6502 GAMES

Learn how to play sophisticated games in the powerful 6502 assembly-level language—and learn the assembly language as well.

6502 Games lets you play the ten games shown below, but it does much more: it teaches assembly language programming in a straightforward and enjoyable manner. You will learn the techniques of algorithm design and data structures so you can program your 6502 not only to play games but also to perform a variety of tasks, from home applications to industrial controls.

With a minimum of external hardware, you will be able to play:

- Magic Squares
- Hexguess
- Echo
- Tic-Tac-Toe
- Slot Machine
- Spinner
- Mindbender
- Music
- Translate
- Blackjack

THE AUTHOR

Dr. Rodnay Zaks has taught courses on programming and microprocessors to several thousand people worldwide. He received his Ph.D. in Computer Science from the University of California, Berkeley, developed a microprogrammed APL implementation, and worked in Silicon Valley, where he pioneered the use of microprocessors in industrial applications. He has authored several best-selling books on microcomputers, now available in ten languages. This book, like the others in the series, is based on his technical and teaching experience.
ACKNOWLEDGEMENTS

The author would like to acknowledge the contributions of Chris Williams and Eric Novikoff, who thoroughly checked all of the game programs and contributed numerous ideas for improvements.

The author is particularly indebted to Eric Novikoff for his valuable assistance throughout all phases of the manuscript's production, and for his meticulous supervision of the final text.

Notice

SYM is a trademark of Synertek Systems, Inc.
KIM is a trademark of MOS Technology, Inc.
AIM65 is a trademark of Rockwell International, Inc.
"COMPUTEACHER" and "GAMES BOARD" are trademarks of Sybex, Inc.

Cover Design by Daniel Le Noury
Technical Illustrations by Guy S. Orcutt and J. Trujillo Smith

Every effort has been made to supply complete and accurate information. However, Sybex assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which would result. No license is granted by the equipment manufacturers under any patent or patent rights. Manufacturers reserve the right to change circuitry at any time without notice.

Copyright © 1980 SYBEX Inc. World rights reserved. No part of this publication may be stored in retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 80-50896
ISBN 0-89588-022-9
Printed in the United States of America
Printing 10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE ................................................ ix

1. INTRODUCTION ......................................... 1
The Games Board.

2. MUSIC PLAYER ........................................ 20
Play a sequence of up to 255 notes (13 different notes) and record it automatically.

3. TRANSLATE .......................................... 41
The computer displays a binary number. Each player in turn must press the hexadecimal equivalent as quickly as possible. The first to score 10 wins. Designed for two players.

4. HEXGUESS ........................................... 59
Guess a 2-digit hex number generated by the computer. The computer will tell you how far off your guess is. You are allowed up to 10 guesses.

5. MAGIC SQUARE ...................................... 73
Light up a perfect square on the board. Each key inverts some LED pattern. Skill and logic are required.

6. SPINNER ........................................... 87
A light is spinning around a square. You must catch it by hitting the corresponding key. Every time you succeed, it will spin faster. A game of skill.

7. SLOT MACHINE ..................................... 99
A Las Vegas type slot machine is simulated, with three spinning wheels. Try your luck.
8. ECHO ........................................... 137
Recognize and duplicate a sound/light sequence (also known as SIMON—a manufacturer trademark).

9. MINDBENDER .................................. 162
Play against the dealer (the computer) with a deck of 10 cards. You may hit or stay. Don’t bust!

10. BLACKJACK .................................... 189
Guess a sequence of numbers generated by the computer. It will tell you how many digits are correct and in the right position (also known as MASTERMIND—a manufacturer trademark).

11. TIC-TAC-TOE ................................... 218
Try to achieve three in a row before the computer does in this favorite game of strategy. The computer’s ability improves with yours. Can you outsmart it?

