problems

Where data transfer rate problems were related to how fast we can get data in and out of the computer, the data manipulation rate problems are concerned with how fast the data is processed once the computer has it. Where data transfer rate problems will be solved mainly be adding or changing system hardware, data manipulation rate problems will be solved mainly by restructuring the system's software.

The typical data manipulation problem arises when some section (or sections) of the system program takes an excessive amount of time to execute. The more commonly used that portion of the program, the worse the problem. This type of problem is characterized by your pushing a button and waiting for fifteen seconds until the teletypewriter prints the ten digit answer to your equation. Using some hand held scientific calculators for complex calculations (try SIN 89^o) provides some excellent examples of data manipulation rate limitations.

Some problems of this type are unavoidable in microprocessor systems. Their low speed (relative to minicomputers and large computers), modest instruction sets, and small data element size limit the efficiency with which any program will run. They are simply not designed for complex data processing applications. No matter how good the algorithm, certain classes of operations are going to take up significant amounts of computing time. Some examples of this group are complex mathematics routines (anything more complicated than a sixteen-bit integer divide can safely be considered complex), large memory searches, array operations, and moving blocks of data around in memory. In the large and minicomputer world, another primary cause of this problem is multiple user systems. Fortunately, to date the microprocessor world has been spared this particular problem. If your system requires any of these types of operations, you will end up paying some speed penalty. You will be able to minimize it to some extent, but it will be there. Fortunately, the types of applications which will use microprocessors do not normally require large numbers of complex operations. If you have one that does, you might seriously consider one of the sixteen-bit microprocessors or a low end minicomputer.

6.5.3 System Cost Problems

System cost problems become significant when you have a working system which must be made more economical for practical production. The term "problems" in this context is probably misleading. Virtually all systems intended for high volume production will go through some cost optimization procedure between prototype and final production. Usually you will have decided that the cost range for the product is acceptable before proceeding with the development. This decision is based on market studies, comparison with existing products, and other evaluations of what is a reasonable final selling price of the product. This number can then be projected back to arrive at a cost range for the product.

In general, the techniques for lowering product cost will be the reverse of techniques to increase speed. You will want to remove extraneous hardware, compact all programs into minimum memory space, and in general, make the maximum use of the processor and software to implement system functions. This must all be done without creating any system speed problems. Therefore, the proceedure is best carried out in discrete steps. You refine one section of the system, make sure the system still works, and move on to the next section. Ultimately you will reach a point where no further cost economies can be achieved without compromising system performance.

Cost optimization should always be undertaken with the firm realization that the end must justify the means. It is an expensive process that is usually only vigorously applied to products whose high volume will justify the expense. Otherwise the cost of the optimization will overshadow any savings made in production.

6.6 Hardware Speed Trade Offs

When you must modify system speed using hardware, you will be trying to either increase or decrease the amount of work done by the processor. In the first case you will be trying to simplify the system hardware or replace much of it with software. This results in decreased hardware cost and lower system speed. In the second case you will be trying to transfer some of the work being performed by the software out to the hardware. This will result in higher system cost. Within this framework let's examine some of the alternatives available.

6.6.1 Processors and Memories

A simple solution to some system speed problems may be to change processors within the same family. Some manufacturers provide microprocessors which are graded by speed. If the nominal processor speed is 2 MHz, some devices may be available in selected speed ranges from 1 to 4 MHz. Since the processor cost goes up with the speed, using this method you only have to pay for the speed you require.

If you are considering a faster (or slower) processor, you must also consider the effect that memory speed has on program execution. The computer must get all instructions and data from memory. If the memory is not at least as fast as the processor, there is no point in increasing processor speed. Similarly, you may be able to increase system speed by using the same processor with faster memories.

6.6.2 Decode Logic

Decode logic is required for a variety of purposes in a microcomputer system. Most decoding is done to determine I/O device addresses and memory addresses. This logic is almost all done with hardware, and it can usually be minimized in a dedicated system. For example, many microprocessors can address 65K bytes of memory using 16 address lines. Very few applications will require this much memory, so after you determine how much memory the system requires, you can eliminate the excess decoding. For example, if you only need 4096 bytes of memory, you need only decode 12 address lines to access all valid memory addresses in your system. Similar minimization can be applied to the I/O device addresses.

One added benefit of reducing the decoding is that the undecoded lines can be used as extra control lines in the system. Usually the full address bus runs everywhere in the system. If system speed permits, the undecoded address lines may be used to eliminate further hardware control logic. In the case of the system with 4096 bytes of memory we mentioned earlier, the four unused address lines could be used individually (or even decoded) to provide system control signals. Similar trade offs can be performed in systems which require fewer I/O devices than the maximum available.

6.6.3 Memory Buffers

Memory buffers are used to collect or hold data that is in transit between the CPU and system peripheral devices. The addition of a high speed buffer dedicated to a specific peripheral can be used to solve processor data transfer rate problems. This is particularly effective if the peripheral has a low average data rate with high speed burst transfers of data. A buffer can be used to collect the data during the burst transmission, with the CPU reading the individual data elements from the buffer after the transmission is complete. This type of buffering can also be used in conjunction with the computer's DMA facility. In this case, the buffer accumulates the data and transfers it into the main computer memory in a single block transfer.

Buffers can also be used to solve peripheral data rate problems. In this case, the CPU transfers the data out to the peripheral buffer. The peripheral can then take the characters at its own rate with no further processor intervention.

Addition of buffers to the system requires the addition of considerable hardware expense. Accordingly, they should only be added if the system really needs them. As long as speed is not a problem, most microprocessors can do a good job of implementing buffers. They can do this using already present main memory and some programming. Data is transferred into and out of this type of buffer using an interrupt. The device interrupts when it is ready for a transfer and the CPU performs a single transfer. When the buffer becomes full or empty, the data is then processed, just as with a dedicated buffer. This is always much cheaper than an external buffer system. In the course of the design, if you think you need data buffering, look very carefully to see if it can be done using software. Even after the design is done you may discover that a hardware buffer initially thought necessary can actually be done in this way. It may be worth the redesign cost to save the hardware cost, particularly if production volume will be high.

6.6.4 Specialized Interface Devices

A specialized interface device is designed to perform some defined function in the system. Usually the function to be performed could be performed using either software or the specialized device. You will consider a trade off when you either find yourself with a speed problem and no interface device or the interface device and lots of program time available. In the first case you design in the device to free up the program time that performing the function is tying up. In the second case you take out the device and replace the function with software.

A common example of this type of device is the UART (Universal Asynchronous Receiver Transmitter). This device accepts parallel data and converts it to a serial bit stream conforming to the EIA RS232C data transmission standard. The function can easily be performed under program control, but as mentioned earlier, each character sent or received will take up 100 milliseconds of computer time. During this time the software must convert a character from parallel to serial, add start and stop bits and generate all timing and control signals required to perform the transfer. If your system has the time, fine. If it doesn't, you add a UART. The only time required now is the time required to write one parallel byte out to the UART. After that, the UART generates all those functions that were done by the software, freeing your processor to do other things. Similar trade offs can be made using other pre-defined functional devices.

6.6.5 Interrupts

In many systems the computer must spend considerable time responding to interrupts. If there is more than one possible interrupting device, the processor must determine which device generated the interrupt before it can process any data. This identification can be done in a combination of hardware and software that can be varied to meet system speed/cost requirements.

For maximum system speed you design the hardware so that each interrupting device responds to CPU acknowledgement with the address of its own dedicated service routine. This gives maximum response speed, since no time is spent decoding any device identification codes. In some processors this can be reduced to the interrupting device providing an actual subroutine call instruction, making the interrupt almost transparent in terms of overhead time loss.

To lower hardware expense, the device identification can be moved into the service routines. In this case, the interrupting devices all provide the same routine address. The software must then poll all devices in the system to see who generated the interrupt. This adds a significant amount of overhead time to the routine, and will probably not be satisfactory for faster devices.

As a compromise, the system can be implemented as a combination of direct and indirect interrupt decoding. In this case, you assign your highest priority or fastest (usually the same) devices their own identification address. They will then interrupt directly to their routines with minimum time loss. The lower priority devices can then be assigned to a common address and these can be decoded under slower, cheaper software control.

6.7 Software Trade Offs

Software trade offs are made for the same reason as hardware trade offs, namely modifications of system cost and speed. Where we traded off hardware for different hardware or a combination of less hardware and some software, with software we will usually be trading off program speed for memory size. Increases in program speed will often take more memory, thereby costing more money. Conversely, if speed is not a problem, certain program types can be replaced by markedly less code, with a subsequent lowering of memory size and cost. It must be kept in mind, however, that not all decreases in program size lower memory cost nor do all increases in program size increase cost. The only time changes in program size affect memory cost at all is when the change results in the saving or use of an entire memory. For example, if your program is to be located in 2K x 8 mask programmed ROMs, the only time that your cost will change is when your program size exceeds multiples of 2048 bytes. Up to that point, the memory is essentially free. Similarly, if you discover your new, improved, program is now 2075 bytes long, you may want to expend some time eliminating those 27 extra bytes. (The terms and techniques discussed in the next few sections are covered in greater detail in the software lessons.)

6.7.1 Program Loops and Subroutines

Program loops and subroutines are used to minimize program size and control execution. A sequence of operations which is to be executed a fixed number of times can be placed in a loop. A section of code common to several portions of the program can be placed in a subroutine. The actual coding is thereby only written one time no matter how many times the loop is executed or the subroutine is called. Loops and subroutines minimize program size at the expense of some program speed.

The instructions which must be executed to control execution of the loop or the calling of the subroutine take a certain amount of time that is not required for the actual function being performed. In speed critical situations the effect of these overhead instructions can be eliminated or modified to increase execution speed. This is done by replacing the loop or subroutine with the actual straight line code that was originally there. This eliminates the overhead instructions completely. Alternatively, a loop may be modified to use a lower percentage of its time for overhead. This is done by partially replacing the loop with the straight line code and lowering the number of times through the loop. For example, say a certain function is to be performed 10 times, once for each execution of the loop. In this case, say loop overhead is 20%. By duplicating the function and lowering the loop count to five we would do the same processing with only 10% overhead. The price would be a doubling of the amount of memory occupied by the function.

6.7.2 Functional Computations

Throughout your program you will use functional computations to evaluate data and decide on program responses to input conditions. You will be able to vary the execution speed and memory usage of many of these blocks based on how you evaluate the data. For example, let's say we have an application where we need to multiply two eight-bit integers. One solution is to write an algorithm which will multiply the two numbers. If for some reason the speed of the algorithm execution was not adequate for our application, we might consider storing all possible results in a ROM (or part of a ROM). We would then use our two numbers to compute the address of the product, thus removing most of the computations. This method should execute considerably faster. Again, the price is more memory usage.

In practice, not many mathematical functions can be produced in the manner just described. However, the technique is very often applicable to memory addresses. A required address can often be computed as part of the program execution or stored as fixed data. Computation by algorithm is more efficient, but fetching defined data is faster. These types of alternatives can be traded off throughout the course of system software design.

6.7.3 Repeated Computations

Related to functional computations is the class of program operations called repeated computations. Analysis of programs over the years has shown that in most programs 90% of the execution time is spent executing 10% of the program. These software "critical paths" are what we call repeated computations. If your system has a speed problem, the first thing to do is to see if you have any repeated computations. You can then devote your optimization effort in those areas where it will do the most good. Some common types of repeated computations are common mathematical functions, table searches, data movement routines, and data sorts.

If you find you have a clearly defined repeated computation, you may find it worthwhile to study it. See if you can find a better algorithm in the data processing literature. If you can't find one, do your best to devise one. Time spent thoroughly optimizing a repeated calculation can be far more valuable than partially optimizing several sections of less frequently executed code.

6.8 Summary

The hardware/software design procedure is something that you only learn by practice. You must gain first hand experience in the real world. It is a process which becomes more than designing the hardware and then designing the software. It is an integrated proceedure which will allow you to implement some of the most creative digital systems ever imagined. We have only scratched the surface of what is available, and what is available is just the beginning.

REPRESENTING BINARY DATA

REPRESENTING BINARY DATA

When working with digital computers it is necessary to work with binary data. Computer components are built up from electronic devices which can only represent data as 0's and 1's. This means you will have to use binary to represent numbers. In spite of this, it is impossible to escape from the fact that binary data is not overly convenient to use. We have all used base 10 numbers for years and the base 2 number system seems quite inefficient by comparison. It takes 6 binary digits to represent the number 50_{10} (110010_2), and it gets worse. In this section we will discuss how the individual binary data elements are represented and how they can be grouped together for more convenient use. Binary arithmatic and logic are discussed in the following supplementary section.

7.1 - Binary Data Elements

A computer data element of arbitrary length N is shown below.

N	.	. .	6	5	4	3	2	1	0

The right most bit (bit 0) is considered to be the least significant bit. Bit N, at the left most position, is considered to be the most significant bit. Thus a computer with a 16-bit data element would have data in bit positions 0-15:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

16-Bit Data Word

Similarly, an 8-bit microcomputer would have data in bit positions 0-7:

7	6	5	4	3	2	1	0

8-Bit Data Word

Thus when we speak of loading one's into bits five and seven of an eight-bit register, we will be loading the following pattern.

7	6	5	4	3	2	1	0	Bit Position
1	0	1	0	0	0	0	0	

References to the bit positions of a register or memory location rather than to the binary number in a register or memory location are common in control and logic applications.

7.2 - Binary Numbers

All number systems (including binary) represent numbers as a function of the radix (number base) and the position of the individual digits. Any number composed of digits A_N-A_0 in a radix R can be represented as follows:

$$A_N A_{N-1} \;.\;.\;.\; A_1 A_0 = A_N R^N + A_{N-1} x R^{N-1} + \;.\;.\;.\;.\; + A_1 x R^1 + A_0 x R^0$$

where A is any digit in the range 0 to R-1 and N is the digit position. For example, consider the number 136 in base 10. We have digits in positions 0, 1 and 2. In this case, all values of A must be in the range 0-9, and R = 10. We thus have a number represented as

$$136_{10} = 1 \times R^2 + 3 \times R^1 + 6 \times R^0$$

(subscript to identify number base.)

$$= 1 \times (10)^2 + 3 \times (10) + 6 \times 1$$
$$= 100 + 30 + 6$$
$$= 136_{10}$$

Binary numbers can be similarly represented. The difference is that where in decimal we have ten possible numbers (0-9), in binary we only have two (0 and 1). This means that representing a given number in binary will require more digit positions than representing the same number in decimal. Thus the binary number 10110 is represented as

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= 1 \times 16 + 0 + 4 + 2 + 0$$
$$= 22_{10}$$

NUMBER SYSTEM CONVERSIONS

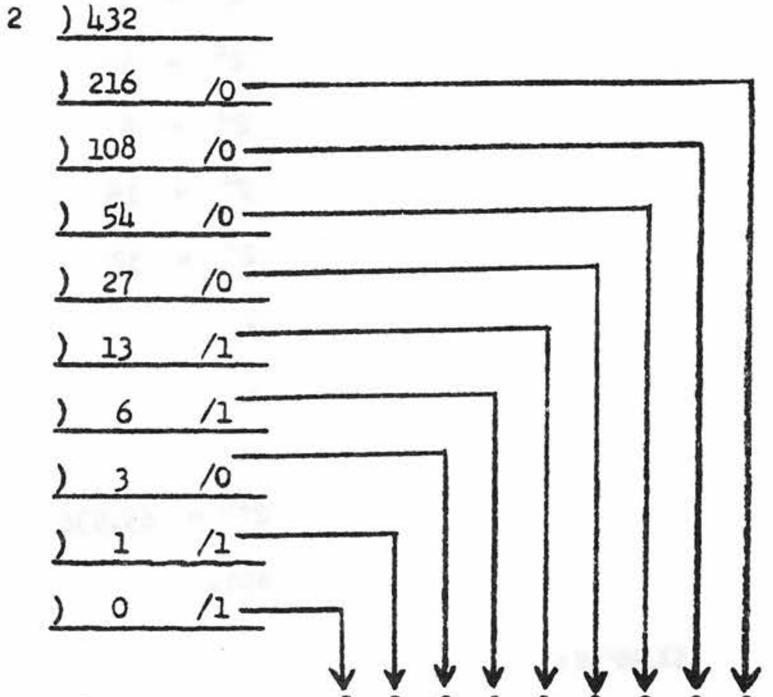
NUMBER SYSTEM CONVERSIONS

DECIMAL TO BINARY

To convert any decimal number to a binary number, take the decimal number and successively divide by "2" and write down the remainder (1 or 0) as you continue dividing until the number becomes "0".

EXAMPLE:

Convert 432_{10} to a binary number.



$$432_{10} \quad = \quad 1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0_2$$

BINARY TO DECIMAL

As in the decimal number system, the least significant digit is on the right and the most significant digit is on the left and each digit is a multiple of a certain power of 10.

$$432_{10} = 4 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$$

This is also true for a binary number, except that it is a multiple of a certain power of "2".

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

So to convert a binary number to a decimal number, take each power of "2" and change it to its respective decimal number.

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

.

.

.

$2^{16} = 65,536$

etc.

EXAMPLE:

Take the number 101101

$$101101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$

$$= 32 + 0 + 8 + 4 + 0 + 1$$

$$101101_2 = 45_{10}$$

CONVERTING DECIMAL TO OCTAL

Use the same method as to convert decimal to binary except divide by the base of "8" instead of "2".

EXAMPLE:

Convert 131_{10} to Octal

```
8   ) 131
    ) 16    /3
    ) 2     /0
    ) 0     /2
```

$131_{10} = 203_8$

CONVERTING OCTAL TO DECIMAL

To convert from octal to decimal use the same method as for binary to decimal conversion, except use the powers of "8" instead of "2".

EXAMPLE:

$8^0 = 1$

$8^1 = 8$

$8^2 = 64$

$8^3 = 512$

$8^4 = 4096$

$8^5 = 32768$

$8^6 = 262144$

etc.

EXAMPLE:

Convert 203_8 to decimal

$$\begin{aligned} 203_8 &= 2 \times 8^2 + 0 \times 8^1 + 3 \times 8^0 \\ &= 2 \times 64 + 0 \times 8 + 3 \times 1 \\ &= 128 + 0 + 3 \\ &= 131 \end{aligned}$$

CONVERTING DECIMAL TO HEXADECIMAL

Again, use the same method for decimal to binary conversion, except use the base "16" instead and replace the remainders of "10" to "15" by the letters A to F respectively.

10	=	A
11	=	B
12	=	C
13	=	D
14	=	E
15	=	F

EXAMPLE:

Convert 9192_{10} to hexadecimal

$$\begin{array}{lll} 16 \;\underline{)\;9192} & & \\ \;\;\;\;\;\underline{)\;574\;/\;8} & = & 8 \\ \;\;\;\;\;\underline{)\;35\;/\;14} & = & E \\ \;\;\;\;\;\underline{)\;2\;/\;3} & = & 3 \\ \;\;\;\;\;\underline{)\;0\;/\;2} & = & 2 \\ \;\;\;\;\;\;\;9192_{10} & = & 23E8_{16} \end{array}$$

CONVERTING HEXADECIMAL TO DECIMAL

Again, use the same method for binary to decimal conversion, except use the powers of 16 instead, and convert the letters A through F to 10 through 15 respectively.

$$16^0 = 1$$

$$16^1 = 16$$

$$16^2 = 256$$

$$16^3 = 4096$$

$$16^4 = 65536$$

EXAMPLE:

Convert $23E8_{16}$ to decimal

$$\begin{aligned}23E8_{16} &= 2 \times 16^3 + 3 \times 16^2 + E \times 16^1 + 8 \times 16^0\\ &= 2 \times 16^3 + 3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0\\ &= 2 \times 4096 + 3 \times 256 + 14 \times 16 + 8 \times 1\\ &= 8192 + 768 + 224 + 8\\ 23E8_{16} &= 9192_{10}\end{aligned}$$

CONVERTING OCTAL TO BINARY, HEXADECIMAL TO BINARY, OCTAL TO HEXADECIMAL; AND BACK

Convert to binary first, then if needed, regroup the binary numbers into the desired groups of three or four binary digits, (three for octal or four for hexadecimal). These translate directly to the desired number system. Always start the regrouping with the LSB.