APPENDICES ..................................... 287
A. 6502 Instructions—Alphabetic
B. 6502—Instruction Set: Hex and Timing

INDEX ............................................. 290


THE 6502 SERIES

BOOKS
Vol. 1—Programming the 6502 (Ref. C202)
Vol. 2—Programming Exercises for the 6502 (Ref. C203)
Vol. 3—6502 Applications Book (Ref. D302)
Vol. 4—6502 Games Book

SOFTWARE
6502 Assembler in BASIC
Games Cassette for SYM
Application Programs
8080 Simulator for 6502 (KIM and APPLE versions)

EDUCATIONAL SYSTEM
Computeacher™
Games Board™

PREFACE

“Complex algorithms can be fun!”

Programming is often treated by programmers as a game, although they may not readily admit it. In fact, using and programming a computer may well be one of the ultimate intellectual games devised to date.

A program is a projection of one’s intelligence and skills. Writing games programs adds an essential ingredient to it: fun. However, most interesting games are fairly complex to program, and demand specific programming skills.

This book will teach you how to program a complete array of games ranging from passive ones (Music) to strategic ones (Tic-Tac-Toe). In the process of learning how to program these games, you will sharpen your skills at using input/output techniques, such as timers and interrupts. You will also use various data structures, and improve or develop your assembly-level programming skills.

This book has been designed as an educational text. After reading it you should be able to create programs for additional games and to use your programming skills for other applications.

If you have access to a microcomputer board, you can also enjoy the results of your work in a very short time. The programs presented in this book are listed for the SYM board (from Synertek Systems), but can be adapted to other 6502-based microcomputers. Playing the games will require building a simple, low-cost “Games Board,” which is described in Chapter 1. To facilitate game playing, a “Games Cassette” is also available in SYM format.

The many games studied in this book include: musical games (MUSIC), educational games (TRANSLATE and HEXGUESS will teach you hexadecimal), games involving the use of logic (MAGIC SQUARES), games involving coordination (SPINNER), memory games (ECO), games of chance (SLOT MACHINES), games involving strategy (TIC-TAC-TOE), and games involving various combinations of skills (BLACKJACK).

A basic format has been followed in presenting each game program. It includes:

1. The rules of the game
2. Instructions for playing a typical game

ix
3. The algorithm(s) (theory of operation)
4. The program: data structures, programming techniques, subroutines.

Variations and exercises are also suggested throughout the book.

Thus, you will first learn how to play the game, and then how to devise a possible solution (the algorithm). Finally, you will actually implement a complete, programmed version of the algorithm in 6502 assembly-level language, paying specific attention to the required data structures and techniques used for efficient programming.

Learning to program in assembly-level language has traditionally been unappealing or difficult. It need not be. It can be fun. If you are familiar with elementary programming techniques on the level of reference text C202—Programming the 6502, this book will teach you practical programming techniques in a game context. It will both integrate theoretical concepts into complex programs and present a simple step-by-step analysis of program development. These same concepts and techniques can be applied to any programming problem, from industrial control to business applications.

It is hoped that you will have as much fun learning how to program as you will have playing the games. If you have invented, developed, or know of other games that you would like to see included in a games book, please write to me.

RODNAY ZAKS
The programs in this book can be used to play real games. They require using a 6502-based board such as the SYM board (manufactured and trademarked by Synertek Systems) and they require building a simple “Games Board.” A complete description of the Games Board will be provided in this chapter. The Games Board is shown in Figure 1.1.

The programs in this book will all run as they are presented on a SYM board, but they can easily be adapted to any other 6502-based computer. The input/output lines available, however, are usually specific to the microcomputer used. The input/output segments of the various programs must then be modified accordingly. Naturally, the algorithms themselves as well as the programming techniques used to implement them normally remain unchanged.

After reading this book, especially if you should try to run the programs on the Games Board, you will probably agree that:

“Complex algorithms can be fun!”

HARDWARE REQUIRED

In order to run the programs presented in this book on an actual microcomputer, a SYM or other 6502-based board should be used. Additionally, a Games Board will be required to play the games. A photograph of the Games Board is shown in Figure 1.1. The Games Board is the input/output board on which the games will be played. The keyboard on the right is used to provide an input to the microcomputer board, while the LEDs on the left are used to display the information sent by the program. The use of the keys and the LEDs will be explained for each game in this book. A speaker is also attached for sound effects. It has been mounted in an enclosure (box), for improved sound quality. (See Figure 1.2.)

The Games Board may easily be built at home from a small number of low-cost components, or may be obtained from Sybex. Since its assembly is quite simple, the reader interested in obtaining a better understanding of the hardware is strongly encouraged to purchase the parts and build the board. On the other hand, building the Games Board is not a required action in order to use this text. It simply offers additional depth of understanding.

CONNECTING THE SYSTEM

It is assumed here that you own a 6502-based microcomputer board, such as a SYM board, and that you have built or obtained a
Games Board. This section will describe how to interconnect the elements of the system so that you can actually play the games which will be described in the following chapters. If you do not have access to this hardware, it is not essential that you read through this section. However, you may wish to refer to it later, in order to implement the games described in this book, or to understand the interfacing and input/output techniques.

Four essential components are required:

1 - the power supply
2 - the SYM board
3 - the Games Board
4 - (preferably) a cassette recorder

The first requirement is to connect the wires to the power supply. If it is not already so equipped, two sets of wires must be connected to it. (See Figure 1.3.) First, it must be connected to a power cord. Second, the ground and plus 5V wires must be connected to the SYM power connector, as per the manufacturer’s specifications.

Next, the Games Board should be physically connected to the SYM. Two edge connectors are required for the SYM: both the A connector and the AA connector are used. (See Figure 1.4.) There is also a power source connector.

Always be careful to insert the connectors with the proper side up (usually the printed side). An error in inserting the power connector, in particular, will have highly unpleasant results. Errors in inserting the I/O connectors are usually less damaging.

Finally, if a cassette recorder is to be used (highly recommended), the SYM board must be connected to a tape recorder. At the minimum, the “monitor” or “earphone” wires should be connected, and preferably the “remote” wire as well. If new programs are going to be stored on tape, the “record” or “microphone” wire should also be connected. (See Figure 1.5.) Details for these connections are given in the SYM manual.

At this point the system is ready to be used. (See Figure 1.6.) If you have one of the games cassettes (available separately from Sybex), simply load the cassette into the tape recorder. Press the RST key after powering up your SYM, and load the appropriate game into your SYM. You are ready to play.

Otherwise, you should enter the hexadecimal object code of the game on the SYM keyboard. All games are started by jumping to location 200 (“GO 200”).
Fig. 1.5: Connecting the Cassette Recorder

INTRODUCTION

GAMES BOARD INTERCONNECT

The Keyboard

The board's components are shown in Figure 1.7. The LED arrangement used for the games is shown in Figure 1.8. The keyboard used here is of the "line per key" type, and does not use a matrix arrangement. Sixteen keys are required for the games, even though more keys are often provided on a number of "standard keyboards," such as the one used in the prototype of Figure 1.7. On this prototype, the three keys at the bottom right-hand corner are not used (keys H, L, and "shift").

Figure 1.9 shows how a 1-to-16 decoder (the 74154) is used to identify the key which has been pressed, while tying up only four output lines (PB0 to PB3) — four lines allow 16 codes. The keyboard scanning program will send the numbers 0-15 in succession out on lines PB0-PB3. In response, the 74154 decoder will decode its input (4 bits) into each one of the 16 outputs in sequence. For example, when the number "0000" (binary) is output on lines PB0 to PB3, the 74154 decoder grounds line 1 corresponding to key "0". This is illustrated in Figure 1.9. After outputting each four-bit combination, the scanning program reads the value of PA7. If the key currently grounded was not pressed, PA7 will be high. If the corresponding key was pressed, PA7 will be grounded and a logical "0" will be read. For example, in
Figure 1.10, a key closure for key 1 has been detected. As in any scanning algorithm, a good program will debounce the key closures by implementing a delay. For more details on specific keyboard interfacing techniques, the reader is referred to reference C207 — *Microprocessor Interfacing Techniques*.

In the actual design, the four inputs to the 74154 (PB0 to PB3) are connected to VIA #3 of the SYM. PA7 is connected to the same VIA. The 3.3 K resistor on the upper right-hand corner of Figure 1.9 pulls up PA7 and guarantees a logic level "1" as long as no grounding occurs.