Binary	Octal	Binary	Hexadecimal
000	0	0000	0
001	1	0001	1
010	2	0010	2
011	3	0011	3
100	4	0100	4
101	5	0101	5
110	6	0110	6
111	7	0111	7
		1000	8
		1001	9
		1010	A
		1011	B
		1100	C
		1101	D
		1110	E
		1111	F

EXAMPLE:

1) Convert $4A2BC_{16}$ to Octal

$4A2BC_{16}$ = 0100 1010 0010 1011 1100

= 01/00 1/010 /001/0 10/11 1/100

= 01 001 010 001 010 111 100

= 1 1 2 1 2 7 4

$4A2BC_{16}$ = 1121274_8

2) Convert 1435_8 to Hexadecimal

1435_8 = 001 100 011 101

= 001 1/00 01/1 101

= 0011 0001 1101

= 3 1 D

1435_8 = $31D_{16}$

BCD NUMBERS

BCD NUMBERS

In some applications it is desirable to be able to directly represent decimal numbers in the binary computer. This is done using Binary Coded Decimal, BCD. When using BCD we do not use all possible data values that the binary data element can represent. Instead, we limit ourselves to the following four bit patterns:

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The other six four bit combinations (1010-1111) are not used in BCD.

An eight-bit data element can hold two BCD digits. This means it can represent decimal numbers from 0-99. BCD is very commonly encountered in control and instrument interface applications. As a result, many computers provide instructions to allow direct arithmetic with BCD numbers.

BINARY FRACTIONS

Binary numbers are generally considered as whole integers (i.e., 1, 2, 3, ...). However, it often becomes necessary to represent numbers other than whole numbers. Binary fraction representation is analagous to decimal fraction representation. In decimal numbers a fraction consists of digits to the right of a decimal point; in binary, we consider the bits to the right of the binary point to be a fraction. In a binary fraction the bits represent 2^{-N}, where N = the bit position to the right of the binary point. The powers of 2^{-N} are shown in the number tables. Consider the following binary number:

Binary Point

1101.1101

This binary number representation means

$$2^3 + 2^2 + 0 + 2^0 + 2^{-1} + 2^{-2} + 0 + 2^{-3}$$

$$= 8 + 4 + 1 + .5 + .25 + .0625$$

$$= 13.8125$$

Numbers can be converted to and from binary fractions using the techniques already shown for converting whole binary numbers. Unfortunately, not all fractions are as well behaved as the above example. Consider the decimal number 3 1/3. When we try to convert it we end up with -

$$3\ 1/3_{10} = 3.3333......3_{10}$$

$$3\ 1/3_{10} = 11.01010101......01_{2}$$

The fraction repeats and there is obviously no exact result. We will have to choose a bit position where we truncate the value. For examole, if we choose bit position six, we end up with an approximation.

$$3\ 1/3_{10}\ N\ 11.010101$$

The rounding error introduced by this truncation can be computed by converting the truncated fraction.

$$11.01010101 = 2 + 1 + .25 + .0625 + .015625$$
$$= 3.328125$$

The error is about .1%. The possibility of this type of rounding error must always be taken into consideration when using binary fractions, particularly in division operations. Very few numbers result in exact binary fractions, so the possibility of error will be ever present.

BINARY ARITHMETIC AND LOGIC INSTRUCTIONS

BINARY ARITHMETIC AND LOGIC INSTRUCTIONS

All computers provide a number of instructions which are used to perform arithmetic and logic operations on data. As discussed elsewhere, computer data consists of patterns of bits in registers and memory locations. However, there is a fundamental difference between the way the arithmetic instructions and the logic instructions treat this binary data. The arithmetic instructions interpret the data as numbers. The logic instructions, on the other hand, interpret the data as a collection of individual bits.

11.1 Computer Arithmetic Instructions

The most basic computer arithmetic instruction is addition. This instruction and the logic complement instruction can be used to to implement any known mathematical function. As a result, many computers offer addition as their only arithmetic instruction. After addition, the next most common arithmetic instruction is subtraction. This is because subtraction can be performed using the same basic hardware as addition. After addition and subtraction you have to go to a considerably more complex computer to get multiplication and division as built-in functions. The hardware required for these operations is considerably more complex than that used for addition and subtraction. As of this writing (July 1976) there are no microprocessors with built-in multiply and divide hardware. This will certainly change. All of these operations use the contents of the computer's accumulator(s) and

11.1.2 Binary Arithmetic

Binary Arithmetic is performed using the ALU and two operands. The operation can be performed using either unsigned numbers or signed twos complement numbers, depending upon the operation being perfomed. Most computers perform addition and subtraction as unsigned operations. They do provide flags to indicate the result in signed twos complement, but it is up to you to keep track of the sign and magnitude.

Addition is performed by adding the contents of an operand to the contents of the accumulator. If the result is greater than the largest number which can be represented in the accumulator, a flag will be set to indicate a carry out has occurred. For example, consider the operation of adding the number 15_{10} to an 8-bit accumulator which contains 25_{10}. The operation would be performed as follows.

```
Accumulator  00011001
  + Operand  00001111
             --------
     Result  00101000 = 40₁₀
```

Now consider the addition of 111_{10} to 145_{10} in the accumulator.

```
Accumulator  10010001
  + Operand  01101111
             --------
          1/ 00000000
     carry out
```

The result of this operation is 256_{10}. It causes a carry out to indicate that the accumulator overflowed.

Subtraction is performed by taking the twos complement of the subtrahend and adding it to the minuend in the accumulator. Thus to subtract 10_{10} from 25_{10} we would perform the following operation.

Subtrahend	00001010
Form Twos Complement	11110110
Add to Accumulator	00011001
carry out	1/ 00001111 = 15_{10}

Ignoring the carry out, we have a result of 15_{10}. Now consider the subtraction of 35 from 15.

Subtrahend	00100011
Form Twos Complement	11011101
Add to Accumulator	00001111
no carry	0/ 11101100

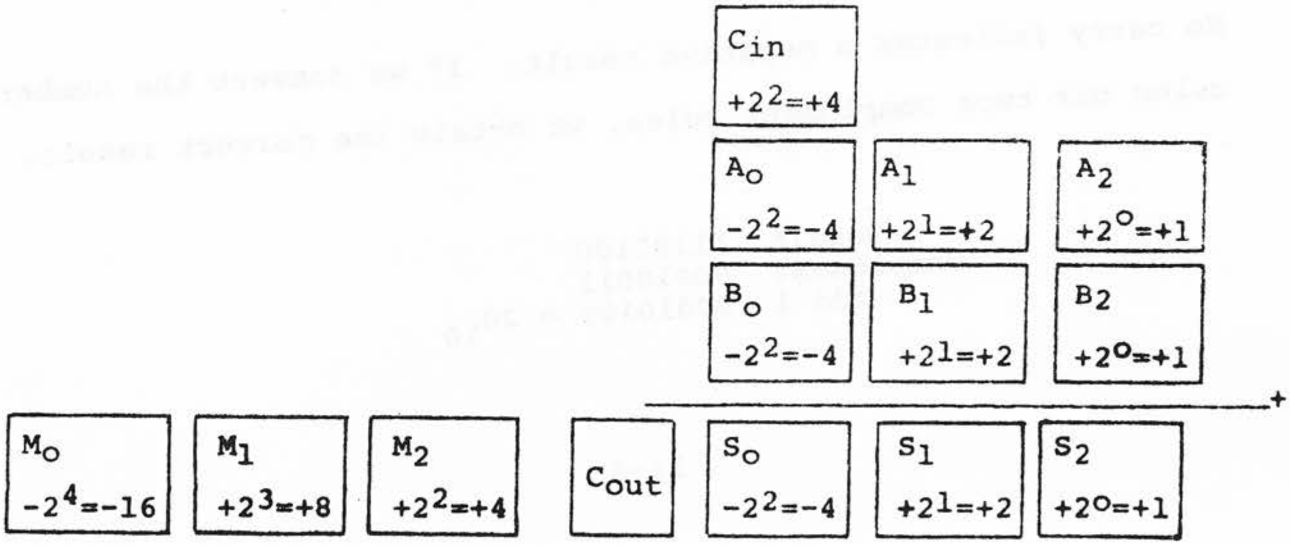
No carry indicates a negative result. If we convert the number using our twos complement rules, we obtain the correct result, -20.

Result	11101100
Complement	00010011
Add 1	00010100 = 20_{10}

Notice that the sense of a carry after subtraction is reversed from that of addition. A carry out indicates that the subtrahend was smaller than the minuend and the result of the subtraction was positive. No carry out indicates that the subtrahend was larger than the minuend and that the result was negative. This is called a borrow condition, and it is analagous to overflow in an addition operation. To avoid confusion about the reversal of the state of the carry flag, many computer ALU's automatically complement the carry flag after a subtraction. This makes its state after a subtraction match more closely its state after an addition (i.e. carry set if result caused a borrow, clear if the result did not cause a borrow).

11.1.3 Overflow and Underflow with Signed Arithmetic

For the purpose at hand we will use a 3-bit binary adder which will accept a pair of 3-bit inputs (the addends) to form a 4-bit output by the rules of straight ("unsigned") binary addition. In addition to the 10 bits making up the inputs and the output, we will define one more signal, which is the carry into the leftmost bit position of the addends:



To keep a running count of overflow and underflow events, we will need one more register, here also shown as 3 bits wide. Because spill-out from the adder may have a weight of +4 or -4 (overflow or underflow), it will be convenient to assign a weight of 4 to M_2, and to treat the contents of M as another signed number which may be incremented or decremented by the same add/subtract strategy we will develop for A, B, and S. The bit weights for all bits are shown in the diagram; they correspond to the standard convention for two's complement signed integers. In the examples below, the bit locations and formats will be as shown above, but the bit weights will not be shown.

Two factors may be held accountable for most of the confusion around signed arithmetic:

1. The term MSB (most significant bit) is often used with an implicit convention which may assign the name MSB to either bit 0 or bit 1 of a word. In recognition of this, we will not use the term MSB here.
2. The signals in the 0 column may have either a positive weight (C_{in}) or a negative weight (A_o, B_o). The adder makes no such distinction; the interpretation of bits as weighted numbers is strictly ours. Having recognized the sources of confusion, let us attempt to create some order by inspecting all possible combinations of sign bits and carry-in signals:

A. $C_{in} = 0; A_o = 0; B_o = 0$

	0		
	0 1 0	+2	Addition of two small positive
	0 0 1	+1	numbers.
000 0	0 1 1	+3	No overflow, no underflow

B. $C_{in} = 0; A_o = 0; B_o = 1$ (or $A_o = 1; B_o = 0$)

	0		
	0 1 0	+2	Addition of a positive and a
	1 0 1	-3	negative number, result negative.
000 0	1 1 1	-1	No overflow, no underflow

C. $C_{in} = 0; A_o = 1; B_o = 1$

	0		
	1 0 1	-3	Addition of two negative numbers.
	1 1 0	-2	Underflow
000 1	0 1 1	-5	C_o has a weight of 2 x -4, which
111			is redistributed to M_2 and S_o.
111	1 1 1	-4 + -1	

D. $C_{in} = 1; A_o = 0; B_o = 0$

	1		
	0 1 1	+3	Addition of two positive numbers.
	0 0 1	+1	Overflow
000 0	1 0 0	+4	S_o as formed has a weight of +4,
001	0 0 0	+4 + 0	which is moved to M_2

E. $C_{in} = 1$; $A_o = 0$; $B_o = 1$ (or $A_o = 1$; $B_o = 0$)

$$\begin{array}{rcc} & 1 & \\ & 0\ 1\ 0 & +2 \\ & \underline{1\ 1\ 0} & \underline{-2} \\ 000\ 1 & 0\ 0\ 0 & 0 \end{array}$$

Addition of a positive and a negative number.

No overflow, no underflow

Note that the weight of C_o is zero; C_o is produced by "addition" of C_{in} with weight +4 and B_o with weight -4.

F. $C_{in} = 1$; $A_o = 1$; $B_o = 1$

$$\begin{array}{rcc} & 1 & \\ & 1\ 1\ 0 & -2 \\ & \underline{1\ 1\ 1} & \underline{-1} \\ 000\ 1 & 1\ 0\ 1 & -3 \end{array}$$

Addition of two small negative numbers.

No underflow, no overflow

Again C_{in} neutralizes one of the sign bits. The other sign bit reappears as S_o

In summary:

1. If carry-in is produced, but no carry-out is generated, then overflow has occurred.
2. If a carry-out is produced without help from a carry-in, then underflow occurred.
3. If no carries are generated, neither overflow nor underflow occurred.

4. If a carry-in produces a carry-out, this amounts to neutraliz tion of the positive weight of the carry-in by the negative weight of one of the sign bits. Neither overflow nor underflow has occurred.

5. Whenever overflow or underflow occurs, S_o must be complemented, to make S suitable as input to the adder for further arithmetic operations. If $C_{out} = 1$, M must be decremented; if $C_{out} = 0$, M must be incremented.

In logical terms, spill-out can be detected as $C_{in} \oplus C_{out}$. The sign of the spill-out is given by C_{out}.

When all additions have been made, we should combine the spill-out counter with the S register to make a double-length bit string representing the end result in two's complement format. Now there are two bit positions with an absolute weight of 4: the low-order sign bit, S_o, and the LSB of the spill-out counter, M_2. These two bits must be combined, and then the vacated bit position must be eliminated to shift the high order bits into positions corresponding to their assigned weight.

11.2 Computer Logic Instructions

In contrast to the arithmetic instructions, the logic instructions perform their operations with no regard to the number representation being used. The numbers being operated upon are simply treated as strings of bits. That is why these operations are often referred to as bit by bit operations. The operation performed on one bit in no way affects the operation upon adjacent bits.

The four most common computer logic instructions are Complement, AND, OR, and Exclusive OR. These operations (except complement) use the contents of the accumulator and another data source as operands, with the result ending up in the accumulator.

11.2.1 Logic Complement

The complement instruction replaces each bit in the accumulator with its logic complement. Thus if the accumulator contains 1 0 1 0 1 1 0 1, the complement operation yields the following result.

Accumulator	1 0 1 0 1 1 0 1
Complement	0 1 0 1 0 0 1 0

11.2.2 Logic AND

The Logic AND operations (Symbol $\wedge$) operates upon the bits of the accumulator and an operand according to the following truth table.

Accumulator Bit	0	0	1	1
Operand Bit	0	1	0	1
Result Bit	0	0	0	1

Thus only those bit positions which are logic ones in both the accumulator and the operand will be logic ones in the accumulator after a Logic AND operation has been performed. For example, consider the following Logic AND operation.

Accumulator	0 1 1 0 1 1 0 1
Λ Operand	1 1 0 1 1 0 1 1
Result	0 1 0 0 1 0 0 1

Only those bits which were ones in both operands are in the result.

11.2.3 Logic OR

The Logic OR operation (Symbol V) operates upon the bits of the accumulator and an operand according to the following truth table.

Accumulator Bit	0 0 1 1
Operand Bit	0 1 0 1
Result Bit	0 1 1 1

Bit positions which are logic ones in either the accumulator or the operand will be Logic ones in the accumulator after a Logic OR operation has been performed. For example, consider the following Logic OR operation. Bit positions which are Logic ones in either the accumulator or the operand will be Logic ones in the accumulator after a Logic OR operation has been performed. For example, consider the following Logic OR operation.

Accumulator	1 0 1 1 0 1 1 0
V Operand	0 0 1 1 0 0 1 1
Result	1 1 1 1 0 1 1 1

All bits which were ones in both operands are ones in the results.

11.2.4 Logic XOR

The Logic Exclusive OR operation (Symbol A, o-ten called XOR) is not found in all computers. It operates upon the bits of the accumulator and an operand according to the following truth table

Accumulator Bit	0 0 1 1
Operand Bit	0 1 0 1
Result Bit	0 1 1 0

Bit positions which are a Logic one in either the accumulator or the operand but not both will be Logic ones in the accumulator after a logic XOR operation has been performed. For example, consider the following Exclusive OR operation.

$$\begin{array}{rl} \text{Accumulator} & 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \forall\ \ \text{Operand} & \underline{1\ 0\ 1\ 1\ 0\ 1\ 1\ 0} \\ \text{Result} & 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1 \end{array}$$

Those bits which were ones in only one of the operands are ones in the result.

APPENDIX A
Sym/650X Information Sources

6500 MICROPROCESSOR SUPPLIERS

Commodore Business Machines
901 California Avenue
Palo Alto, CA. 94304 415-326-4000

Rockwell Microelectronic Devices
P.O. Box 3669
Anaheim, CA. 92803 714-632-3729

Synertek Inc.
3001 Stender Way
Santa Clara, CA. 95051 408-988-5600

6500 BASED MICROCOMPUTER SUPPLIERS

AB Computers
P.O. Box 104
Perkasic, PA. 18944 215-257-8195

Apple Computer Inc.
20863 Stevens Creek Blvd.
Bldg. B3-C
Cupertino, CA. 95014 408-996-1010

Carnegic Electronics
100 Kings Road
Madison, N.J. 07940 201-822-1236

Commodore Business Machines
901 California Avenue
Palo Alto, CA. 94304 415-326-4000

The Computerist Inc.
P.O. Box 3
So. Chelmsford, MA. 01824 617-256-3649

Ohio Scientific
11679 Hayden
Hiram, OH. 44234

Riverside Electronics Design Inc.
1700 Niagara St.
Buffalo, N.Y. 14207 716-875-7070

RNB Enterprises, Inc.
2967 West Fairmont Ave.
Phoenix, AZ. 85017 602-265-7564

Seawell Marketing Inc.
315 N.W. 85th
Seattle, WA. 98117 206-782-9480

650X SOFTWARE SOURCES

650X USER NOTES
109 Centre Avenue
West Norristown, PA. 19401

ARESCO
314 Second Avenue
Haddon Heights, N.J. 08035

6502 PROGRAM EXCHANGE
2920 Moana
Reno, Nevada 89509

THE COMPUTERIST
P.O. Box 3
So. Chelmsford, MA. 01824

PYRAMID DATA SYSTEMS
6 Terrace Avenue
New Egypt, N.J. 08533

MICRO-WARE, LTD.
27 Firstbrooke Road
Toronto, Ontario
Canada M4E 2L2

KENNETH W. ENSELE
1337 Foster Road
Napa, CA. 94558

JOHNSON COMPUTER
P.O. Box 523
Medina, Ohio 44258

SEAWELL MARKETING INC.
315 N.W. 85th
Seattle, WA. 98117

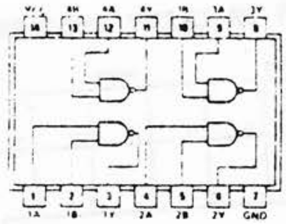
SYNERTEK SYSTEMS CORP.
3001 Stender Way
Santa Clara, CA. 95052