The GETKEY program, or a similar routine, is used by all the programs in this book and will be described below.

**The LEDs**

The connection of the fifteen LEDs is shown in Figure 1.11. Three 7416 LED drivers are used to supply the necessary current (16 mA).

The LEDs are connected to lines PA0 to PA7 and PB0 to PB7, excepting PB6. These ports belong to VIA #1 of the SYM. An LED is lit by simply selecting the appropriate input pin of the corresponding driver. The resulting arrangement is shown in Figure 1.12 and Figure 1.13.
The resistors shown in Figure 1.11 are 330-ohm resistors designed as current limiters for the 7416 gates.

The output routines will be described in the context of specific games.

Required Parts

- One 6" × 9" vector-board
- One 4-to-16 decoder (74154)
- Three inverting hex drivers (7416)
- One 24-pin socket
- Three 14-pin sockets (for the drivers)
- One 16-key keyboard, unencoded
- Fifteen 330-ohm resistors
- One 3.3 K-ohm resistor
- One decoupling capacitor (.1 mF)
- Fifteen LEDs
- One speaker
- One 50-ohm or 110-ohm resistor (for the speaker)
- Two 15"-20" long 16-conductor ribbon cables
- One package of wire-wrap terminal posts
- Wire-wrap wire
- Solder

A soldering iron and a wire-wrapping tool will also be required.
A suggested assembly procedure is the following: the keyboard can be glued directly to the perf board. Sockets and LEDs can be positioned on the board and held in place temporarily with tape. All connections can then be wire-wrapped. In the case of the prototype, the connections to the keyboard were soldered in order to provide reliable connections since they were not designed as wire-wrap leads. Wire-wrap terminal posts were used for common connections.

Additionally, on the prototype two sockets were provided for convenience when attaching the ribbon cable connector to the Games Board. They are not indispensable, but their use is strongly suggested in order to be able to conveniently plug and unplug cables. (They appear in the top left corner of the photograph in Figure 1.14.) A 14-pin socket and a 16-pin socket are used for this purpose. Wire-wrap terminal posts can be used instead of these sockets to attach the ribbon cable directly to the perf board. The other end of the ribbon cable is simply attached to the edge connectors of the SYM. When connecting the ribbon cable at either end, always be very careful to connect it to the appropriate pins (do not connect it upside down). The Games Board derives its power from the SYM through the ribbon cable connection. Connecting the cable in reverse will definitely have adverse effects.

The speaker may be connected to any one of the output drivers PB4, PB5, PB6, or PB7 of VIA #3. Each of these output ports is equipped with a transistor buffer. A 110-ohm current-limiting resistor is inserted in series with the speaker.

The Keyboard Input Routine

This routine, called “GETKEY,” is a utility routine which will scan the keyboard and identify the key that was pressed. The corresponding code will be contained in the accumulator. It has provisions for bounce, repeat, and rollover.

Keyboard bounce is eliminated by implementing a 50 ms delay upon detection of key closure.

The repeat problem is solved by waiting for the key currently
 pressed to be released before a new value is accepted. This cor­
responds to the case in which a key is pressed for an extended period
of time. Upon entering the GETKEY routine, a key might already be
depressed. It will be ignored until the program detects that a key is no
longer pressed. The program will then wait for the next key closure. If
the processing program using the GETKEY routine performs long
computations, there is a possibility that the user may push a new key
on the keyboard before GETKEY is called again. This key closure will
be ignored by GETKEY, and the user will have to press the key again.

Most of the programs described in this book have audible prompts
in the form of a tone which is generated every time the player should
respond. Note that when a tone is being generated or during a delay
loop in a program, pressing a key will have absolutely no effect.

Fig. 1.15: VIA Connection to Keyboard Decoder

Fig. 1.16: GETKEY Flowchart
The hardware configuration for the GETKEY routine is shown in Figure 1.9. The corresponding input/output chip on the SYM is shown in Figure 1.15. VIA #3 of the SYM board is used to communicate with the keyboard. Port B of the VIA is configured for output and lines 0 through 3 are gated to the 74154 (4-to-16 decoder), connected to the keyboard itself. The GETKEY routine will output the hexadecimal numbers “0” through “F,” in sequence, to the 74154. This will result in the grounding of the corresponding output line of the 74154. If a key is pressed, bit 7 of VIA #3 of Port A will be grounded. The program logic is, therefore, quite simple, and the corresponding flowchart is shown in Figure 1.16.

The program is shown in Figure 1.17. Let us examine it. The GETKEY routine can be relocated, i.e., it may be put anywhere in the memory. In order to conserve space, it has been located at memory stack memory area. Any user programs which might require a full stack would overwrite this routine and thus destroy it. To prevent this possibility, it could be located elsewhere. For all of the programs that will be developed in this book, however, this placement is adequate.

The first four instructions of the routine condition the data direction registers of Port A (all zeroes), while the data direction register for Port B is set for output (all ones). This is illustrated in Figure 1.15.

LDA #0
STA DDR3A
LDA #$FF
STA DDR3B

Two instructions are required to test bit 7 of Port 3A, which indicates whether a key closure has occurred:

START
BIT PORT3A
BPL START

The key counter is initially set to the value 15, and will be decremented until a key closure is encountered. Index register X is used to contain this value, as it can readily be decremented with the DEX instruction:

RSTART
LDX #15

This value (15) is then output to the 74154 and results in the selection of line 17 connected to key 15 (“F”). The BIT instruction above is used to test the condition of bit 7 of Port 3A to determine whether this key has been pressed.

NXTKEY STX PORT3B
BIT PORT3A
BPL BOUNCE

If the key were closed, a branch would occur to “BOUNCE,” and a
delay would be implemented to debounce it; otherwise, the counter is
decremented, then tested for underflow. As long as the counter does
not become negative, a branch back occurs to location NXTKEY.
This loop is repeated until a key is found to be depressed or the
counter becomes negative. In that case, the routine loops back to loca-
tion RSTART, restarting the process:

```
DEX
BPL NXTKEY
BMI RSTART
```

Note that this will result in the detection of the highest key pressed
in the case in which several keys are pressed simultaneously. In other
words, if keys “F” and “3” were pressed simultaneously, key “F”
would be identified as depressed, while key “3” would be ignored.
Avoiding this problem is called multiple-key rollover protection and
will be suggested as an exercise:

**Exercise 1-1:** In order to avoid the multiple-key rollover problem,
modify the GETKEY routine so that all 15 key closures are monitored.
If more than one key is pressed, the key closure is to be ignored until
only one key closure is sensed.

Once the key closure has been identified, the corresponding key
number is saved in the accumulator. A delay loop is then implemented
in order to provide a 50 ms debouncing time. During this loop, the key
closure is constantly monitored. If the key is released, the routine is
restarted. The delay itself is implemented using a standard two-level,
nested loop technique.

```
BOUNCE   TXA
         LDY #$12
LP1      LDX #$FF
LP2      BIT PORT3A
         BMI RSTART
         DEX
         BNE LP2
         DEY
         BNE LP1
```

**Exercise 1-2:** The value used for the outer loop counter ("$12," or 12
hexadecimal) may not be quite accurate. Compute the exact duration
of the delay implemented by the instructions above, using the tables
showing the duration of each instruction in the Appendix.

**SUMMARY**

Executing the games programs requires a simple Games Board which
provides the basic input/output facilities. The required hardware and
software interface has been described in this chapter. Photographs of
the assembled board which evolved from the prototype are shown in
Figures 1.18 and 1.19.
2

MUSIC PLAYER

THE RULES

This game allows music to be played directly on the keyboard of a computer. In addition, the program will simultaneously record the notes that are played, and then automatically play them back upon request. Keys "0" through "C" on the keyboard are used to play the musical notes. (See Figure 2.1.) Key "D" is used to specify a rest. Key "E" is used to play back the musical sequence stored in the memory. Finally, key "F" is used to clear the memory, i.e., to start a new game. The following paragraph will describe the usual sequence of the game.