APPENDIX B

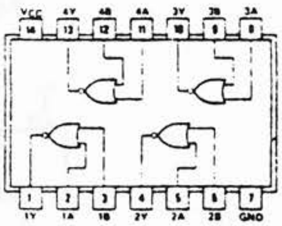
TTL REFERENCE SHEETS

TTL REFERENCE SHEET #1

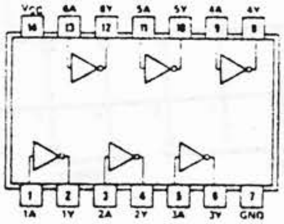
7400



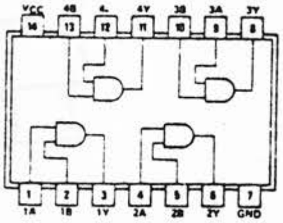
7402



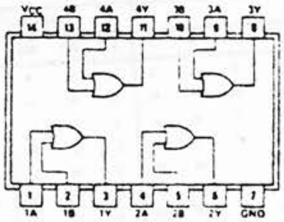
7404



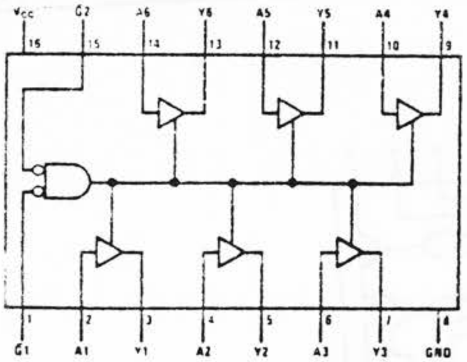
7408



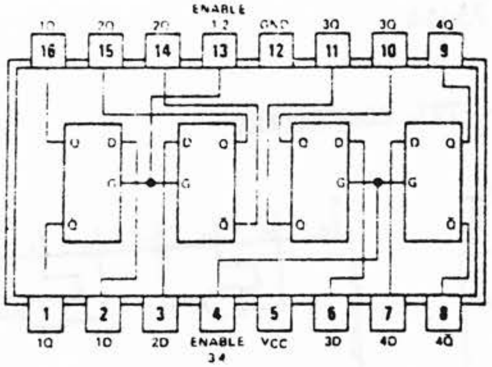
7432



8095 TRI-STATE BUFFER



7475 QUAD LATCH



FUNCTION TABLE
(Each Latch)

INPUTS		OUTPUTS	
D	G	Q	$\bar{Q}$
L	H	L	H
H	H	H	L
X	L	Q_0	$\bar{Q}_0$

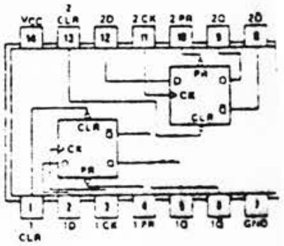
H = high level, L = low level, X = irrelevant

Q_0 = the level of Q before the high-to-low transition of G

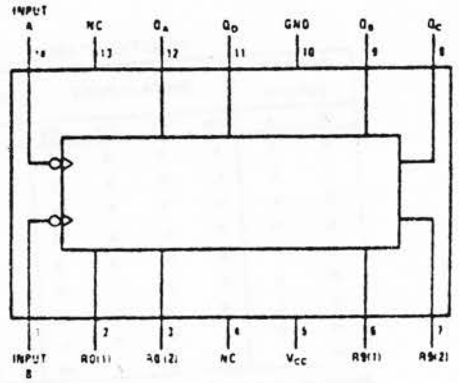
7474

FUNCTION TABLE

INPUTS				OUTPUTS	
PRESET	CLEAR	CLOCK	D	Q	$\bar{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\bar{Q}_0$



7490 DECADE COUNTER



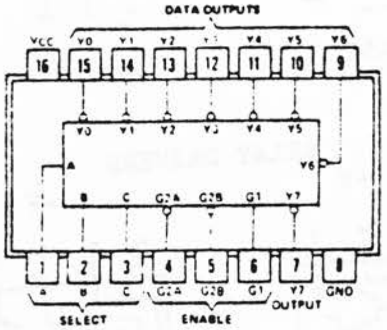
BCD COUNT SEQUENCE
(See Note A)

COUNT	OUTPUT			
	Q_D	Q_C	Q_B	Q_A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H

RESET/COUNT TRUTH TABLE

RESET INPUTS				OUTPUT			
R0(1)	R0(2)	R9(1)	R9(2)	Q_D	Q_C	Q_B	Q_A
H	H	L	X	L	L	L	L
H	H	X	L	L	L	L	L
X	X	H	H	H	L	L	H
X	L	X	L	COUNT			
L	X	L	X	COUNT			
L	X	X	L	COUNT			
X	L	L	X	COUNT			

74138 1-OF-8 DECODER

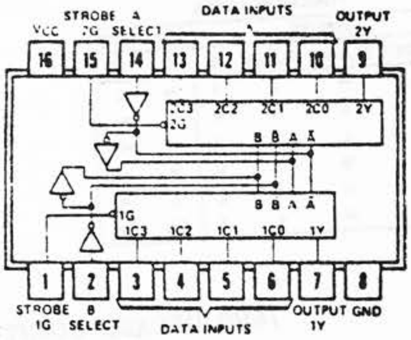


INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

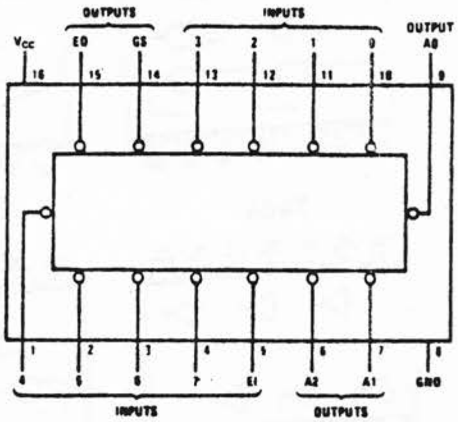
*G2 = G2A + G2B

TTL REFERENCE SHEET #2

74153 MULTIPLEXER



74148 PRIORITY ENCODER



FUNCTION TABLE

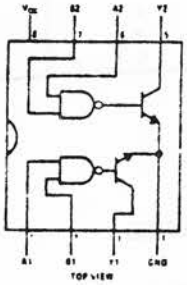
SELECT INPUTS		DATA INPUTS				STROBE	OUTPUT
B	A	C0	C1	C2	C3	G	Y
X	X	X	X	X	X	H	L
L	L	L	X	X	X	L	L
L	L	H	X	X	X	L	H
L	H	X	L	X	X	L	L
L	H	X	H	X	X	L	H
H	L	X	X	L	X	L	L
H	L	X	X	H	X	L	H
H	H	X	X	X	L	L	L
H	H	X	X	X	H	L	H

Select inputs A and B are common to both sections.
H = high level, L = low level, X = irrelevant

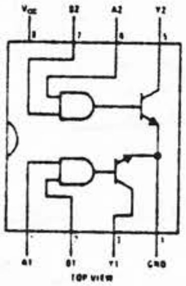
54148/74148

INPUTS									OUTPUTS				
EI	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

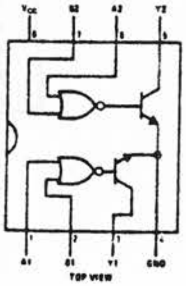
DUAL PERIPHERAL DRIVERS 300 mA, 20 V



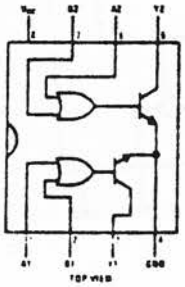
75451



75452



75453

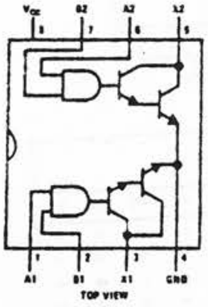


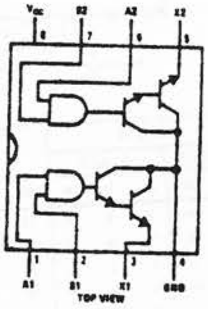
75454

RELAY DRIVERS

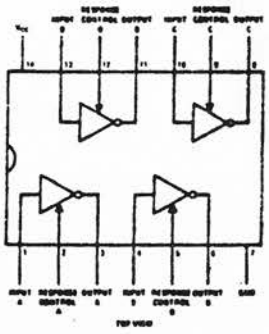
+V

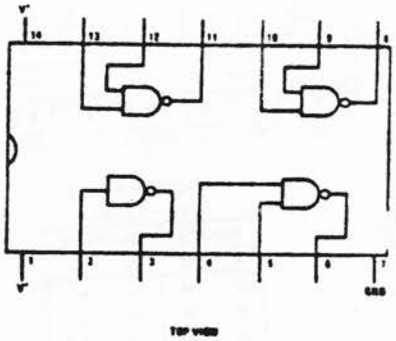
-V





RS-232C - TTL CONVERTERS





DS3686

DS3687

1489

1488

APPENDIX C

MICROCOMPUTER BIBLIOGRAPHY

AN INTRODUCTION TO MICROCOMPUTERS:
by Adam Osborne
Volume I: Basic Concepts #2001
Volume II: Some Real Products #3001
Osborne & Associates, Inc.
P.O. Box 2036
Berkley, CA. 94702

MICROCOMPUTER AND MICROPROCESSOR
by Hilburn and Julick
Prentice-Hall, Inc.
1976
pp. 375

The book is intended for all persons involved in the design, use, or maintenance of digital systems using microcomputers. The book is written at a level which can be understood by persons with little previous experience.

Topics include: digital logic, number systems and codes, microcomputer architecture, software, interfacing and peripheral devices, microcomputer systems (4040, 8080, 8008, 6800, IMP-4, PPS4, COSMAC, PPS-8, PACE) design methodology and applications.

MICROPROCESSORS AND MICROCOMPUTERS
by Branko Soucek
Wiley-Interscience
1976
pp. 607

A general introduction to digital systems and microcomputers with detailed descriptions of popular 4, 8, 12, and 16 bit microprocessors including the 6800, 8080, and LSI-11.

PROGRAMMING MICROPROCESSORS
William Barden Jr.
Radio Shack Inc.

TTL COOKBOOK
By Don Lancaster
1974
pp. 335
Howard W. Sams, Inc.

SOFTWARE DESIGN FOR MICROPROCESSORS
by John G. Wester and William D. Simpson
Texas Instruments, Inc.
1976
pp. 372
order from: Texas Instruments
P.O. Box 3640, M/S-84
Dallas, TX. 75283

Book was written to assist technical and non-technical people in taking their first steps toward designing microprocessors and related software. Topics range from basic binary numbers to complex examples of microcomputer applications. Book was written primarily for those with little or no programming experience, but it contains excellent application examples which should be of interest even to seasoned programmers.

CMOS COOKBOOK
BY Don Lancaster
1977
Howard W. Sams, Inc.

TV TYPEWRITER COOKBOOK
by Don Lancaster
1976
Howard W. Sams Inc.

SOFTWARE DESIGN FOR MICROCOMPUTERS
by Carol Anne Ogden
1978
Prentice-Hall Inc.
Englewood Cliffs, N.J.

MICROCOMPUTER DESIGN
by Carol Anne Ogden
1978
Prentice-Hall Inc.
Englewood Cliffs, N.J.

MICROCOMPUTER SYSTEMS PRINCIPLES
by R. C. Camp, T. A. Smay, C. J. Triska
1978
Matrix Publishing, 30 N.W. 23rd Pl., Portland, OR. 97210

FUNDAMENTALS OF MICROCOMPUTER ARCHITECTURE
by Keith L. Doty
1979
Matrix Publishing, 30 N.W. 23rd Pl., Portland, OR. 97210

B. PERIODICALS

AMERICAN LABORATORY

BYTE

published monthly by Byte Publications, Inc.
70 Main Street
Peterborough, N.H. 03458

COMPUTER DESIGN

published monthly by Computer Design Publishing Co.
P.O. Box A
Winchester, MA 01890

free to qualified persons

CONTROL ENGINEERING

published monthly by Control Engineering
666 Fifth Avenue
New York, N.Y. 10019

free to qualified persons

Contains useful articles on applications of microcomputers to industrial control.

DIGITAL DESIGN

published monthly by Benwill Publishing Corp.
Circulation Director
DIGITAL DESIGN
167 Corey Road
Brookline, MA 02146

free to qualified persons

Dr. Dobbs Journal of COMPUTER CALISTHENICS & ORTHODONTIA

published 10 times peryear by Peoples Computer Company
Box E
Menlo Park, CA 94025

Devoted to publication of microcomputer oriented software such as TINY BASIC.

ELECTRONIC DESIGN

published biweekly by Hayden Publishing Company, Inc.
50 Essex Street
Rochelle Park, NJ 07662

free to qualified persons

ELECTRONIC DESIGN NEWS

published monthly by Cahners Publishing Co.

free to qualified persons, very hard to get

ELECTRONIC ENGINEERING TIMES

published biweekly by CMP Publications
280 Community Drive
Great Neck, NY 11021

free to qualified persons

useful for news and announcements of new microprocessor products. Has bingo card for new product ads.

ELECTRONICS

published biweekly by McGraw-Hill, Inc.
McGraw-Hill Building
1221 Avenue of the Americas
New York, NY 10020

INTERFACE

published monthly by Southern California Computer Society
free with membership to SCCS.

INTERFACE AGE (new magazine by publishers of original INTERFACE)

published monthly by McPheters, Wolfe & Jones
6515 Sunset Blvd.
Suite 202
Hollywood, CA 90028

INSTRUMENTS AND CONTROL SYSTEMS

published monthly by Chilton Company
Chilton Way
P.O. Box 2025
Radnor, PA 19089
Att'n: Circualtion Dept.

free to qualified persons

KILOBAUD

published monthly by Publisher of 73
Kilobaud Magazine
Peterborough, N.H. 03458

Computer Hobby Magazine

MINI-MICRO SYSTEMS (formerly Modern Data)

published monthly by Modern Data Services, Inc.
5 Kane Industrial Lane
Hudson, MA 01749

free to qualified persons

POPULAR ELECTRONICS

published monthly by Ziff-Davis Publishing Co.
One Park Avenue
New York, N.Y. 10016

electronics hobby magazine

RADIO-ELECTRONICS

published monthly by Gernsback Publications, Inc.
Subscription Services
Box 2520
Boulder, Colorado 80302

Hobby Magazine

73 MAGAZINE

published monthly by 73, Inc.
Peterborough, NH 03458

applications of microcomputers to radio communications

PEOPLE'S COMPUTERS

Published bimonthly by People's Computer Company
1263 El Camino Real
Box E
Menlo Park, CA 94025

ELECTRONIC NEWS

published biweekly by Fairchild Publishing Company
7 East 12th Street
New York, NY 10003

MANUALS:

The Bell & Howell. Pressure Transducer Handbook. Pasadena, Calif.: CEC/Instruments Division, 1974.

EDN. Microprocessor Design Series.

Foxboro. Process-Control Instrumentation. Foxboro, Massachusetts. The Foxboro Company.

Honeywell. An Evolutionary Look at Process Control/1. Minneapolis, MINN.: Honeywell, 1975.

Coden: Avinap. Advances in Instrumentation. Pittsburg, Pennsylvania. Instrument Society of America, 1978.

Instrument Society of America. Binary Logic Diagrams for Process Operations. Pittsburg, PA.: 1976.

Instrument Society of America. Instrumentation Symbols and Identification. Pittsburgh, PA.:

Jordan Controls. Remote Positioning Controls for Industry. Milwaukee, WI.: Jordan Controls Inc., 1972.

MOS. KIM Assembler Manual (Preliminary), Norristown, PA.: MOS Technology, Inc. 1976.

Powers Regulator Company. Fundamentals of Control. Skokie, ILL.: Powers Regulator Company, 1970.

Synertek Systems Corporation. SYM Reference Manual. Santa Clara, CA.: Synertek Systems Corporation, 1978.

National Semiconductor. Pressure Transducer Handbook. Santa Clara, CA.: National Semiconductor Corporation, 1977.

APPENDIX D

GLOSSARY OF COMMONLY USED TERMS

ABSOLUTE ADDRESSING - SEE DIRECT ADDRESSING

ABSOLUTE INDEXED ADDRESSING - The effective address is formed by adding the index register (X or Y) to the second and third byte of the instruction.

ACCUMULATOR - A register that holds one of the operands and the result of arithmetic and logic operations that are performed by the central processing unit. Also commonly used to hold data transferred to or from I/O devices.

ACCUMULATOR ADDRESSING - One byte instruction operating on the accumulator.

ACIA - Is an Asynchronous Communications Interface Adapter. This is an NMOS LSI device produced by Motorola for interfacing Serial ASCII devices to a micro-processor system.

ADDRESS - A number that designates a memory or I/O location.

ADDRESS BUS - A multiple-bit output Bus for transmitting an address from the CPU to the rest of the system.

ALGORITHM - The sequence of operations which defines the solution to a problem.

ALPHANUMERIC - Pertaining to a character set that contains both letters and numerals and usually other characters.

ALU (ARITHMETIC/LOGIC UNIT) - The unit of a computing system that performs arithmetic and logic operations.

ARRAY - An organized group of data stored in a block of memory. By convention, the location of the array is specified as the location of the first data item in the array.

ASCII CODE - The American Standard Code for Information Interchange. A seven-bit character code without the parity bit, or an eight-bit character code with the parity bit.

ASSEMBLER - A program that translates symbolic operation codes into machine language, symbolic addresses to memory addresses and assigns values to all program symbols. It translates source programs to object programs.

ASSEMBLY DIRECTIVE - A mnemonic that modifies the assembler operation but does not produce an object code (e.g., a pseudo instruction).

ASSEMBLY LANGUAGE - A collection of symbolic labels, mnemonics, and data which are to be translated into binary machine codes by the assembler.

ASYNCHRONOUS - Not occurring at the same time, or not exhibiting a constant repetition rate; irregular.

BASE - SEE RADIX

BCD (BINARY CODED DECIMAL) - A means by which decimal numbers are represented as binary values, where digits from Ø - 9 are represented by the four-bit binary codes from ØØØØ-1ØØ1.

BIDIRECTIONAL DATA BUS - A data bus in which digital information can be transferred in either direction.

BINARY - The base-two number system. All numbers in this system are presented by bit strings, in which each bit indicates the presence or absence of an integral power of two. The power of two represented by each bit is called the weight of that bit; it is defined by the position of the bit in the bit string.

BIT - The smallest unit of information which can be represented. A bit may be in one of two states, represented by the binary digits Ø and 1.

BLOCK DIAGRAM - A diagram in which the essential units of any system are drawn in the form of blocks, and their relationship to each other is indicated by appropriate connecting lines.

BRANCH INSTRUCTION - An instruction that causes a program jump to a specified address and execution of the instruction at that address. During the execution of the branch instruction, the central processor replaces the contents of the program counter with the specified address.

BREAKPOINT - Pertaining to a type of instruction, instruction digit, or other condition used to interrupt or stop a computer at a particular place in a program. A place in a program where such an interruption occurs or can be made to occur.

BUFFER - A noninverting digital circuit element that may be used to handle a large fan-out or to invert input and output levels.

- A storage device used to compensate for a difference in rate flow of data, or time of occurrence of events, during transmission of data from one device to another.

BYTE - A sequence of eight adjacent binary digits treated as a unit.

CALL - A special type of jump in which the central processor is logically required to "remember" the contents of the program counter at the time that the jump occurs. This allows the processor later to resume execution of the main program, when it is finished with the last instruction of the subroutine.

CARRY - The overflow value that results from the addition of two bits or digits in an addition column.

CARRY (2)
- A single bit register that receives and holds the overflow information resulting from an addition. The same register often is also used to receive underflow information (BORROW) generated during a subtraction.

CASCADE - An arrangement of two or more similar circuits in which the output of one circuit provides the input of the next.

CLOCK - A device or a part of a device that generates all the timing pulses for the coordination of a digital system. System clocks usually generate two or more clock phases. Each phase is a separate square wave pulse train output.

CODING - The process of preparing a program from the flow chart defining an algorithm.

COMPILER - A language translator which converts individual source statements into multiple machine instructions. A compiler translates the entire program before it is executed.

COMPLEMENT - Reverse all binary bit values (ones become zeros, zeros become ones).

CONDITIONAL - In a computer, subject to the result of a comparison made during computation.

CONDITIONAL BREAKPOINT INSTRUCTION - A conditional jump instruction that causes a computer to stop if a specified bit is set. The routine then may be allowed to proceed as coded, or a jump may be forced.

CONDITIONAL JUMP - Also called conditional transfer of control. An instruction to a computer which will cause the proper one of two (or more) addresses to be used in obtaining the next instruction, depending on some property of one or more numerical expressions or other conditions.

CONTACT BOUNCE - The uncontrolled making and breaking of a contact when the switch or relay contacts are closed. An important problem in digital circuits, where bounces can act as clock pulses.