<table>
<thead>
<tr>
<th>A (A)</th>
<th>B (B)</th>
<th>C (C)</th>
<th>D (REST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (A)</td>
<td>2 (B)</td>
<td>3 (C)</td>
<td>E (PBK)</td>
</tr>
<tr>
<td>4 (D)</td>
<td>5 (E)</td>
<td>6 (F)</td>
<td>F (RST)</td>
</tr>
<tr>
<td>7 (F#)</td>
<td>8 (G)</td>
<td>9 (G#)</td>
<td>0 (G)</td>
</tr>
</tbody>
</table>

Fig. 2.1: Playing Music on the Keyboard

<table>
<thead>
<tr>
<th>KEY NUMBER</th>
<th>NOTE</th>
<th>KEY NUMBER</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>G</td>
<td>B</td>
<td>G</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>9</td>
<td>G#</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>D</td>
<td>REST</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>E</td>
<td>PLAY</td>
</tr>
<tr>
<td>7</td>
<td>F#</td>
<td>F</td>
<td>RESTART</td>
</tr>
</tbody>
</table>

9th Symphony:
5 5 6 8 8 6 5 4 3 3 4 5 4 4 D 5 5
5 6 8 8 6 5 4 3 3 4 5 4 4 D 5 4 4
5 3 4 6 5 3 4 6 5 4 3 4 4 D

Clementine:
3 3 3 D 2 D 5 5 5 D 3 3 8 D 6
8 6 5 4 D D D D 4 5 6 6 6 6 5 4 5 5 D
3 D 3 5 4 D D 2 3 4 3

Frere Jacques:
3 4 5 3 4 5 3 5 6 8 D 5 6 6 8 D 8
A B 6 5 D 3 D 8 A B 6 5 5 3 D D
2 D 3 D D D D 2 D 3

Jingle Bells:
5 5 5 D 5 5 5 D 5 8 3 4 5 D D 6
6 6 6 6 5 5 5 8 8 6 6 4 3

London Bridge:
8 A 8 6 5 6 8 D 4 5 6 5 6 8 D 8
A 8 6 5 6 8 D 4 5 6 8 D 5 3

Mary Had a Little Lamb:
5 4 3 4 5 5 5 D 4 4 4 D 5 8 8 D 5
4 3 4 5 5 5 5 4 4 5 4 3

Row Row Row Your Boat:
3 D 3 D 3 4 5 5 4 5 4 5 6 8 D D 6 5 6 5 5 5 4 4 5 6 5 4 4

Silent Night:
8 D D A 8 D D D 8 D 5 5 5 D 8 D D D 8 D D A 8 D D
D 8 D D 3 D 3 D D B D D D D D C D D C
D 8 D D 8 D D C D 8 5 8 D 6 6 4 D 3

Twinkle Twinkle Little Star:
3 3 8 8 A A 8 D 6 6 5 5 4 4 3 D B
8 6 6 5 5 4 D 3 3 8 8 A A 8 D 6 6
5 5 4 4 3

Fig. 2.2: Simple Tunes for Computer Music
A TYPICAL GAME

Press key "F" to start a new game. A three-note warble will be heard, confirming that the internal memory has been erased. Play the tune on keys "0" through "D" (using the notes and the rest features). Up to 254 notes may be played and stored in the memory. At any point, the playback key ("E") may be pressed and the notes and rests that were just played on the keyboard (and simultaneously stored in the memory) will be reproduced. The musical sequence may be played as many times as desired by simply pressing key "E." Examples of simple tunes or musical sequences that can be played on the computer are shown in Figure 2.2.

THE CONNECTIONS

This game uses the keyboard plus the speaker. The speaker is connected in series to one of the buffered output lines of PORT B of VIA #3, via a 110-ohm current limiting resistor. PB4, PB5, PB6, or PB7 of VIA #3 are used, as they are driven by a transistor buffer on the SYM. For higher quality music, it is recommended that the speaker be placed in a small box-type enclosure. The value of the resistor may also be adjusted for louder volume (without going below 50-ohm) to limit the current in the transistor.