CPU (CENTRAL PROCESSING UNIT) - The unit of a computing system that controls the interpretation and execution of instructions; includes the ALU.

DATA BUS - A multi-line parallel path carrying information between a number of data sources and destinations. At any time, only a single data word may travel on the bus. This means that only a single device may act as a data source at any time, and generally only a single device will be allowed to accept data from the bus at any moment.

DEBUG - Detect, locate, and correct problems in a program or hardware.

EFFECTIVE ADDRESS - The actual address of the desired location in memory, usually derived by some form of calculation.

EMULATOR - A hardware system (with associated software) that can replace the processor in a micro-processor system. The emulator accepts and generates all signals that the processor would handle, preferably at the same speed. Inside the emulator, features are available to detect specific address and data patterns, etc.; this makes the emulator a powerful debugging tool, particularly for system hardware. Emulators are typically connected to a system under development by means of a plug and multi-lead cable that is inserted in place of the processor.

EVENT-TRIGGERED - A device, circuit, or program that responds to an event, rather than to the presence (or absence) of a condition.

FAN-OUT - The number of parallel loads within a given logic family that can be driven from one output node of a logic circuit.

FETCH - The operation by which a processor obtains a new op-code defining the next instruction to be executed. The term FETCH may also be used more loosely to describe the retrieval of an entire instruction from memory. The use of FETCH as a synonym for READ, e.g., to retrieve data from memory, is not recommended.

FIELD - An area of an instruction mnemonic.

FILE - A linearly organized string of data. Although FILE may refer to a string of data in memory, the name is usually reserved for data stored on magnetic tape, disc, or other storage media.

FIFO - A memory system organized in such a way that information can be stored and retrieved sequentially in such a way that the order is preserved: First-In-First-Out.

FLAG - A status bit which indicates that a certain condition has arisen during the course of arithmetic or logical manipulations or data transmission between a pair of digital electronic devices. Some flags may be tested and thus be used to determine subsequent actions.

FLAG REGISTER - A register consisting of a group of flag flipflops.

FLOPPY DISC - A flexible disc, usually made of mylar, and coated with magnetic oxide. Information may be stored on circular TRACKS on the disc, which are subdivided into SECTORS. The recording mechanism is similar to that used for magnetic tape. Because the read/write head can be positioned rapidly over any track on the disc, the disc can be used as a random-access storage device. In contrast, information held on tape can only be accessed sequentially.

DEBOUNCED - Refers to a switch or relay that no longer exhibits contact bounce.

DECODER/DRIVER - A code conversion device that can also have sufficient voltage or current output to drive an external device such as a display or a lamp monitor.

DEMULTIPLEXER - A digital device that directs information from a single input to one of several outputs. Information for output-channel selection usually is presented to the device in binary weighted form and is decoded internally. The device also acts as a single-pole multiposition switch that passes digital information in a direction opposite to that of a multiplexer.

DESTINATION - Register, memory location or I/O device which can be used to receive data during instruction execution.

DEVICE SELECT PULSE - A software-generated positive or negative clock pulse from a computer that is used to strobe the operation of one or more I/O devices, including individual integrated circuit chips.

DIRECT ADDRESSING - The second and third byte of the instruction contain the address of the operand to be used.

DMA (DIRECT MEMORY ACCESS) - Suspension of processor operation to allow peripheral units external to the CPU to exercise control of memory for both READ and WRITE without altering the internal state of the processor.

DYNAMIC RAM - A random access memory that uses a capacitive element for storing a data bit. They require REFRESH. A type of memory in which information is stored in the form of charge on tiny capacitors. Because the capacitors are not perfect, they tend to discharge with time. To avoid loss of information, the capacitors must be periodically read and restored to their original state. The read/restore operation is called REFRESH.

EBCDIC - The Extended Binary Coded Decimal Interchange Code, a digital code primarily used by IBM. It closely resembles the half-ASCII code.

EDGE - The transition from logic Ø to logic 1, or from logic 1 to logic Ø, in a clock pulse.

EDGE TRIGGERED - A device which responds to a transition from one logic level to the other on one of its input leads is called an edge-triggered device. The device only responds to the transition, but not to either the high or the low level of the input signal.

EDITOR - A program used for preparing and modifying a source program or other file by addition, deletion or change.

FLOW CHART - A symbolic representation of the algorithm required to solve a problem.

FREQUENCY - The number of recurrences of a periodic phenomenon in a unit of time. Electrical frequency is specified as cycles per second, or Hertz. (Hz)

FULL DUPLEX - A data transmission mode which provides simultaneous and independent transmission and reception.

HALF-ASCII - A 64-character ASCII code that contains the code words for numeric digits, alphabetic characters, and symbols but not keyboard operations.

HALF DUPLEX - A data transmission and reception mode which provides both transmission and reception but not simultaneously.

HANDSHAKE - Interactive communication between two systems or system components, such as between the CPU and a peripheral; required whenever the communicating systems operate on independent time scales.

HARDWARE - Physical equipment: mechanical, electrical, or electronic devices.

HEXADECIMAL - A number system based upon the radix-16, in which the decimal numbers 0 through 9 and the letters A through F represent the sixteen distinct states in the code.

HIGH ADDRESS BYTE - The eight most significant bits in the 16-bit memory address word. Abbreviated H or HI.

IC (INTEGRATED CIRCUIT) - (1) A combination of interconnected circuit elements inseparably associated on or within a continuous substrate. (2) Any electronic device in which both active and passive elements are contained in a single package. In digital electronics, the term chiefly applies to circuits containing semiconductor elements.

IMMEDIATE ADDRESSING - An addressing mode in which the data are imbedded as part of the instruction.

IMPLIED ADDRESSING - A one-byte instruction that stipulates an operation internal to the processor. DOES NOT require any additional operand.

INCREMENT - To increase the value of a binary word by one.

INDEXED ADDRESS - An indexed address is a memory address formed by adding immediate data included with the instruction to the contents of some register or memory location.

INDEXED INDIRECT ADDRESSING - The instruction contains the address of an array. An item is selected from the array according to the value held in an index register. The selected array item is then used as the address of the data. If the array storage locations cannot hold a complete address, then some form of addressing economy, such as zero-page addressing, may have to be invoked.

INDIRECT ABSOLUTE ADDRESSING - The second and third bytes of the instruction contain the address of the first of two bytes in memory that contain the effective address.

INDIRECT INDEXED ADDRESSING - The instruction contains the address of a pointer. The pointer in turn points to an array of data words (or bytes). A single item is selected from the array according to the value held in an index register. If the array address cannot be held in a single storage location, the pointer may be held in a pair of adjacent locations, with the instruction pointing at the first of these locations.

INDIRECT ADDRESS - An address used with an instruction that indicates a memory location or a register that in turn contains the actual address of an operand. The indirect address may be included with the instruction, contained in a register (register indirect address) or contained in memory location (memory directed indirect address).

INSTRUCTION - A statement that specifies an operation and the values or locations of its operands.

INSTRUCTION CODE - A unique binary number that defines an operation that a computer can perform.

INSTRUCTION CYCLE - A successive group of machine cycles, as few as one or as many as seven, which together perform a single microprocessor instruction within the microprocessor chips.

INSTRUCTION DECODER - A decoder within a CPU that decodes the instruction code into a series of actions that the computer performs.

INSTRUCTION REGISTER - A register that contains the instruction code.

INTERPRETER - A language translator which converts individual source statements into multiple machine instructions by translating and executing each statement as it is encountered. Can not be used to generated object code.

INTERRUPT - In a computer a break in the normal flow of a system or routine such that the flow can be resumed from that point at a later time. The source of the interrupt may be internal or external.

I/O DEVICE (INPUT/OUTPUT DEVICE) - Any digital device, including a single intergrated circuit chip, that transmits data or strobe pulses to a computer or receives data or strobe pulses from a computer.

JUMP - (1) To cause the next instruction to be selected from a specified storage location in a computer. (2) A deviation from the normal sequence of execution of instructions in a computer.

LABEL - One or more characters that serve to define an item of data or the location of an instruction or subroutine. A character is one symbol of a set of elementary symbols, such as those corresponding to typewriter keys.

LATCH - A simple logic storage element. A feedback loop used in a symmetrical digital circuit, such as a flipflop, to retain a state.

LEADING EDGE - The transition of a pulse that occurs first.

LED (LIGHT-EMITTING-DIODE) - A pn junction that emits light when biased in the forward direction.

LEVEL-TRIGGERED - The state of the clock input, being either logic Ø or logic 1 carries out a transfer of information or completes an action.

LIFO (LAST IN, FIRST OUT) - The latest data entered is the first data obtainable from a LIFO stack or memory section.

LSB (LEAST SIGNIFICANT BIT) - The digit with the lowest weighing in a binary number.

LISTING - An assembler output containing a listing of program mnemonics, the machine code produced, and diagnostics, if any. A program which is printed in a format that includes a location column and a contents column side by side with the corresponding program source statements. A listing, or listing file, is one of the output files that may be obtained from an assembler program. Listings are intended primarily as a diagnostic aid, but they are often supplied as part of the documentation for a program.

LOGIC - (1) The science dealing with the basic principles and applications of truth tables, switching, gating, etc. (2) See Logical Design. (3) Also called symbolic logic, a mathematical approach to the solution of complex situations by the use of symbols to define basic concepts. The three basic logic symbols are AND (•), OR (+), and NOT ($\overline{x}$ or x^1). Although the symbols • and + are used in the two disciplines, there is no relationship between the logical OR and the algebraic ADD, even though the jargon, and the printed literature, abound with questionable terms such as "sum-of-products." (4) In computers and information- processing networks, the systematic method that governs the operations performed on information, usually with each step influencing the one that follows. (5) The systematic plan that defines the interactions of signals in the design of a system for automatic data processing.

LOGICAL DECISION - The ability of a computer to make a choice between two alternatives; basically, the ability to answer yes or no to certain fundamental questions concerning equality and relative magnitude.

LOGICAL DESIGN - The synthesizing of a network of logical elements to perform a specified function. In digital electronics, these logical elements are digital electronic devices, such as gates, flipflops, decoders, counters, etc.

LOGICAL ELEMENT - In a computer or data-processing system, the smallest building blocks which operators can represent in an appropriate system of symbolic logic. Typical logical elements are the AND gate and the "flipflop."

LOOP - A sequence of instructions that is repeated until a conditional exit situation is met.

LOW ADDRESS BYTE - The eight least significant bits in the 16-bit memory address word. Abbreviated L or LO.

LSI (LARGE SCALE INTERGRATION) - Integrated circuits that perform complex functions. Such chips usually contain 1ØØ to 2,ØØØ gates.

MACHINE CODE - A binary code that a computer decodes to execute a specific function.

MACHINE CYCLE - A subdivision of an instruction cycle during which time a related set of actions occur within the microprocessor chip. All instructions are combinations of one or more of these machine cycles.

MACRO ASSEMBLER - An assembler routine capable of assembling programs which contain and reference macro instructions.

MACRO INSTRUCTION - A symbol that is used to represent a specified sequence of source instructions.

MAGNETIC CORE - A type of computer storage which employs a core of magnetic material with wires threaded through it. The core can be magnetized to represent a binary 1 or Ø.

MAGNETIC DRUM - A storage device consisting of a rapidly rotating cylinder, the surface of which will retain the data. Information is stored in the form of magnetized spots on the drum surface.

MAGNETIC DISC - A flat circular plate with a magnetic surface on which data can be stored by selective magnetization of portions of the flat surface.

MAGNETIC TAPE - A storage system based on the use of magnetic spots (bits) on metal or coated-plastic tape. The spots are arranged so that the desired code is read out as the tape travels past the read-write head.

MASKING - A process that uses a bit pattern to select bits from a data byte for use in a subsequent operation.

MEMORY - Any device that can store logic 1 and logic Ø bits in such a manner that a single bit or group of bits can be accessed and retrieved.

MEMORY ADDRESS - If n bits are available to define memory locations, then 2^n locations are available. Each possible pattern of n bits corresponds to one memory location. A bit pattern used to define a memory location may be called a memory address; it is generally handled as a binary or hex number.

MEMORY CELL - A single storage element of memory, capable of storing one bit of digital information.

MICROCOMPUTER - A computer system based on a microprocessor and containing all the memory and interface hardware necessary to perform calculations and specified information transformations.

MICROPROCESSOR - A central processing unit fabricated as one integrated circuit.

MICROPROGRAM - A computer program written in the most basic instructions or subcommands that can be executed by the computer. Frequently, it is stored in a read-only memory.

MNEMONIC - Symbols representing machine instructions designed to allow easy identification of the functions represented.

MODULUS - (1) The number of possible states of a counter or similar circuit. A decimal counter will count with a modulus of 1Ø, or "modulo 1Ø". Hence a decimal counter may also be called a "modulo 1Ø" counter.
(2) If a large number is counted with a modulo "n" counter, the final count displayed by the counter will be equal to the remainder that would be obtained if the large number is divided by the modulus, n. For example, "17 modulo 5" = 2, because the remainder or 17 divided by 5 equals 2.

MONITOR - Software or hardware that observes, supervises, controls, or verifies system operation.

MONOSTABLE MULTIVIBRATOR - Also called one-shot multivibrator, single-shot multi-vibrator, or start-stop multivibrator. A circuit having only one stable state, from which it can be triggered to change the state, but only for a predetermined interval, after which it returns to the original state.

MSI (MEDIUM SCALE INTEGRATION) - Integrated circuits that perform simple, self-contained logic operations, such as counters and flipflops.

MSB (MOST SIGNIFICANT BIT) - The digit with the highest weighting in a binary number.

MULTIPLEXER - A digital device that can select one of a number of inputs and pass the logic level of that input on to the output. Information for input-channel selection usually is presented to the device in binary weighted form and decoded internally. The device acts as a single-pole multiposition switch that passes digital information in one direction only.

NEGATIVE EDGE - The transition from logic 1 to logic Ø in a clock pulse.

NEGATIVE - EDGE TRIGGERED - Transfer of information occurs on the negative edge of the clock pulse.

NEGATIVE LOGIC - A form of logic in which the more positive voltage level represents logic Ø and the more negative level represents logic 1.

NESTING - A program structure in which subroutines may call other subroutines. Also, a program structure which allows interruption of interrupt service routines by other interrupting devices.

NYBBLE - A sequence of four adjacent bits, or half a byte, is a nybble. A hexadecimal or BCD digit can be represented in a nybble.

NON-OVERLAPPING TWO-PHASE CLOCK - A two phase clock in which the clock pulses of the individual phases do not overlap.

NON-VOLATILE MEMORY - A semiconductor memory device in which the stored digital information is not lost when the power is removed.

OCTAL - A number system based upon the radix 8, in which the decimal numbers Ø through 7 represent the eight distinct states.

ONE-BYTE-INSTRUCTION - A instruction that consists of eight contiguous bits occupying one memory location.

OPCODE - The bit pattern defining the operation that a processor must perform. It is always the first part of each instruction. Depending on the type of operation, the instruction may contain additional bytes of information, e.g., data or address information.

OPEN-COLLECTOR OUTPUT - An output from an integrated circuit device in which the final "pull-up" resistor in the output transistor for the device is missing and must be provided by the user before the circuit is completed.

OPERAND - Data which will be operated upon by an arithmetic/logic instruction; usually identified by the address portion of an instruction, explicity or implicity.

OPERATION - Moving or manipulating data in the CPU or between the CPU and peripherals.

PAGE - A page consists of all the locations that can be addressed by 8-bits (a total of 256 locations) starting at Ø and going through 255. The address within a page is determined by the lower 8-bits of the address and the page number (Ø through 255) is determined by the higher 8-bits of a 16-bit address.

PARITY - A method of checking the accuracy of bit strings. If even parity is used, the sum of all the 1's in a string and its corresponding parity bit is always even. If odd parity is used, the sum of all the 1's in the string, including the parity bit is always odd.

PARTITIONING - The process of assigning specified portions of a system's responsibility to perform specified functions.

PC - SEE PROGRAM COUNTER

PIA - PERIPHERAL INTERFACE ADAPTOR (MOS Technology's MPS 6520)

PERIPHERAL - A device or subsystem that is not part of a processor or its memory, but that is connected in such a way that it can communicate with the processor as the need arises. Greek for "on-the-outside-edge."

POLLING - Periodic interrogation of each of the devices that share a communications line to determine whether it requires servicing. The multiplexer or control station sends a poll that has the effect of asking the selected device, "Do you have anything to transmit?"

POP - Retrieving data from a stack.

PORT - A device or network through which data may be transferred or where device or network variables may be observed or measured.

POSITIVE EDGE - The transition from logic Ø to logic 1 in a clock pulse.

POSITIVE-EDGE TRIGGERED - Transfer of information occurs on the positive edge of the clock pulse.

POSITIVE LOGIC - A form of logic in which the more positive voltage level represents logic 1 and the more negative level represents logic Ø.

PRIORITY - A preferential rating. Pertains to operations that are given preference over other system operations.

PROCESSOR - Shorthand word for microprocessor.

PROGRAM - A group of instructions which causes the computer to perform a specified function.

PROGRAM COUNTER - A register containing the address of the next instruction to be executed. It is automatically incremented each time program instructions are executed.

PROGRAM LABEL - A symbol which is used to represent a memory address.

PROM (PROGRAMMABLE READ-ONLY MEMORY) - A read-only memory that is field programmable by the user.

PROPAGATION DELAY - The time required for a logic signal to travel through a logic device or a series of logic devices. It occurs as the result of four types of circuit delays - storage, rise fall and turn-on delay - and is the time between when the input signal crosses the threshold - voltage point and when the responding voltage at the output crosses the same voltage point.

PSEUDO-INSTRUCTION - A mnemonic that modifies the assembler operatior but does not produce an object code.

PULL-UP RESISTOR - A resistor connected to the positive supply voltage from the output collector of open-collector logic. Also used occasionally with mechanical switches to insure the voltage of one or more switch positions.

PULSE WIDTH - Also called pulse length. The time interval between the points at which the instantaneous value on the leading and trailing edges bears a specified relationship to the peak pulse amplitude.

PUSH - Putting data into a stack.

RADIX - Also called the base. The total number of distinct marks or symbols used in a numbering system. For example, since the decimal numbering system uses ten symbols, the radix is 1Ø. In the binary numbering system, the radix is 2, because there are only two marks or symbols (Ø and 1). In the octal numbering system, the radix is 8, and in the hexadecimal numbering system, the radix is 16.

RAM (RANDOM ACCESS MEMORY) - A semiconductor memory into which logic Ø and logic 1 states can be written (stored) and then read out again (retrieved).

READ - (1) Retrieval of information from memory. (2) In general, information is "read" from a source if the source supplies the information on the initiative from a device or system that is not part of the source, i.e., when the source responds. If a source acts to supply information, the source is said to write data.

REFRESH - The process of restoring transient information to its original quality before it has degraded to a level at which it can no longer be retrieved reliably. (1) For dynamic RAM memory: the process of reading the charge on the capacitive storage elements; during the read operation, the information stored on each element is restored to its original quality. (2) Scope displays: graphic information shown on the face of a cathode ray screen must be rewritten periodically at least 25 times per second to avoid flickering of the image.

REFRESH LOGIC - The Logic required to generate all the refresh signals and timing for dynamic RAM.

RELATIVE ADDRESSING - An addressing mechanism in which the location of an operand in memory is defined relative to the location of the instruction that refers to the operand. The address information contained in the instruction represents the "distance" in memory space between the instruction and the location of the operand. The processor then generates the actual address of the operand by adding the distance information to the content of the program counter.

RESET - A signal applied to a processor to force it into a pre-defined sequence of operations without regard for its current state. Normally, a RESET pulse will be applied on initial power-up, and when the processor has gone out of control owing to a software or hardware error. The RESET sequence should force all I/O ports into a known innocuous state, and it should load the program counter with a known address, so that the processor can begin execution of a start-up program. For the 6502 processor, the RESET sequence loads the program counter with the contents of memory locations $FFFC and $FFFD.

RETURN - A jump instruction for which the target address is not contained in the instruction, but is retrieved from the stack. This means that the target address must have been placed on the stack before the RETURN instruction is executed, normally by a subroutine call (JSR in 6502 assembly language).

RIPPLE COUNTER - A counter consisting of a string of flipflops connected in such a way that the output of each flipflop serves as the clock signal for its successor. A clock pulse applied to the first flipflop in the string must "ripple through" all the flipflops before it can affect the last one in the string.

RISE TIME - The time required for an output voltage of a digital circuit to change from a logic Ø to a logic 1 state.

ROM - A memory system whose contents can be read, but not changed, by the processor to which it is attached. The term is often used to describe memory devices that do not lose the information stored in them when power is disconnected. Strictly speaking, memory that is immune to loss of power should be called nonvolatile.

ROUTINE - A group of instructions that causes the computer to perform a specified function, e.g., a program.

SCRATCH PAD - The term applies to memory that is used temporarily by the CPU to store intermediate results.

SEVEN-SEGMENT DISPLAY - An electronic display that contains seven lines or segments spatially arranged in such a manner that the digits Ø through 9 can be represented through the selective lighting of certain segments to form the digit.

SEMICONDUCTOR MEMORY - A digital electronic memory device in which 1's and Ø's are stored, that is a product of semi-conductor manufacturing.

SHIFT REGISTER - A digital storage circuit in which information is shifted from one flipflop of a chain to the adjacent flipflop upon application of each clock pulse. Data may be shifted several places to the right or left, depending on additional gating and the number of clock pulses applied to the register. Depending on the number of positions shifted, the right-most bits are lost in a right shift, and the left-most bits are lost in a left shift.

SIMULATOR - A program that makes one computer behave as if it were another machine. Simulator programs are often used to test software written for a micro-processor on a large machine. Good simulators contain extensive diagnostic facilities; they allow insertion of multiple break points, and can maintain records of the amount of time spent in specified program loops, etc. Because each instruction must be decoded by software in the simulator, rather than by hardware, the execution time of any given program will generally be much slower on the simulator than on the machine for which the program is actually designed.

SOFTWARE - The means by which any defined procedure is specified for computer execution.

SOURCE - Register, memory location or I/O device which can be used to supply data.

SOURCE PROGRAM - A group of statements conforming to the syntax requirements of a language processor.

SPLIT DATA BUS - Two data buses, one for incoming communications and one for outgoing communications. An 8-bit data bus in a split data bus sytem takes 16 lines.

STACK - A specified section of sequential memory locations used as a LIFO (Last In, First Out) file. The last element entered is the first one available for output. A stack is used to store program data, subroutine return addresses, processor status, etc.

STACK POINTER (SP) - A register which contains the address of the system read/write memory used as a stack. It is automatically incremented or decremented as instructions perform operations with the stack.

STATEMENT - An instruction in source language.

STATIC RAM - A random access memory that uses a flipflop for storing a binary data bit. Does not require refresh.

STRING - A collection of data, (bits, bytes, words, characters, etc.) between which a sequential relationship exists.

SUBROUTINE - A routine that causes the execution of a specified function and which also provides for transfer of control back to the calling routine upon completion of the function.

SYMBOL - Any character string used to represent a label, mnemonic, or data constant.

SYMBOLIC ADDRESSING - An addressing notation in which the location of variables, entry points, etc., in a program are referred to by symbolic names rather than by (numerical) memory addresses. When the assembler translates the source program into machine language, it must assign the actual memory address for each symbolic address defined in the source. Symbolic addressing allows the programmer to revise a program without the need to update the addresses to which the program refers.

SYMBOLIC CODE - A code by which programs are expressed in source language; that is, storage locations and machine operations are referred to by symbolic names and addresses that do not depend upon their hardware-determined names and addresses.

SYMBOLIC CODING - In digital computer programming, any coding system using symbolic rather than actual computer addresses.

SYNCHRONOUS - Operation of a switching network by a clock pulse generator. All circuits in the network switch simultaneously, and all actions take place synchronously with the clock.

SYNTAX ERROR - An occurrence in the source program of a label expression or condition that does not meet the format requirements of the assembler program.

TABLE - A data structure used to contain sequences of instructions, addresses, or data constants.

TRAILING EDGE - The transition of a pulse that occurs last, such as the high-to-low transition of a positive clock pulse.

TRANSITION - The instance of changing from one state to a second state.

THREE-STATE DEVICE or TRI-STATE® DEVICE - A semiconductor logic device in which there are three possible output states: (1) a "logic Ø" state, (2) a "logic 1" state, or (3) a state in which the output is, in effect, disconnected from the rest of the circuit and has no influence upon it.

THREE-BYTE INSTRUCTION - An instruction that consists of twenty-four contiguous bits occupying three successive memory locations.

TREE - A group of gates connected to combine a number of parallel input signals into a single output signal.

TRUTH TABLE - A tabulation that shows the relation of all output logic levels of a digital circuit to all possible combinations of input logic levels in such a way as to characterize the circuit functions completely.

TWO-BYTE INSTRUCTION - An instruction that consists of sixteen contiguous bits occupying two successive memory locations.

TWO-PHASE CLOCK - A timing circuit that generates a pair of pulse strings, so that the true state from one output coincides with the false state from the other output. If the timing is controlled so the two outputs are never simultaneously true, then the pulse strings are called non-overlapping. The pulse strings are often referred to as phase 1 and phase 2.

UNCONDITIONAL - Not subject to conditions external to the specific computer instruction.

UNCONDITIONAL CALL - A call instruction that is unconditional.

UNCONDITIONAL JUMP - A computer instruction that interrupts the normal process of obtaining the instructions in an ordered sequence and specifies the address from which the next instruction must be taken.

UNCONDITIONAL RETURN - A return instruction that is unconditional.

VLSI (VERY LARGE-SCALE INTEGRATION) - Monolithic digital integrated circuit chips with a typical complexity of two thousand or more gates or gate-equivalent circuits.

VOLATILE MEMORY - A semiconductor memory device in which the stored digital information is lost when the power is removed.

WEIGHTING - If the numerical meaning of each bit in a bit string is defined uniquely by the position of the bit in the string, the string represents numerical information in a weighted code. The most common weighted codes are binary and BCD code.

WIRED-OR CIRCUIT - A circuit consisting of two or more semi-conductor devices with open collector outputs in which the outputs are wired together. The output from the circuit is at a logic Ø if device A or device C or ...... is at a logic Ø state.

WORD - The group of bits that can travel in parallel on the data bus of a computer system. For the 6502, a word is one byte or 8 bits wide. For large computers, a word may be up to 8 bytes wide.

WRITE - In semiconductors and other types of memory devices - to transmit data into a memory device from some other digital electronic device. To WRITE is to STORE.

ZERO-PAGE - The lowest 256 address locations in memory. Where the highest 8-bits of address are always Ø's and the lower 8-bits identify any location from Ø to 255. Therefore, only a single byte is needed to address a location in zero-page.

ZERO-PAGE ADDRESSING - The address portion of the instruction contains a zero-page address.

ZERO-PAGE INDEXED ADDRESSING - The second byte of the instruction is added to an index register (X or Y) to form a zero-page effective address. The carry (if any) is ignored.

SYM TAPE USAGE GUIDE

SYM TAPE USAGE GUIDE

I.D. NUMBER	PROGRAM NAME	MEMORY ADDRESS	START
01	Hunt the Wumpus	0000-03FF	0200
02	Real Time Clock	0200-02BC	0200
03	Black Jack	0200-03EC	0200
04	One Armed Bandit	0200-02DB	0200
05	Lunar Lander	0200-0310	0200
06	Stop Watch	0000-00CB	0000
08	Music Box	0200-028B	0200
09	Tunes for Music Box (Works only with Music Box)	0000-00C0	

HUNT THE WUMPUS

Wumpus lives in a cavern with 16 rooms labeled O-F. The program starts at $0200. You and Wumpus are placed at random in the rooms. Also 2 bottomless pits and two rooms with superbats. If you enter a bat's room you will be picked up and placed at random in another room. You will be warned when bats, pits, or Wumpus are nearby. If you enter the room with the Wumpus, he wakes and may move to an adjacent room or eat you (you lose).

To catch the beast you are given 3 cans of mood change gas. When thrown into a room containing the Wumpus, the gas changes him from a beast to a lovable creature. He'll give you a hug (you win).

However, once you throw the can of gas, the room is contaminated and will turn you into a beast if you enter (lose). Old gas doesn't bother Wumpus.

In addition, whenever you throw the gas, the Wumpus may move. He might even charge if you're in an adjacent room and each you. It's all so sudden you'll never know what hit you. (You lose.)

To throw a can of gas, press "MEM." The display will ask "room"? Enter an adjacent room.

When entering a room with a pit or bat you will be informed of the surrounding hazards before you either fall to your doom or are moved by the bats. Take note of the information as it may be helpful to you.

If you wish to change the speed of the display, change the data in $00E9 and enter the program at $0206.

If you wish to try again leaving everything else the same, enter at $0218. The program uses almost all of Page 0, 2, and 3.

Credit: Gregory Yob from The Best of Creative Computing.