THE ALGORITHM

A tone (note) is simply generated by sending a square wave of the appropriate frequency to the speaker, i.e., by turning it on and off at the required frequency. This is illustrated in Figure 2.3. The length of time during which the speaker is on or off is known as the half-period. In this program, the frequency range of 195 to 523 Hertz is provided. If N is the frequency, the period \( T \) is the inverse of the frequency, or:

\[
T = \frac{1}{N}
\]

Therefore, the half-periods will range from \( 1/(2 \times 195) = 0.002564 \) to \( 1/(2 \times 523) = 0.000956 \) microseconds. A classic loop delay will be used to implement the required frequency.

Actual computations for the various program parameters will be presented below.

THE PROGRAM

The program is located at memory addresses 200 through 2DD, and the recorded musical sequence or tune is stored starting at memory location 300. Up to 254 notes may be recorded in 127 bytes.

Data Structures

Three tables are used in this program. They are shown in Figure 2.4. The recorded tune is stored in a table starting at address 300. The note constants, used to establish the frequency at which the speaker will be toggled, are stored in a 16-byte table located at memory address 2C4. The note durations, i.e., the number of half-cycles required to implement a uniform note duration of approximately .21 second, are stored in a 16-byte table starting at memory address 2D1. Within the tune table, two "nibble"-pointers are used: PILEN during input and PTR during output. (Each 8-bit byte in this table contains two notes.) In order to obtain the actual table entry from the nibble-pointer, the pointer is simply shifted one bit position to the right. The remaining value becomes a byte-pointer, while the bit shifted into the carry flag specifies the left or the right half of the byte. The two tables called CONSTANTS and NOTE DURATIONS are simply reference tables used to determine the half-frequency of a note and the number of times the speaker should be triggered once a note has been identified or specified. Both of these tables are accessed indirectly using the X register.

Some Music Theory

A brief survey of general music conventions is in order before describing the actual program. The frequencies used to generate the desired notes are derived from the equally tempered scale, in which the frequencies of succeeding notes are in the ratio:

\[
1 : \sqrt[12]{2}
\]

The frequencies for the middle C octave are given in Figure 2.5. When computing the corresponding frequencies of the higher or the
Generating the Tone

The half-period delay for the square wave sent to the speaker is implemented using a program loop with a basic 10 $\mu$s cycle time. In the program, the “loop index,” or iteration counter is used to count the number of 10 $\mu$s cycles executed. The loop will result in a total delay of:

$$(\text{loop index}) \times 10 - 1 \text{ microseconds}$$

On the last iteration of the loop (when the loop index is decremented to zero), the branch instruction at the end will fail. This branch instruction will execute faster, so that one microsecond (assuming a 1 MHz clock) must be subtracted from the total delay duration. The tone generation routine is shown below:

```
 TONE
 mm
 FL2
 FL1
 STA FREQ
 LDA #$FF
 STA DDRB
 LDA #$00
 LDX DUR

 INNER
 LOOP

 OUTER
 LOOP

 CLC
 BCC +2
 BNE FL1
 EOR #$FF
 STA OPB
 DEX
 BNE FL2
 RTS
```

Note the “classic” nested loop design. Every time it is entered, the outer loop adds an additional thirteen microseconds delay: 14 microseconds for the extra instructions (LDY, EOR, STA, DEX, and
BNE), minus one microsecond for responding to the unsuccessful inner loop branch. The total outer loop delay introduced is therefore:

\[(\text{loop index}) \times 10 + 13 \text{ microseconds}\]

Remember that one pass through the outer loop represents only a half-period for the note.