and

Stan Ockers from The First Book of KIM.

```
LINE # LOC     CODE        LINE

       0000                ;HUNT THE WUMPUS GAME
       0000                MSGTAB=$0304
       0000                T1CL=$A004
       0000                ASCNIB=$8275
       0000                POINT=$00FE
       0000                INCCMP=$82B2
       0000                GETSGS=$89EA
       0000                SEGSM1=$8C28
       0000                SCAND=$8906
       0000                RMB=$00E5
       0000                SAVER=$8188
       0000                MESSID=$00E4
       0000                SPEED=$00E9
       0000                RESALL=$81C4
       0000                WPRM=$00F2
       0000                CANS=$00EA
       0000                YOURM=$00F3
       0000                GETKEY=$88AF
       0000                ACCESS=$8B86
       0000                        *=$0200
       0200    A9 10               LDA #$10         ;SET DISPLAY SPEED
       0202    85 E9               STA SPEED
       0204    A9 03               LDA #$03         ;GIVE 3 CANS OF GAS
       0206    85 EA               STA $00EA
       0208    20 86 8B            JSR ACCESS       ;OPEN SYS RAM
       020B    A9 FF               LDA #$FF         ;INITIALIZE ROOMS TO $FF
       020D    A2 0C               LDX #$0C
       020F    95 EB       INIT    STA $00EB,X
       0211    CA                  DEX
       0212    10 FB               BPL INIT
       0214    A0 05               LDY #$05         ;GET RANDOM VARIABLES
       0216    10 02               BPL GETN
       0218    A0 00       BATENT  LDY #$00         ;SUBERBATS ENTRYPOINT
       021A    A2 05       GETN    LDX #$05
       021C    20 51 00            JSR RAND
       021F    29 0F               AND #$0F
       0221    D5 EE       CKNO    CMP $00EE,X      ;MAKE SURE ALL ARE DIFF.
       0223    F0 F5               BEQ GETN
       0225    CA                  DEX
       0226    10 F9               BPL CKNO
       0228    99 EE 00            STA $00EE,Y      ;STORE IN $00EE-$00F3
       022B    88                  DEY
       022C    10 EC               BPL GETN
       022E    20 E1 02    BEGIN   JSR ADJRMS       ; SET UP ADJACENT ROOM LIST
       0231    A0 04       HAZARD  LDY #4           ;CHECK FOR HAZARDS
       0233    A2 07       NEXT1   LDX #7
       0235    B9 F3 00            LDA $00F3,Y
       0238    D5 EB       NEXT    CMP $00EB,X
       023A    D0 03               BNE FINISH
       023C    20 90 02            JSR HZD          ;FOUND HAZARD TAKE ACTION
       023F    CA          FINISH  DEX
       0240    10 F6               BPL NEXT
       0242    88                  DEY
       0243    10 EE               BPL NEXT1
       0245    A9 08       PROMPT  LDA #$08         ;YOU ARE IN ... MESS
```

```
LOC    CODE        LINE

0247   20 00 00            JSR MESS
024A   20 AF 88            JSR GETKEY
024D   C9 4D               CMP #'M'
024F   F0 09               BEQ TRWCAN
0251   20 8A 00            JSR VALID       ;ROOM NUMBER ?
0254   B0 EF               BCS PROMPT
0256   85 F3               STA YOURM       ;UPDATE YOUR ROOM
0258   90 D4               BCC BEGIN
025A   A9 09       TRWCAN  LDA #$09        ;PITCH A CAN
025C   20 00 00            JSR MESS
025F   20 AF 88            JSR GETKEY
0262   20 8A 00            JSR VALID
0265   B0 F3               BCS TRWCAN
0267   A6 EA               LDX CANS
0269   95 EA               STA CANS,X
026B   C6 EA               DEC CANS        ;USED ANOTHER CAN
026D   C5 F2               CMP WPRM        ;GET WUMPUS ?
026F   D0 08               BNE NOWIN       ;NO
0271   A9 0A       WIN     LDA #$0A        ;YES
0273   20 00 00            JSR MESS
0276   18                  CLC
0277   90 F8               BCC WIN
0279   A9 0B       NOWIN   LDA #$0B
027B   20 00 00            JSR MESS
027E   A5 EA               LDA CANS
0280   D0 08               BNE NOLOSE
0282   A9 0C       LOSE    LDA #$0C
0284   20 00 00            JSR MESS
0287   18                  CLC
0288   90 F8               BCC LOSE
028A   20 6D 00    NOLOSE  JSR WPMOVE      ;MOVE THE WUMPUS
028D   18                  CLC             ;CONTINUE GAME
028E   90 9E               BCC BEGIN
0290   20 88 81    HZD     JSR SAVER       ;ACT ON HAZARD
0293   E0 03       COMPES  CPX #3          ;GENERATE HAZARD CODE
0295   B0 02               BCS NO
0297   A2 00               LDX #0          ;GAS
0299   E0 07       NO      CPX #7
029B   90 02               BCC NO1
029D   A2 01               LDX #1          ;WUMPUS
029F   E0 05       NO1     CPX #5
02A1   90 02               BCC NO2
02A3   A2 02               LDX #2          ;BATS
02A5   E0 03       NO2     CPX #3
02A7   90 02               BCC NO3
02A9   A2 03               LDX #3
02AB   C0 00       NO3     CPY #0          ;CHECK IF YOUR ROOM
02AD   D0 04               BNE NO4
02AF   E8                  INX
02B0   E8                  INX
02B1   E8                  INX
02B2   E8                  INX
02B3   8A          NO4     TXA             ;PRINT MESSAGE
02B4   20 00 00            JSR MESS
02B7   A6 E4               LDX MESSID
```

```
LINE # LOC     CODE         LINE

0112  02B9  E0 04              CPX #4          ;GASSED
0113  02BB  D0 02              BNE NOPE
0114  02BD  F0 C3              BEQ LOSE
0115  02BF  E0 07        NOPE  CPX #7          ;PITFALL
0116  02C1  D0 02              BNE NOPE1
0117  02C3  F0 BD              BEQ LOSE
0118  02C5  E0 06        NOPE1 CPX #6          ;SUPERBATS
0119  02C7  D0 0E              BNE NOPE2
0120  02C9  20 51 00           JSR RAND
0121  02CC  29 0F              AND #$0F
0122  02CE  85 F3              STA YOURM
0123  02D0  68           PULL  PLA
0124  02D1  CA                 DEX
0125  02D2  D0 FC              BNE PULL
0126  02D4  4C 18 02           JMP BATENT
0127  02D7  E0 05        NOPE2 CPX #5          ;BUMPED WUMPUS
0128  02D9  D0 03              BNE OUTY
0129  02DB  20 6D 00           JSR WPMOVE
0130  02DE  4C C4 81     OUTY  JMP RESALL
0131  02E1  A5 F3        ADJRMS LDA YOURM
0132  02E3  4A           ADJAC LSR A
0133  02E4  AA                 TAX
0134  02E5  6A                 ROR A
0135  02E6  85 E5              STA RMB
0136  02E8  A0 03              LDY #3
0137  02EA  B5 9C        AGN   LDA TABLE,X
0138  02EC  24 E5              BIT RMB
0139  02EE  10 04              BPL *+6
0140  02F0  29 0F              AND #$0F
0141  02F2  10 04              BPL *+6
0142  02F4  4A                 LSR A
0143  02F5  4A                 LSR A
0144  02F6  4A                 LSR A
0145  02F7  4A                 LSR A
0146  02F8  99 F4 00           STA $00F4,Y
0147  02FB  8A                 TXA
0148  02FC  18                 CLC
0149  02FD  69 08              ADC #$08
0150  02FF  AA                 TAX
0151  0300  88                 DEY
0152  0301  10 E7              BPL AGN
0153  0303  60                 RTS
0154  0304               ;MESSAGE TABLE
0155  0304  80                 .BYT $80,$00
0155  0305  00
0156  0306               ;WUMPUS CLOSE MESSAGE
0157  0306  80                 .BYT $80,$01,$00,$3E,$1C,$37,$73,$1C,$6D
0157  0307  01
0157  0308  00
0157  0309  3E
0157  030A  1C
0157  030B  37
0157  030C  73
0157  030D  1C
0157  030E  6D
```

```
LINE # LOC      CODE         LINE

0158   030F  00                      .BYT $00,$39,$38,$3F,$6D,$79,$00
0158   0310  39
0158   0311  38
0158   0312  3F
0158   0313  6D
0158   0314  79
0158   0315  00
0159   0316                    ; BATS CLOSE MESSAGE
0160   0316  80                      .BYT $80,$02,$00,$7C,$77,$78,$6D,$00,$39
0160   0317  02
0160   0318  00
0160   0319  7C
0160   031A  77
0160   031B  78
0160   031C  6D
0160   031D  00
0160   031E  39
0161   031F  38                      .BYT $38,$3F,$6D,$79,$00
0161   0320  3F
0161   0321  6D
0161   0322  79
0161   0323  00
0162   0324                    ;PITS CLOSE MESSAGE
0163   0324  80                      .BYT $80,$03,$00,$73,$06,$78,$6D,$00,$39
0163   0325  03
 .63   0326  00
0163   0327  73
0163   0328  06
0163   0329  78
0163   032A  6D
0163   032B  00
016    032C  39
0164   032D  38                      .BYT $38,$3F,$6D,$79,$00
0164   032E  3F
0164   032F  6D
0164   0330  79
0164   0331  00
0165   0332                    ;GASSED MESSAGE
0166   0332  80                      .BYT $80,$04,$00,$3D,$77,$6D,$6D,$79,$5E
0166   0333  04
0166   0334  00
0166   0335  3D
0166   0336  77
0166   0337  6D
0166   0338  6D
0166   0339  79
0166   033A  5E
0167   033B  00                      .BYT $00
0168   033C                    ;BUMPED WUMPUS MESSAGE
0169   033C  80                      .BYT $80,$05,$00,$3F,$3F,$73,$6D,$00,$7C
  59   033D  05
0169   033E  00
0169   033F  3F
0169   0340  3F
0169   0341  73
```

```
LINE # LOC     CODE      LINE

0169  0342  6D
0169  0343  00
0169  0344  7C
0170  0345  1C                .BYT $1C,$37,$73,$79,$5E,$00,$77,$00
0170  0346  37
0170  0347  73
0170  0348  79
0170  0349  5E
0170  034A  00
0170  034B  77
0170  034C  00
0171  034D  3E                .BYT $3E,$1C,$37,$73,$1C,$6D,$00
0171  034E  1C
0171  034F  37
0171  0350  73
0171  0351  1C
0171  0352  6D
0171  0353  00
0172  0354             ;SUPERBATS SNATCHED YOU MESSAGE
0173  0354  80                .BYT $80,$06,$00,$6D,$1C,$73,$79,$50,$7C
0173  0355  06
0173  0356  00
0173  0357  6D
0173  0358  1C
0173  0359  73
0173  035A  79
0173  035B  50
0173  035C  7C
0174  035D  77                .BYT $77,$78,$6D,$00,$6D,$54,$77,$78
0174  035E  78
0174  035F  6D
0174  0360  00
0174  0361  6D
0174  0362  54
0174  0363  77
0174  0364  78
0175  0365  58                .BYT $58,$76,$79,$5E,$00,$6E,$3F,$1C
0175  0366  76
0175  0367  79
0175  0368  5E
0175  0369  00
0175  036A  6E
0175  036B  3F
0175  036C  1C
0176  036D  00                .BYT $00
0177  036E             ;FELL IN PIT MESSAGE
0178  036E  80                .BYT $80,$07,$00,$6E,$04,$79,$79,$79,$00
0178  036F  07
0178  0370  00
0178  0371  6E
0178  0372  04
0178  0373  79
0178  0374  79
0178  0375  79
0178  0376  00
```

```
LINE # LOC     CODE       LINE

0179  0377  71                .BYT $71,$79,$38,$38,$00,$04,$54,$00
0179  0378  79
0179  0379  38
0179  037A  38
0179  037B  00
0179  037C  04
0179  037D  54
0179  037E  00
0180  037F  73                .BYT $73,$04,$78,$00
0180  0380  04
0180  0381  78
0180  0382  00
0181  0383               ;YOU ARE IN ... MESSAGE
0182  0383  80                .BYT $80,$08,$00,$6E,$3F,$1C,$00,$77,$50
0182  0384  08
0182  0385  00
0182  0386  6E
0182  0387  3F
0182  0388  1C
0182  0389  00
0182  038A  77
0182  038B  50
0183  038C  79                .BYT $79,$00,$04,$54,$00,$CB,$00,$78
0183  038D  00
0183  038E  04
0183  038F  54
0183  0390  00
0183  0391  CB
0183  0392  00
0183  0393  78
0184  0394  1C                .BYT $1C,$54,$54,$79,$38,$6D,$00,$38
0184  0395  54
0184  0396  54
0184  0397  79
0184  0398  38
0184  0399  6D
0184  039A  00
0184  039B  38
0185  039C  79                .BYT $79,$77,$5E,$00,$78,$3F,$00
0185  039D  77
0185  039E  5E
0185  039F  00
0185  03A0  78
0185  03A1  3F
0185  03A2  00
0186  03A3  CC                .BYT $CC,$CD,$CE,$CF,$00
0186  03A4  CD
0186  03A5  CE
0186  03A6  CF
0186  03A7  00
0187  03A8               ;ROOM ? MESSAGE
0188  03A8  80                .BYT $80,$09,$00,$50,$3F,$3F,$37,$53
0188  03A9  09
0188  03AA  00
0188  03AB  50
```

```
LINE # LOC      CODE        LINE

 188  03AC  3F
 188  03AD  3F
0188  03AE  37
0188  03AF  53
0189  03B0                 ;YOU WIN MESSAGE
0190  03B0  80                     .BYT $80,$0A,$00,$3D,$50,$79,$77,$78,$00
0190  03B1  0A
0190  03B2  00
0190  03B3  3D
0190  03B4  50
0190  03B5  79
0190  03B6  77
0190  03B7  78
0190  03B8  00
0191  03B9  6E                     .BYT $6E,$3F,$1C,$00,$3D,$79,$78,$00
0191  03BA  3F
0191  03BB  1C
0191  03BC  00
0191  03BD  3D
0191  03BE  79
0191  03BF  78
0191  03C0  00
0192  03C1  77                     .BYT $77,$00,$76,$1C,$3D,$00,$71,$50
0192  03C2  00
0192  03C3  76
 192  03C4  1C
 192  03C5  3D
0192  03C6  00
0192  03C7  71
0192  03C8  50
0193  03C9  3F                     .BYT $3F,$37,$00,$3E,$1C,$37,$73,$1C
0193  03CA  37
0193  03CB  00
0193  03CC  3E
0193  03CD  1C
0193  03CE  37
0193  03CF  73
0193  03D0  1C
0194  03D1  6D                     .BYT $6D,$00
0194  03D2  00
0195  03D3                 ;? CANS LEFT MESSAGE
0196  03D3  80                     .BYT $80,$0B,$00,$37,$04,$6D,$6D,$79,$5E
0196  03D4  0B
0196  03D5  00
0196  03D6  37
0196  03D7  04
0196  03D8  6D
0196  03D9  6D
0196  03DA  79
0196  03DB  5E
 197  03DC  00                     .BYT $00,$3F,$54,$38,$6E,$00,$C2,$00
  97  03DD  3F
0197  03DE  54
0197  03DF  38
0197  03E0  6E
```

```
LINE # LOC      CODE          LINE

0197   03E1   00
0197   03E2   C2
0197   03E3   00
0198   03E4   39                          .BYT $39,$77,$54,$6D,$00,$38,$79,$71
0198   03E5   77
0198   03E6   54
0198   03E7   6D
0198   03E8   00
0198   03E9   38
0198   03EA   79
0198   03EB   71
0199   03EC   78                          .BYT $78,$00
0199   03ED   00
0200   03EE                        ;YOU LOSE MESSAGE
0201   03EE   80                          .BYT $80,$0C,$6E,$3F,$1C,$00,$38,$3F
0201   03EF   0C
0201   03F0   6E
0201   03F1   3F
0201   03F2   1C
0201   03F3   00
0201   03F4   38
0201   03F5   3F
0202   03F6   6D                          .BYT $6D,$79,$00
0202   03F7   79
0202   03F8   00
0203   03F9   80                          .BYT $80


0205   03FA                               *=$0000


0207   0000                        ;MESSAGE DISPLAY ROUTINE

0209   0000   85 E4            MESS       STA MESSID
0210   0002   A9 03                       LDA #,MSGTAB-1
0211   0004   85 FE                       STA $00FE
0212   0006   A9 03                       LDA #.MSGTAB
0213   0008   85 FF                       STA $00FF
0214   000A   20 47 00         BACK       JSR GTNXT
0215   000D   D0 FB                       BNE BACK
0216   000F   20 47 00                    JSR GTNXT
0217   0012   C5 E4                       CMP MESSID
0218   0014   D0 F4                       BNE BACK
0219   0016   20 47 00         BACK1      JSR GTNXT
0220   0019   D0 01                       BNE OVR3
0221   001B   60                          RTS
0222   001C   90 09            OVR3       BCC OUT4
0223   001E   29 3F                       AND #$3F
0224   0020   AA                          TAX
0225   0021   B5 E8                       LDA $00E8,X
0226   0023   AA                          TAX
0227   0024   BD 29 8C                    LDA SEGSM1+1,X
0228   0027   20 41 00         OUT4       JSR OUTVAR
```

```
LINE # LOC      CODE          LINE

0229   002A   A4 E9                   LDY SPEED
0230   002C   A2 20          OUTDLY  LDX #$20
0231   002E   8A             INDELY  TXA
0232   002F   48                      PHA
0233   0030   98                      TYA
0234   0031   48                      PHA
0235   0032   20 06 89                JSR SCAND
0236   0035   68                      PLA
0237   0036   A8                      TAY
0238   0037   68                      PLA
0239   0038   AA                      TAX
0240   0039   CA                      DEX
0241   003A   D0 F2                   BNE INDELY
0242   003C   88                      DEY
0243   003D   D0 ED                   BNE OUTDLY
0244   003F   F0 D5                   BEQ BACK1
0245   0041   20 88 81       OUTVAR  JSR SAVER
0246   0044   4C F1 89                JMP GETSGS+7

0248   0047   20 B2 82       GTNXT   JSR INCCMP
0249   004A   A0 00                   LDY #0
0250   004C   B1 FE                   LDA (POINT),Y
0251   004E   C9 80                   CMP #$80
0252   0050   60                      RTS


0254   0051                  ;GET A RANDOM NUMBER

0256   0051   8A             RAND    TXA
0257   0052   48                      PHA
0258   0053   D8                      CLD
0259   0054   38                      SEC
0260   0055   A5 DF                   LDA $00DF
0261   0057   65 E1                   ADC $00E1
0262   0059   65 E2                   ADC $00E2
0263   005B   85 DE                   STA $00DE
0264   005D   A2 04                   LDX #$04
0265   005F   B5 DE          NXTN    LDA $00DE,X
0266   0061   95 DF                   STA $00DF,X
0267   0063   CA                      DEX
0268   0064   10 F9                   BPL NXTN
0269   0066   85 DD                   STA $00DD
0270   0068   68                      PLA
0271   0069   AA                      TAX
0272   006A   A5 DD                   LDA $00DD
0273   006C   60                      RTS


0275   006D                  ;MOVE THE WUMPUS

 277   006D   A5 F2          WPMOVE  LDA WPRM
0278   006F   20 E3 02                JSR ADJAC
0279   0072   AD 04 A0                LDA T1CL
```

```
LINE # LOC     CODE          LINE

0280   0075   29 07                 AND #$07
0281   0077   4A                    LSR A
0282   0078   69 00                 ADC #0
0283   007A   AA                    TAX
0284   007B   B5 F3                 LDA $00F3,X
0285   007D   85 F2                 STA WPRM
0286   007F   C5 F3                 CMP YOURM
0287   0081   D0 03                 BNE OPS
0288   0083   20 82 02              JSR LOSE
0289   0086   20 E1 02     OPS      JSR ADJRMS
0290   0089   60                    RTS


0292   008A                ;RETUN BINARY VALIDITY IN CARRY
0293   008A                ;CHECK TO SEE IF KEY WAS VALID

0295   008A   20 75 02     VALID    JSR ASCNIB
0296   008D   B0 0B                 BCS OUTZ
0297   008F   A2 03                 LDX #3
0298   0091   D5 F4        NX       CMP $00F4,X
0299   0093   D0 02                 BNE OVR1
0300   0095   18                    CLC
0301   0096   60                    RTS
0302   0097   CA           OVR1     DEX
0303   0098   10 F7                 BPL NX
0304   009A   38           OUTZ     SEC
0305   009B   60                    RTS


0307   009C                ;TABLE OF ADJACENT ROOMS USED BY
0308   009C                ;ADJRMS AND ADJAC

0310   009C                TABLE=*
0311   009C   22                    .BYT $22,$01,$10,$34,$06,$70,$9A,$14
0311   009D   01
0311   009E   10
0311   009F   34
0311   00A0   06
0311   00A1   70
0311   00A2   9A
0311   00A3   14
0312   00A4   53                    .BYT $53,$12,$32,$56,$58,$98,$BC,$B7
0312   00A5   12
0312   00A6   32
0312   00A7   56
0312   00A8   58
0312   00A9   98
0312   00AA   BC
0312   00AB   B7
0313   00AC   84                    .BYT $84,$34,$76,$7A,$9A,$FC,$DE,$CA
0313   00AD   34
0313   00AE   76
0313   00AF   7A
0313   00B0   9A
```

```
LINE # LOC      CODE        LINE

0313  00B1  FC
0313  00B2  DE
0313  00B3  CA
0314  00B4  BE                    .BYT $BE,$56,$F8,$9F,$BC,$DE,$EF,$DD
0314  00B5  56
0314  00B6  F8
0314  00B7  9F
0314  00B8  BC
0314  00B9  DE
0314  00BA  EF
0314  00BB  DD
0315  00BC                        .END

ERRORS = 0000 ,0000.
```

```
SYMBOL TABLE

SYMBOL   VALUE

ACCESS   8B86   ADJAC    02E3   ADJRMS   02E1   AGN      02EA
ASCNIB   8275   BACK     000A   BACK1    0016   BATENT   0218
BEGIN    022E   CANS     00EA   CKNO     0221   COMPES   0293
FINISH   023F   GETKEY   88AF   GETN     021A   GETSGS   89EA
GTNXT    0047   HAZARD   0231   HZD      0290   INCCMP   82B2
INDELY   002E   INIT     020F   LOSE     0282   MESS     0000
MESSID   00E4   MSGTAB   0304   NEXT     0238   NEXT1    0233
NO       0299   NO1      029F   NO2      02A5   NO3      02AB
NO4      02B3   NOLOSE   028A   NOPE     02BF   NOPE1    02C5
NOPE2    02D7   NOWIN    0279   NX       0091   NXTN     005F
OPS      0086   OUT4     0027   OUTDLY   002C   OUTVAR   0041
OUTY     02DE   OUTZ     009A   OVR1     0097   OVR3     001C
POINT    00FE   PROMPT   0245   PULL     02D0   RAND     0051
RESALL   81C4   RMB      00E5   SAVER    8188   SCAND    8906
SEGSM1   8C28   SPEED    00E9   T1CL     A004   TABLE    009C
TRWCAN   025A   VALID    008A   WIN      0271   WPMOVE   006D
WPRM     00F2   YOURM    00F3
END OF ASSEMBLY
```

Real Time Clock

by Gene Zumchak

The Real Time Clock program on your demo tape is an excellent example of an application taking advantage of Timer 1 in the 6522 VIA. For the RTC, Timer 1 is used in the free running mode, and enabled to give an interrupt (IRQ). In this mode, the timer is automatically reloaded with the sixteen bit number held in the latches as soon as timeout occurs, and independently of any response from the 6502. Consequently, the timeout is consistently accurate to the microsecond and contains no "slop" for interrupt response. The maximum period of the counter is 2 to the sixteenth power, or 65,536 microseconds. The largest convenient fraction of a second that we can use is 50,000 microseconds (50 milliseconds) or one-twentieth of a second. Thus we can preload the Timer 1 latches to give us an interrupt every 1/20 of a second. We then increment seconds every twenty interrupts, increment minutes every 60 seconds, and increment hours every sixty minutes. We could extend the program to keep track of days and months as well, which might be useful in a controller application where we had to log the time and date when some event occurred.

Using the Clock The program loads in with ID #02. Start the program at 200 and see hours, minutes, and seconds displayed. All are initially zero. To set the time, enter the minutes with the keyboard and then press the "A" on the keyboard. The moment "A" is pressed, the entered minutes will be displayed and the seconds zeroed. Thus seconds can be synchronized with another clock by waiting for a precise time to press "A". To set the hours, enter the number with the keyboard and then press "B". Let your clock run overnight and see if it looses or gains any seconds. If your 1 MHz crystal clock were off by one part in a million, which is a lot better than we can expect, we would gain or lose one microsecond per second. Since it's probably worse than that, we can expect to lose or gain a few seconds a day. This points out how incredably accurate the crystals for electronic watches must be ground. We can fine tune our clock by changing the value in location $0206 which is intially $4E. This is the number loaded into the lower eight bits of Timer one. Changing this by just one will add or subtract 20 microseconds per second or about 1.7 million microseconds per day. Of course, that'sr almost 2 seconds.

BLACKJACK, BANDIT, and LANDER

by Jim Butterfield
adapted for SYM
by Gene Zumchak

Blackjack, One Armed Bandit, and Lunar Lander are three of the most popular games from the First Book of KIM, all of them authored by Jim Butterfield. It is not simply a matter of changing a few subroutine addresses to convert KIM programs, which used the KIM keyboard and display into SYM programs. Well commented listings for all three programs can be found in the "First Book of KIM". By comparing the listings given to the original KIM listings, it should be possible to figure out how to translate other programs.

BLACKJACK

Blackjack loads with ID #03 and starts at $0200. The SYM uses a real deck, and shuffles when the deck gets low. You will have twenty dollars when you start the game. After a brief "SHUFFL" message, the SYM prompts with "bEt?20. You may now bet from 1 to 9 dollars of your money by pushing the appropriate numbered keys. (SYM will not allow you to ever bet more than you have left.) After indicating your bet, you'll be given two cards, and you'll see one of the houses cards. Face cards and tens are worth ten and are shown as a "F" on the display. Press"0" when you want no more cards. To get a third card press "3", press "4" for a fourth etc. Your total is displayed when you press "0". Aces (A) count one or eleven. You automatically win if you take five cards without going over.

BANDIT

Start the program at $0200. (It loads with ID #04.) Hit any key and start the wheels spinning. You'll start with $25 dollars and risk $1 with each pull. The biggest jackpot, three bars across pays $15. Other combinations pay off. A "cherry" pays $2 whenever it appear is the left had window.

Lunar Lander

Lander loads with ID #05. When the program is started at $0200, you will find yourself at 4,500 feet and falling. You'll pick up spped due to the force of gravity. You have 800 pounds of fuel to start. You can see how much is left by pushing "F". Pushing "A" restores altitude to the display. Set your thrust by using buttons "1" thru "9". (Careful, "0" shuts the engine down.) Minimum thrust, "1" uses the least fuel, but you'll continue to accellerate. A thrust of "5" will counterbalance gravity and you'll fall at constant speed. A thrust of "9" overcomes gravity and you'll decellerate quickly. It also uses up fuel fastest. Too much thrust, and you'll slow to zero and then start climbing. A safe landing is a descent rate of 05 or less. Note that the rate is always displayed in the right two digits. After you land, you can press "F" to see how much fuel you have left. Press Carriage Return to get another flight. Several strategies are possible. You might try to land safely and have as much fuel as possible left. You might try to land safely with as little fuel as possible. When you run out of fuel you'll start falling like a rock. You might see how high you can go and still land safely.

```
0010:
0020: 0200              CLOCK  ORG    $0200
0030:
0040:                    REAL TIME CLOCK
0050:                    BY GENE ZUMCHAK
0060:                        3/11/79
0070:
0080:                     COPYWRITE 1979
0090:                       E. ZUMCHAK
0100:
0110:
0120:
0130: 0200              HR      *     $0000
0140: 0200              MIN     *     $0001
0150: 0200              SEC     *     $0002
0160: 0200              EX      *     $0004
0170: 0200              NUM     *     $0005
0180: 0200              ACCESS  *     $8B86
0190: 0200              UIRQVL  *     $A678
0200: 0200              UIRQVH  *     $A679
0210: 0200              IER     *     $A00E
0220: 0200              ACR     *     $A00B
0230: 0200              TONELL  *     $A004
0240: 0200              TONELH  *     $A007
0250: 0200              TONECH  *     $A005
0260: 0200              TONECL  *     $A004
0270: 0200              SCAND   *     $8906
0280: 0200              SEGS    *     $8C29
0290: 0200              DISBUF  *     $A640
0300: 0200              GETKEY  *     $88AF
0310:
0320: 0200 A9 40                LDAIM $40     INITIALIZE TIMER 1 IN 6522
0330: 0202 8D 0B A0             STA   ACR
0340: 0205 A9 4E                LDAIM $4E     LOW EIGHT BITS (FINE TUNE)
0350: 0207 8D 04 A0             STA   TONELL
0360: 020A A9 C3                LDAIM $C3     HIGH EIGHT BITS
0370: 020C 8D 05 A0             STA   TONECH
0380: 020F 8D 07 A0             STA   TONELH
0390: 0212 A9 C0                LDAIM $C0
0400: 0214 8D 0E A0             STA   IER
0410: 0217 58                   CLI
0420: 0218 20 86 8B             JSR   ACCESS
0430: 021B A9 02                LDAIM IRQ
0440: 021D 8D 79 A6             STA   UIRQVH
0450: 0220 A9 95                LDAIM IRQ
0460: 0222 8D 78 A6             STA   UIRQVL
0470: 0225 A9 00                LDAIM $00     INITIALIZE SEC,MIN,HOURS RAM
0480: 0227 A2 03                LDXIM $03
0490: 0229 95 00        L       STAZX $00
0500: 022B CA                   DEX
0510: 022C 10 FB                BPL   L
0520: 022E 20 06 89     MAIN    JSR   SCAND   MAIN PROGRAM BEGINS HERE
0530: 0231 F0 13                BEQ   DIS     GO TO DIS IF NO KEY PRESSED
0540: 0233 20 AF 88             JSR   GETKEY  OTHERWISE GET KEY VALUE
0550: 0236 C9 3A                CMPIM $3A     NUMBER KEY?
0560: 0238 B0 0E                BCS   LETTER
```

```
0570: 023A 0A               ASLA
0580: 023B 0A               ASLA
0590: 023C 0A               ASLA
0600: 023D 0A               ASLA
0610: 023E A2 04            LDXIM $04
0620: 0240 2A        ROLL   ROLA
0630: 0241 26 05            ROL   NUM
0640: 0243 CA               DEX
0650: 0244 D0 FA            BNE   ROLL
0660: 0246 F0 17     DIS    BEQ   DISPLY
0670: 0248 C9 41     LETTER CMPIM $41     LETTER A SETS MINUTES
0680: 024A D0 0B            BNE   HOUR      CLEARS SECONDS
0690: 024C A9 00            LDAIM $00
0700: 024E 85 02            STA   SEC
0710: 0250 A5 05            LDA   NUM
0720: 0252 85 01            STA   MIN
0730: 0254 4C 5F 02         JMP   DISPLY
0740: 0257 C9 42     HOUR   CMPIM $42     LETTER B SETS HOURS
0750: 0259 D0 04            BNE   DISPLY
0760: 025B A5 05            LDA   NUM
0770: 025D 85 00            STA   HR
0780: 025F A0 00     DISPLY LDYIM $00     DISPLAY CONVERTS HOURS, MINUTE
0790: 0261 84 04            STY   EX      SECONDS INTO SEVEN SEGMENT PAT
0800: 0263 A6 04     DL     LDX   EX      LOADS PATTERNS INTO THE DISPLA
0810: 0265 B5 00            LDAZX HR
0820: 0267 48               PHA
0830: 0268 4A               LSRA
0840: 0269 4A               LSRA
0850: 026A 4A               LSRA
0860: 026B 4A               LSRA
0870: 026C AA               TAX
0880: 026D BD 29 8C         LDAAX SEGS
0890: 0270 99 40 A6         STAAY DISBUF
0900: 0273 C8               INY
0910: 0274 68               PLA
0920: 0275 29 0F            ANDIM $0F
0930: 0277 AA               TAX
0940: 0278 BD 29 8C         LDAAX SEGS
0950: 027B 99 40 A6         STAAY DISBUF
0960: 027E E6 04            INC   EX
0970: 0280 C8               INY
0980: 0281 C0 06            CPYIM $06
0990: 0283 D0 DE            BNE   DL
1000: 0285 A2 03            LDXIM $03
1010: 0287 BD 40 A6  LL     LDAAX DISBUF  PERIOD SEPARATORS ADDED HERE
1020: 028A 09 80            ORAIM $80
1030: 028C 9D 40 A6         STAAX DISBUF
1040: 028F CA               DEX
1050: 0290 CA               DEX
1060: 0291 10 F4            BPL   LL
1070: 0293 30 99            BMI   MAIN
1080:
1090: 0295 48        IRQ    PHA           INTERRUPT PROGRAM BEGINS HERE
1100: 0296 8A               TXA           SAVE A AND X REGISTERS
1110: 0297 48               PHA
1120: 0298 F8               SED
```

```
1130: 0299 AD 04 A0           LDA    TONECL   THIS OPERATION CLEARS INTERRUPT
1140: 029C A2 03              LDXIM $03
1150: 029E B5 00        ADD   LDAZX HR         TIME VARIABLES ARE INCREMENTED I
1160: 02A0 18                 CLC             A LOOP.  NEXT VARIABLE IS INCREME
1170: 02A1 69 01              ADCIM $01        ONLY IF TERMINAL VALUE FROM
1180: 02A3 95 00              STAZX HR         TABLE OCCURS
1190: 02A5 DD B9 02           CMPAX TABLE
1200: 02A8 90 0B              BCC   RET
1210: 02AA A9 00              LDAIM $00
1220: 02AC 95 00              STAZX HR
1230: 02AE CA                 DEX
1240: 02AF 10 ED              BPL   ADD
1250: 02B1 A9 01              LDAIM $01
1260: 02B3 85 00              STA   HR
1270: 02B5 68           RET   PLA
1280: 02B6 AA                 TAX
1290: 02B7 68                 PLA
1300: 02B8 40                 RTI
1310:
1320: 02B9 13           TABLE =     $13
1330: 02BA 60                 =     $60
1340: 02BB 60                 =     $60
1350: 02BC 20                 =     $20
ID=
0010: 0200               BLAKJK ORG    $0200
0020:
0030:                           BLACKJACK
0040:                    BY JIM BUTTERFIELD
0050:
0060:                     ADAPTED FOR SYM
0070:                     BY GENE ZUMCHAK
0080:                          3/11/79
0081:
0090:                      COPYWRITE 1979
0100:                        E. ZUMCHAK
0110:
0120:
0130:
0140: 0200               ONE     *     $003E
0150: 0200               TWO     *     $003F
0160: 0200               DECK    *     $0040
0170: 0200               BET     *     $0079
0180: 0200               PARAM   *     $0075
0190: 0200               DPT     *     $0076
0200: 0200               AMT     *     $0077
0210: 0200               HOLE    *     $007A
0220: 0200               PAUSE   *     $007B
0230: 0200               UCNT    *     $0096
0240: 0200               MTOT    *     $009A
0250: 0200               MACE    *     $009B
0260: 0200               UACE    *     $0098
0270: 0200               UTOT    *     $0097
0280: 0200               YSAV    *     $007F
0290: 0200               RND     *     $0080
0300: 0200               POINTR  *     $0074
0310: 0200               WINDOW  *     $A640
0320: 0200               MCNT    *     $0099
0330:                    SYM REFERENCES
0340: 0200               TIMER   *     $A414
0350: 0200               SCAND   *     $8906
0360: 0200               GETKEY  *     $88AF
0370: 0200               TABLE   *     $8C29
0380: 0200               ACCESS  *     $8B86
0390:
0400: 0200 20 86 8B      START   JSR   ACCESS
0410: 0203 A2 33                 LDXIM $33
0420: 0205 8A            DKONE   TXA
0430: 0206 95 40                 STAZX DECK
0440: 0208 CA                    DEX
0450: 0209 10 FA                 BPL   DKONE
0460: 020B A2 02                 LDXIM $02
0470: 020D BD BC 03      INLOP   LDAAX INIT
0480: 0210 95 75                 STAZX PARAM
0490: 0212 CA                    DEX
0500: 0213 10 F8                 BPL   INLOP
0510: 0215 AD 14 A4              LDA   TIMER
0520: 0218 85 80                 STA   RND
0530: 021A D8            DEAL    CLD
0540: 021B A6 76                 LDX   DPT
0550: 021D E0 09                 CPXIM $09
```

```
0560: 021F B0 3A              BCS   NOSHUF
0570:                  SHUFFLE DECK
0580: 0221 A0 D9              LDYIM SHUF
0590: 0223 20 4E 03           JSR   FILL
0600: 0226 A0 33              LDYIM $33
0610: 0228 84 76              STY   DPT
0620: 022A A9 0A       SHLP   LDAIM $0A
0630: 022C 85 3E              STA   ONE
0640: 022E A9 01              LDAIM $01
0650: 0230 20 37 03           JSR   WLITE +02
0660: 0233 38                 SEC
0670: 0234 A5 81              LDA   RND    +01
0680: 0236 65 82              ADC   RND    +02
0690: 0238 65 85              ADC   RND    +05
0700: 023A 85 80              STA   RND
0710: 023C A2 04              LDXIM $04
0720: 023E B5 80       RMOV   LDAZX RND
0730: 0240 95 81              STAZX RND    +01
0740: 0242 CA                 DEX
0750: 0243 10 F9              BPL   RMOV
0760: 0245 29 3F              ANDIM $3F
0770: 0247 C9 34              CMPIM $34
0780: 0249 B0 DF              BCS   SHLP
0790: 024B AA                 TAX
0800: 024C B9 40 00           LDAAY DECK
0810: 024F 48                 PHA
0820: 0250 B5 40              LDAZX DECK
0830: 0252 99 40 00           STAAY DECK
0840: 0255 68                 PLA
0850: 0256 95 40              STAZX DECK
0860: 0258 88                 DEY
0870: 0259 10 CF              BPL   SHLP
0880: 025B A0 DF       NOSHUF LDYIM BETQ
0890: 025D 20 4E 03           JSR   FILL
0900: 0260 A5 77              LDA   AMT
0910: 0262 20 A5 03           JSR   NUMDIS
0920: 0265 20 AF 88    BETIN  JSR   GETKEY
0930: 0268 29 0F              ANDIM $0F
0940: 026A AA                 TAX
0950: 026B 86 79              STX   BET
0960: 026D CA                 DEX
0970: 026E 30 F5              BMI   BETIN
0980: 0270 E4 77              CPX   AMT
0990: 0272 B0 F1              BCS   BETIN
1000: 0274 A2 0B              LDXIM $0B
1010: 0276 A9 00              LDAIM $00
1020: 0278 9D 40 A6    CLOOP  STAAX WINDOW
1030: 027B 95 90              STAZX $90
1040: 027D CA                 DEX
1050: 027E 10 F8              BPL   CLOOP
1060: 0280 20 6F 03           JSR   YOU
1070: 0283 20 8A 03           JSR   ME
1080: 0286 20 6F 03           JSR   YOU
1090: 0289 20 5B 03           JSR   CARD
1100: 028C 86 7A              STX   HOLE
1110: 028E 20 AF 88    TRY    JSR   GETKEY
```

```
1120: 0291 29 0F             ANDIM $0F
1130: 0293 AA                TAX
1140: 0294 CA                DEX
1150: 0295 30 11             BMI   HOLD
1160: 0297 E4 96             CPX   UCNT
1170: 0299 D0 F3             BNE   TRY
1180: 029B 20 6F 03          JSR   YOU
1190: 029E C9 22             CMPIM $22
1200: 02A0 B0 44             BCS   UBUST
1210: 02A2 E0 05             CPXIM $05
1220: 02A4 F0 57             BEQ   UWIN
1230: 02A6 D0 E6             BNE   TRY
1240: 02A8 AD 45 A6   HOLD   LDA   WINDOW +05
1250: 02AB 48                PHA
1260: 02AC A2 00             LDXIM $00
1270: 02AE 20 1C 03          JSR   SHTOT
1280: 02B1 A2 04             LDXIM $04
1290: 02B3 A9 00             LDAIM $00
1300: 02B5 9D 40 A6   HLOOP  STAAX WINDOW
1310: 02B8 CA                DEX
1320: 02B9 10 FA             BPL   HLOOP
1330: 02BB 68                PLA
1340: 02BC 8D 45 A6          STA   WINDOW +05
1350: 02BF A6 7A             LDX   HOLE
1360: 02C1 20 64 03          JSR   CREC
1370: 02C4 20 8D 03          JSR   MEX
1380: 02C7 20 35 03   PLAY   JSR   WLITE
1390: 02CA A5 9A             LDA   MTOT
1400: 02CC C9 22             CMPIM $22
1410: 02CE B0 2A             BCS   IBUST
1420: 02D0 65 9B             ADC   MACE
1430: 02D2 AE 41 A6          LDX   WINDOW +01
1440: 02D5 D0 18             BNE   IWIN
1450: 02D7 C9 22             CMPIM $22
1460: 02D9 90 02             BCC   POV
1470: 02DB A5 9A             LDA   MTOT
1480: 02DD C9 17      POV    CMPIM $17
1490: 02DF B0 2C             BCS   HOLDTU
1500: 02E1 20 8A 03          JSR   ME
1510: 02E4 D0 E1             BNE   PLAY
1520: 02E6 20 35 03   UBUST  JSR   WLITE
1530: 02E9 20 4C 03          JSR   BUST
1540: 02EC 20 35 03          JSR   WLITE
1550: 02EF A5 77      IWIN   LDA   AMT
1560: 02F1 F8                SED
1570: 02F2 38                SEC
1580: 02F3 E5 79             SBC   BET
1590: 02F5 85 77      JLINK  STA   AMT
1600: 02F7 4C 1A 02   XLINK  JMP   DEAL
1610: 02FA 20 4C 03   IBUST  JSR   BUST
1620: 02FD 20 35 03   UWIN   JSR   WLITE
1630: 0300 A5 77      ADD    LDA   AMT
1640: 0302 F8                SED
1650: 0303 18                CLC
1660: 0304 65 79             ADC   BET
1670: 0306 A0 99             LDYIM $99
```

```
1680: 0308 90 01               BCC   NOFLO
1690: 030A 98                  TYA
1700: 030B D0 E8        NOFLO  BNE   JLINK
1710: 030D A2 03        HOLDTU LDXIM $03
1720: 030F 20 1C 03            JSR   SHTOT
1730: 0312 A5 9A               LDA   MTOT
1740: 0314 C5 97               CMP   UTOT
1750: 0316 F0 DF               BEQ   XLINK
1760: 0318 B0 D5               BCS   IWIN
1770: 031A 90 E4               BCC   ADD
1780: 031C B5 97        SHTOT  LDAZX UTOT
1790: 031E F8                  SED
1800: 031F 18                  CLC
1810: 0320 75 98               ADCZX UACE
1820: 0322 C9 22               CMPIM $22
1830: 0324 B0 02               BCS   SHOVER
1840: 0326 95 97               STAZX UTOT
1850: 0328 D8           SHOVER CLD
1860: 0329 B5 97               LDAZX UTOT
1870: 032B 48                  PHA
1880: 032C A0 E3               LDYIM TOTL
1890: 032E 20 4E 03            JSR   FILL
1900: 0331 68                  PLA
1910: 0332 20 A5 03            JSR   NUMDIS
1920: 0335 A9 0A        WLITE  LDAIM $0A
1930: 0337 85 3F               STA   TWO
1940: 0339 20 44 03     WDA    JSR   LIGHT
1950: 033C C6 3E               DEC   ONE
1960: 033E D0 F9               BNE   WDA
1970: 0340 C6 3F               DEC   TWO
1980: 0342 D0 F5               BNE   WDA
1990: 0344 84 7F        LIGHT  STY   YSAV
2000: 0346 20 06 89            JSR   SCAND
2010: 0349 A4 7F        RET    LDY   YSAV
2020: 034B 60                  RTS
2030: 034C A0 E7        BUST   LDYIM BUSTED
2040: 034E 84 74        FILL   STY   POINTR
2050: 0350 A0 05               LDYIM $05
2060: 0352 B1 74        FILLIT LDAIY POINTR
2070: 0354 99 40 A6            STAAY WINDOW
2080: 0357 88                  DEY
2090: 0358 10 F8               BPL   FILLIT
2100: 035A 60                  RTS
2110: 035B A6 76        CARD   LDX   DPT
2120: 035D C6 76               DEC   DPT
2130: 035F B5 40               LDAZX DECK
2140: 0361 4A                  LSRA
2150: 0362 4A                  LSRA
2160: 0363 AA                  TAX
2170: 0364 18           CREC   CLC
2180: 0365 D0 01               BNE   NOTACE
2190: 0367 38                  SEC
2200: 0368 BD BF 03     NOTACE LDAAX VALUE
2210: 036B BC CC 03            LDYAX SEGS
2220: 036E 60                  RTS
2230: 036F 20 5B 03     YOU    JSR   CARD
```

```
2240: 0372 E6 96          INC   UCNT
2250: 0374 A6 96          LDX   UCNT
2260: 0376 48             PHA
2270: 0377 98             TYA
2280: 0378 9D 3F A6       STAAX WINDOW -01
2290: 037B 68             PLA
2300: 037C A0 10          LDYIM $10
2310: 037E 90 02          BCC   YOVER
2320: 0380 84 98          STY   UACE
2330: 0382 18      YOVER  CLC
2340: 0383 F8             SED
2350: 0384 65 97          ADC   UTOT
2360: 0386 85 97          STA   UTOT
2370: 0388 D8             CLD
2380: 0389 60             RTS
2390: 038A 20 5B 03 ME    JSR   CARD
2400: 038D C6 99   MEX    DEC   MCNT
2410: 038F A6 99          LDX   MCNT
2420: 0391 48             PHA
2430: 0392 98             TYA
2440: 0393 9D 46 A6       STAAX WINDOW +06
2450: 0396 68             PLA
2460: 0397 A0 10          LDYIM $10
2470: 0399 90 02          BCC   MOVER
2480: 039B 84 9B          STY   MACE
2490: 039D 18      MOVER  CLC
2500: 039E F8             SED
2510: 039F 65 9A          ADC   MTOT
2520: 03A1 85 9A          STA   MTOT
2530: 03A3 D8             CLD
2540: 03A4 60             RTS
2550: 03A5 48      NUMDIS PHA
2560: 03A6 4A             LSRA
2570: 03A7 4A             LSRA
2580: 03A8 4A             LSRA
2590: 03A9 4A             LSRA
2600: 03AA A8             TAY
2610: 03AB B9 29 8C       LDAAY TABLE
2620: 03AE 8D 44 A6       STA   WINDOW +04
2630: 03B1 68             PLA
2640: 03B2 29 0F          ANDIM $0F
2650: 03B4 A8             TAY
2660: 03B5 B9 29 8C       LDAAY TABLE
2670: 03B8 8D 45 A6       STA   WINDOW +05
2680: 03BB 60             RTS
2690: 03BC 03      INIT   =     $03
2700: 03BD 00             =     $00
2710: 03BE 20             =     $20
2720: 03BF 01      VALUE  =     $01
2730: 03C0 02             =     $02
2740: 03C1 03             =     $03
2750: 03C2 04             =     $04
2760: 03C3 05             =     $05
2770: 03C4 06             =     $06
2780: 03C5 07             =     $07
2790: 03C6 08             =     $08
```

```
2800: 03C7 09              =  $09
2810: 03C8 10              =  $10
2820: 03C9 10              =  $10
2830: 03CA 10              =  $10
2840: 03CB 10              =  $10
2850: 03CC 77      SEGS    =  $77
2860: 03CD 5B              =  $5B
2870: 03CE 4F              =  $4F
2880: 03CF 66              =  $66
2890: 03D0 6D              =  $6D
2900: 03D1 7D              =  $7D
2910: 03D2 07              =  $07
2920: 03D3 7F              =  $7F
2930: 03D4 6F              =  $6F
2940: 03D5 71              =  $71
2950: 03D6 71              =  $71
2960: 03D7 71              =  $71
2970: 03D8 71              =  $71
2980: 03D9 6D      SHUF    =  $6D
2990: 03DA 76              =  $76
3000: 03DB 3E              =  $3E
3010: 03DC 71              =  $71
3020: 03DD 71              =  $71
3030: 03DE 38              =  $38
3040: 03DF 7C      BETQ    =  $7C
3050: 03E0 79              =  $79
3060: 03E1 78              =  $78
3070: 03E2 53              =  $53
3080: 03E3 78      TOTL    =  $78
3090: 03E4 5C              =  $5C
3100: 03E5 78              =  $78
3110: 03E6 40              =  $40
3120: 03E7 7C      BUSTED  =  $7C
3130: 03E8 3E              =  $3E
3140: 03E9 6D              =  $6D
3150: 03EA 07              =  $07
3160: 03EB 79              =  $79
3170: 03EC 5E              =  $5E
ID=
0010:
0020: 0200           BANDIT ORG    $0200
0030:
0040:                              BANDIT
0050:                BY JIM BUTTERFIELD
0060:
0070:                 ADAPTED FOR SYM
0080:                 BY GENE ZUMCHAK
0090:                     3/11/79
0100:
0110:                  COPYWRITE 1979
0120:                    E. ZUMCHAK
0130:
0131:
0132:
0140: 0200           WINDOW *      $0000
0150: 0200           AMT    *      $000B
0160: 0200           ARROW  *      $0006
0170: 0200           RWD    *      $0007
0180: 0200           STALLA *      $0008
0190: 0200           TUMBLE *      $0009
0200: 0200           TEMP   *      $000A
0210:
0220: 0200           SCAND  *      $8906
0230: 0200           TABLE  *      $8C29
0240: 0200           ACCESS *      $8B86
0250: 0200           DISBUF *      $A640
0260:
0270: 0200 20 86 8B         JSR    ACCESS
0280: 0203 A9 25            LDAIM  $25
0290: 0205 85 0B            STA    AMT
0300: 0207 20 BD 02         JSR    CVAMT
0310: 020A A9 00            LDAIM  $00
0320: 020C 85 06            STA    ARROW
0330:
0340: 020E 20 91 02  LPA    JSR    DISPLY
0350: 0211 F0 FB            BEQ    LPA
0360: 0213 E6 09     ROLL   INC    TUMBLE
0370: 0215 20 91 02         JSR    DISPLY
0380: 0218 D0 F9            BNE    ROLL
0390:
0400: 021A A9 03            LDAIM  $03
0410: 021C 85 06            STA    ARROW
0420: 021E F8               SED
0430: 021F 38               SEC
0440: 0220 A5 0B            LDA    AMT
0450: 0222 E9 01            SBCIM  $01
0460: 0224 85 0B            STA    AMT
0470: 0226 20 BD 02         JSR    CVAMT
0480: 0229 26 09            ROL    TUMBLE
0490:
0500: 022B 20 91 02  LPB    JSR    DISPLY
0510: 022E C6 08            DEC    STALLA
0520: 0230 D0 F9            BNE    LPB
0530: 0232 A6 06            LDX    ARROW
0540: 0234 A5 09            LDA    TUMBLE
```

```
0550: 0236 29 06              ANDIM $06
0560: 0238 09 40              ORAIM $40
0570: 023A 95 02              STAAX WINDOW +02
0580: 023C 46 09              LSR   TUMBLE
0590: 023E 46 09              LSR   TUMBLE
0600: 0240 C6 06              DEC   ARROW
0610: 0242 D0 E7              BNE   LPB
0620:
0630: 0244 A5 05              LDA   WINDOW +05
0640: 0246 C5 04              CMP   WINDOW +04
0650: 0248 D0 37              BNE   NOMAT
0660: 024A C5 03              CMP   WINDOW +03
0670: 024C D0 33              BNE   NOMAT
0680: 024E A2 10              LDXIM $10
0690: 0250 C9 40              CMPIM $40
0700: 0252 F0 0D              BEQ   PAY
0710: 0254 A2 08              LDXIM $08
0720: 0256 C9 42              CMPIM $42
0730: 0258 F0 07              BEQ   PAY
0740: 025A A2 06              LDXIM $06
0750: 025C C9 44              CMPIM $44
0760: 025E F0 01              BEQ   PAY
0770: 0260 CA                 DEX
0780:
0790: 0261 86 07       PAY    STX   RWD
0800: 0263 A9 80       PAX    LDAIM $80
0810: 0265 85 08              STA   STALLA
0820: 0267 20 91 02    LPC    JSR   DISPLY
0830: 026A C6 08              DEC   STALLA
0840: 026C D0 F9              BNE   LPC
0850: 026E C6 07              DEC   RWD
0860: 0270 F0 9C              BEQ   LPA
0870: 0272 18                 CLC
0880: 0273 F8                 SED
0890: 0274 A5 0B              LDA   AMT
0900: 0276 69 01              ADCIM $01
0910: 0278 B0 94              BCS   LPA
0920: 027A 85 0B              STA   AMT
0930: 027C 20 BD 02           JSR   CVAMT
0940: 027F D0 E2              BNE   PAX
0950:
0960: 0281 A2 03       NOMAT  LDXIM $03
0970: 0283 C9 46              CMPIM $46
0980: 0285 F0 DA              BEQ   PAY
0990: 0287 20 91 02    LOK    JSR   DISPLY
1000: 028A A5 0B              LDA   AMT
1010: 028C F0 F9              BEQ   LOK
1020: 028E 4C 0E 02           JMP   LPA
1030:
1040:
1050: 0291 A6 06       DISPLY LDX   ARROW
1060: 0293 10 02              BPL   INDIS
1070: 0295 F6 03       OVER   INCAX WINDOW +03
1080: 0297 CA          INDIS  DEX
1090: 0298 10 FB              BPL   OVER
1100: 029A A9 04              LDAIM $04
```

```
1110: 029C 85 0A            STA    TEMP
1120: 029E A0 00            LDYIM  $00
1130: 02A0 A2 05            LDXIM  $05
1140: 02A2 B5 00     SWITCH LDAAX  WINDOW
1150: 02A4 29 7F            ANDIM  $7F
1160: 02A6 99 40 A6         STAAY  DISBUF
1170: 02A9 C8               INY
1180: 02AA CA               DEX
1190: 02AB 10 F5            BPL    SWITCH
1200: 02AD A9 00            LDAIM  $00
1210: 02AF 8D 43 A6         STA    DISBUF +03
1220: 02B2 20 06 89  DISL   JSR    SCAND
1230: 02B5 C6 0A            DEC    TEMP
1240: 02B7 D0 F9            BNE    DISL
1250: 02B9 20 06 89         JSR    SCAND
1260: 02BC 60               RTS
1270:
1280: 02BD A5 0B     CVAMT  LDA    AMT
1290: 02BF 29 0F            ANDIM  $0F
1300: 02C1 AA               TAX
1310: 02C2 BD 29 8C         LDAAX  TABLE
1320: 02C5 85 00            STA    WINDOW
1330: 02C7 A5 0B            LDA    AMT
1340: 02C9 4A               LSRA
1350: 02CA 4A               LSRA
1360: 02CB 4A               LSRA
1370: 02CC 4A               LSRA
1380: 02CD AA               TAX
1390: 02CE BD 29 8C         LDAAX  TABLE
1400: 02D1 85 01            STA    WINDOW +01
1410: 02D3 60               RTS
ID=
0010:
0020: 0200           LANDER ORG   $0200
0030:
0040:                   LUNAR LANDER
0050:                BY JIM BUTTERFIELD
0060:
0070:                 ADAPTED FOR SYM
0080:                 BY GENE ZUMCHAK
0090:                      3/11/79
0100:
0110:                   COPYWRITE 1979
0120:                     E. ZUMCHAK
0130:
0140:
0150:
0160: 0200           ALT    *     $00D5
0170: 0200           VEL    *     $00D8
0180: 0200           THTWO  *     $00DB
0190: 0200           THRUST *     $00DD
0200: 0200           FUEL   *     $00DE
0210: 0200           MODE   *     $00E1
0220: 0200           DOWN   *     $00E2
0230: 0200           DECK   *     $00E3
0240: 0200           EX     *     $00E4
0250:
0260: 0200           SCANDS *     $8906
0270: 0200           KEYQ   *     $8923
0280: 0200           LRNKEY *     $892C
0290: 0200           DISBUF *     $A640
0300: 0200           BEEP   *     $8972
0310: 0200           SEGSMI *     $8C29
0320: 0200           ACCESS *     $8B86
0330: 0200           POINTH *     $00EF
0340: 0200           POINTL *     $00EE
0350: 0200           INH    *     $00ED
0360:
0370: 0200 20 86 8B         JSR   ACCESS
0380: 0203 A2 0D     GO     LDXIM $0D
0390: 0205 BD 03 03  LPA    LDAAX INIT
0400: 0208 95 D5            STAZX ALT
0410: 020A CA               DEX
0420: 020B 10 F8            BPL   LPA
0430:
0440: 020D A2 05     CALC   LDXIM $05
0450: 020F A0 01     RECAL  LDYIM $01
0460: 0211 F8               SED
0470: 0212 18               CLC
0480: 0213 B5 D5     DIGIT  LDAZX ALT
0490: 0215 75 D7            ADCZX ALT     +02
0500: 0217 95 D5            STAZX ALT
0510: 0219 CA               DEX
0520: 021A 88               DEY
0530: 021B 10 F6            BPL   DIGIT
0540: 021D B5 D8            LDAZX ALT     +03
0550: 021F 10 02            BPL   INCR
0560: 0221 A9 99            LDAIM $99
```

```
0570: 0223 75 D5        INCR   ADCZX ALT
0580: 0225 95 D5               STAZX ALT
0590: 0227 CA                  DEX
0600: 0228 10 E5               BPL   RECAL
0610: 022A A5 D5               LDA   ALT
0620: 022C 10 0D               BPL   UP
0630: 022E A9 00               LDAIM $00
0640: 0230 85 E2               STA   DOWN
0650: 0232 A2 02               LDXIM $02
0660: 0234 95 D5        DD     STAZX ALT
0670: 0236 95 DB               STAZX THTWO
0680: 0238 CA                  DEX
0690: 0239 10 F9               BPL   DD
0700: 023B 38           UP     SEC
0710: 023C A5 E0               LDA   FUEL  +02
0720: 023E E5 DD               SBC   THRUST
0730: 0240 85 E0               STA   FUEL  +02
0740: 0242 A2 01               LDXIM $01
0750: 0244 B5 DE        LPB    LDAZX FUEL
0760: 0246 E9 00               SBCIM $00
0770: 0248 95 DE               STAZX FUEL
0780: 024A CA                  DEX
0790: 024B 10 F7               BPL   LPB
0800: 024D B0 0C               BCS   TANK
0810: 024F A9 00               LDAIM $00
0820: 0251 A2 03               LDXIM $03
0830: 0253 95 DD        LPC    STAZX THRUST
0840: 0255 CA                  DEX
0850: 0256 10 FB               BPL   LPC
0860: 0258 20 F4 02            JSR   THRSET
0870: 025B A5 DE        TANK   LDA   FUEL
0880: 025D A6 DF               LDX   FUEL  +01
0890: 025F 09 F0               ORAIM $F0
0900: 0261 A4 E1               LDY   MODE
0910: 0263 F0 20               BEQ   ST
0920: 0265 F0 9C        GOLINK BEQ   GO
0930: 0267 F0 A4        CLINK  BEQ   CALC
0940: 0269 A2 FE               LDXIM $FE
0950: 026B A0 5A               LDYIM $5A
0960: 026D 18                  CLC
0970: 026E A5 D9               LDA   VEL   +01
0980: 0270 69 05               ADCIM $05
0990: 0272 A5 D8               LDA   VEL
1000: 0274 69 00               ADCIM $00
1010: 0276 B0 04               BCS   GOOD
1020: 0278 A2 AD               LDXIM $AD
1030: 027A A0 DE               LDYIM $DE
1040: 027C 98           GOOD   TYA
1050: 027D A4 E2               LDY   DOWN
1060: 027F F0 04               BEQ   ST
1070: 0281 A5 D5               LDA   ALT
1080: 0283 A6 D6               LDX   ALT   +01
1090: 0285 85 EF        ST     STA   POINTH
1100: 0287 86 EE               STX   POINTL
1110: 0289 A5 D9               LDA   VEL   +01
1120: 028B A6 D8               LDX   VEL
```

```
1130: 028D 10 05                BPL    FLY
1140: 028F 38                   SEC
1150: 0290 A9 00                LDAIM  $00
1160: 0292 E5 D9                SBC    VEL      +01
1170: 0294 85 ED       FLY      STA    INH
1180: 0296 A9 06                LDAIM  $06
1190: 0298 85 E3                STA    DECK
1200: 029A D8          FLITE    CLD
1210: 029B A2 03       PRESCN   LDXIM  $03
1220: 029D 86 E4                STX    EX
1230: 029F A0 00                LDYIM  $00
1240: 02A1 C6 E4       PSLP     DEC    EX
1250: 02A3 30 1E                BMI    SCAN
1260: 02A5 A6 E4                LDX    EX
1270: 02A7 B5 ED                LDAZX  INH
1280: 02A9 48                   PHA
1290: 02AA 4A                   LSRA
1300: 02AB 4A                   LSRA
1310: 02AC 4A                   LSRA
1320: 02AD 4A                   LSRA
1330: 02AE 20 BA 02             JSR    DIG
1340: 02B1 68                   PLA
1350: 02B2 29 0F                ANDIM  $0F
1360: 02B4 20 BA 02             JSR    DIG
1370: 02B7 4C A1 02             JMP    PSLP
1380: 02BA AA          DIG      TAX
1390: 02BB BD 29 8C             LDAAX  SEGSM1
1400: 02BE 99 40 A6             STAAY  DISBUF
1410: 02C1 C8                   INY
1420: 02C2 60                   RTS
1430: 02C3 20 06 89    SCAN     JSR    SCANDS
1440: 02C6 F0 14                BEQ    NOKEY
1450: 02C8 20 2C 89             JSR    LRNKEY
1460: 02CB 48                   PHA
1470: 02CC 20 72 89             JSR    BEEP
1480: 02CF 20 23 89    Q        JSR    KEYQ
1490: 02D2 D0 FB                BNE    Q
1500: 02D4 68                   PLA
1510: 02D5 C9 0D                CMPIM  $0D
1520: 02D7 F0 8C                BEQ    GOLINK
1530: 02D9 20 E2 02             JSR    DOKEY
1540: 02DC C6 E3       NOKEY    DEC    DECK
1550: 02DE D0 E3                BNE    SCAN
1560: 02E0 F0 85                BEQ    CLINK
1570: 02E2 C9 40       DOKEY    CMPIM  $40
1580: 02E4 90 05                BCC    NUMBER
1590: 02E6 49 46                EORIM  $46
1600: 02E8 85 E1                STA    MODE
1610: 02EA 60          RETRN    RTS
1620: 02EB 29 0F       NUMBER   ANDIM  $0F
1630: 02ED AA                   TAX
1640: 02EE A5 DD                LDA    THRUST
1650: 02F0 F0 F8                BEQ    RETRN
1660: 02F2 86 DD                STX    THRUST
1670:
1680: 02F4 A5 DD       THRSET   LDA    THRUST
```

```
1690: 02F6 38                  SEC
1700: 02F7 F8                  SED
1710: 02F8 E9 05               SBCIM $05
1720: 02FA 85 DC               STA   THTWO  +01
1730: 02FC A9 00               LDAIM $00
1740: 02FE E9 00               SBCIM $00
1750: 0300 85 DB               STA   THTWO
1760: 0302 60                  RTS
1770:
1780: 0303 45           INIT   =     $45
1790: 0304 01                  =     $01
1800: 0305 00                  =     $00
1810: 0306 99                  =     $99
1820: 0307 81                  =     $81
1830: 0308 00                  =     $00
1840: 0309 99                  =     $99
1850: 030A 97                  =     $97
1860: 030B 02                  =     $02
1870: 030C 08                  =     $08
1880: 030D 00                  =     $00
1890: 030E 00                  =     $00
1900: 030F 01                  =     $01
1910: 0310 01                  =     $01
ID=
```

SYM STOPWATCH

S.E.Curd
319 Zoe Avenue
Buckner, MO 64016

This program uses the 6522 #1 timer as a self-reloading, non-interrupting 0.0100 second timer.

Stopwatch commands include:

- '1' -- start time
- '2' -- store current time in lap memory
- '0' -- stop time
- '→' -- exchange displayed time with lap memory
- 'C' -- clear time and lap memory to zero

It is very important to note two consequences based on monitor I/O routines. First, the timer does not begin timing until the <u>release</u> of the '1' key, rather than its depression. Secondly, the time value stored in lap memory is the value of the timer at the <u>release</u> of the '2' key.

After beginning the execution of this program at $0000 and starting the timer, you will see displayed

mm.ss.ff

as minutes,seconds,and fractional seconds (tenths and hundredths). You may then press and release the '2' key at any time to store a lap, or intermediate time. After stopping the timer, you may consequently examine cumulative or lap time by depressing '→'.

```
0000 20 86 8B  JSR access
0003 A9 40     LDA# $40
0005 8D 08 A0  STA  $A008
0008 A9 10     LDA# $10
000A 8D 06 A0  STA  $A006
000D A9 27     LDA# $27
000F 8D 07 A0  STA  $A007
0012 8D 05 A0  STA  $A005
0015 A2 07     LDX# $07
0017 A9 00     LDA# $00
0019 95 8D     STAz $8D,x
001B A9 3F     LDA# $3F
001D 15 86     ORAz $86,x
```

```
001F 9D 3F A6  STA  $A63F,x
0022 95 C4     STAz $C4,x
0024 CA        DEX
0025 D0 F0     BNE  $F0 (-16)
0027 20 AF 8B  JSR getkey
002A C9 43     CMP# 'C'
002C F0 E7     BEQ  $E7 (-25)
002E C9 3E     CMP# '→'
0030 D0 0F     BNE  $0F (+15)
0032 A2 06     LDX# $06
0034 BC 3F A6  LDY  $A63F,x
0037 B5 C4     LDAz $C4,x
0039 9D 3F A6  STA  $A63F,x
```

```
003C 94 C4     STYz $C4,x
003E CA        DEX
003F D0 F3     BNE  $F3 (-13)
0041 C9 31     CMP# '1'
0043 D0 E2     BNE  $E2 (-30)
0045 20 06 89  JSR scand
0048 20 23 89  JSR keyq
004B 20 2C 89  JSR lrnkey
004E C9 30     CMP# '0'
0050 F0 D5     BEQ  $D5 (-43)
0052 C9 32     CMP# '2'
0054 D0 0A     BNE  $0A (+10)
0056 A2 06     LDX# $06
0058 BD 3F A6  LDA  $A63F,x
005B 95 C4     STAz $C4,x
005D CA        DEX
005E D0 F8     BNE  $F8 (-8)
0060 2C 0D A0  BIT  $A00D
0063 50 E0     BVC  $E0 (-32)
0065 AD 0D A0  LDA  $A00D
0068 8D 0D A0  STA  $A00D
006B 38        SEC
006C A2 06     LDX# $06
```

```
006E B5 BD     LDAz $BD,x
0070 69 00     ADC# $00
0072 D5 AF     CMPz $AF,x
0074 D0 03     BNE  $03 (+3)
0076 38        SEC
0077 A9 00     LDA# $00
0079 95 BD     STAz $BD,x
007B A8        TAY
007C B9 29 8C  LDA  $8C29,y
007F 15 B6     ORAz $B6,x
0081 9D 3F A6  STA  $A63F,x
0084 CA        DEX
0085 D0 E7     BNE  $E7 (-25)
0087 F0 BC     BEQ  $BC (-68)

;DATA TABLES

00B0 06,0A,06,0A,0A,0A,00

00B7 00,80,00,80,00,00,00

00BE 00,00,00,00,00,00,00

00C5 00,00,00,00,00,00,00
```

As written, SYM STOPWATCH will display with 0.01 second resolution up to 59 minutes, 59.99 seconds, and store one lap time.

Note that the lap time stored cannot be read unless the timer is stopped, and that the stored time reflects the time at the last pressing of the '2' key.

After stopping the timer, restarting it will cause it to resume timing from the last timer value, even if you have exchanged the displayed value and the lap memory. However this exchange will put the last timer value into the lap memory and eliminate the previous lap time.

AMERICAN INSTITUTE FOR PROFESSIONAL
EDUCATION
Carnegie Building, Hillcrest Road
Madison, New Jersey 07940

PROGRAM ____________________

PROGRAMMER ____________________

DATE ____________________

ADDR	INSTRUCTIONS			LABEL	NMEMONIC	OPERAND	COMMENTS
	B1	B2	B3				

	#	AB	ZP	AC
ADC	69	6D	65	
AND	29	2D	25	
ASL		ØE	Ø6	ØA
BIT		2C	24	
CMP	C9	CD	C5	
CPX	EØ	EC	E4	
CPY	CØ	CC	C4	
DEC		CE	C6	
EOR	49	4D	45	
INC		EE	E6	
LDA	A9	AD	A5	
LDX	A2	AE	A6	
LDY	AØ	AC	A4	
LSR		4E	46	4A
ORA	Ø9	ØD	Ø5	
ROL		2E	26	2A
ROR		6E	66	6A
SBC	E9	ED	E5	
STA		8D	85	
STX		8E	86	
STY		8C	84	

BRK	ØØ	SEC	38
CLC	18	SED	F8
CLD	D8	SEI	78
CLI	58	TAX	AA
CLV	B8	TAY	A8
DEX	CA	TSX	BA
DEY	88	TXA	8A
INX	E8	TXS	9A
INY	C8	TYA	98
JMP	4C	BCC	9Ø
JSR	2Ø	BCS	BØ
NOP	EA	BEQ	FØ
PHA	48	BMI	3Ø
PHP	Ø8	BNE	DØ
PLA	68	BPL	1Ø
PLP	28	BVC	5Ø
RTI	4Ø	BVS	7Ø
RTS	6Ø		