**Computing the Note Constants**

Let “ID” be the inner loop delay and “OD” be the outer loop additional delay. It has been established in the previous paragraph that the half-period is \(T/2 = (\text{loop index}) \times 10 + 13\) or,

\[T/2 = (\text{loop index}) \times \text{ID} + \text{OD}\]

The note constant stored in the table is the value of the “index” required by the program. It is easily derived from the equation that:

\[\text{note constant} = \text{loop index} = (T - 2 \times \text{OD})/2 \times \text{ID}\]

The period may be expressed in function of the frequency as \(T = 1/N\) or, in microseconds:

\[T = 10^6/N\]

Finally, the above equation becomes:

\[\text{note constant} = (10^6/N - 2 \times \text{OD})/2 \times \text{ID}\]

For example, let us compute the note constant corresponding to the frequency for middle C. The frequency corresponding to middle C is shown in Figure 2.5. It is 261.62 Hertz. The “OD” delay has been shown above to be 13 microseconds, while “ID” was set to 10 microseconds. The note constant equation becomes:

\[\text{note constant} = (10^6/N - 2 \times 13)/2 \times 10\]

\[= \frac{1000000/261.62 - 26}{20}\]

\[= 190 \text{ (or BE in hexadecimal)}\]

It can be verified that this corresponds to the fourth entry in the table.

---

**Exercise 2-1:** Using the table in Figure 2.6, compute the corresponding frequency, and check to see if the constants have been chosen correctly.

**Computing the Note Durations**

The DURTAB table stores the note durations expressed in numbers equivalent to the number of half-cycles for each note. These durations have been computed to implement a uniform duration of approximately .2175 second per note. If \(D\) is the duration and \(T\) is the period, the following equation holds:

\[D \times T = .2175\]

where \(D\) is expressed as a number of periods. Since, in practice, half-periods are used, the required number \(D'\) of half-periods is:

\[D' = 2D = 2 \times .2175 \times N\]

For example, in the case of the middle C:

\[D = 2 \times .2175 \times 261.62 = 133.8 \approx 114 \text{ decimal (or 72 hexadecimal)}\]

**Exercise 2-2:** Compute the note durations using the equation above, and the frequency table in Figure 2.5 (which needs to be expanded). Verify that they match the numbers in table DURTAB at address 2D1. (See Figure 2.9)
Program Implementation

The program has been structured in two logical parts. The corresponding flowchart is shown in Figure 2.7. The first part of the program is responsible for collecting the notes and begins at label 6.

![Flowchart](image)

Fig. 2.7: Music Flowchart (Continued)
"NUMKEY." (The program is shown in Figure 2.9). The second part
begins at the label "PLAYEM" and its function is to play the stored
notes. Both parts of the program use the PLAYNOTE subroutine
which looks up the note and duration constants, and plays the note.
This routine begins at the label "PLAYIT," and its flowchart is
shown in Figure 2.8.

![Fig. 2.8: PLAYIT Flowchart](image_url)
Fig. 2.9: Music Program (Continued)
The main routines are called, respectively, NXKEY, NUMKEY, and BEEP3 for the note-collecting program, and PLAYEM and DELAY for the note-playing program. Finally, common utility routines are TONE and PLAYIT.

Let us examine these routines in greater detail. The program resides at memory addresses 200 and up. Note that the program, like most others in this book, assumes the availability of the GETKEY routine described in Chapter 1.

The operation of the NXKEY routine is straightforward. The next key closure is obtained by calling the GETKEY routine:

```
START    LDA #0
          STA PILEN   Initialize length of list to 0
          CLC
NXKEY     JSR GETKEY
```

The value read is then compared to the constants “15” and “14” for special action. If no match is found, the constant is stored in the note list using the NUMKEY routine.

```
NXTST    CMP #15
          BNE NXTST
          JSR BEEP3
          BCC START

NXTST    CMP #14
          BNE NUMKEY
          JSR PLAYEM
          CLC
          BCC NXKEY
```

Exercise 2-3: Why are the last two instructions in this routine used instead of an unconditional jump? What are the advantages and disadvantages of this technique?

Every time key number 15 is pressed, a special three-tone routine called BEEP3 is played. The BEEP3 routine is shown at address 0287. It plays three notes in rapid succession to indicate to the user that the notes in the memory have been erased. The erasure is performed by resetting the list length PILLEN to zero. The corresponding routine appears below:

```
BEEP3  LDA #$FF  Beep duration constant
       STA DUR
       LDA #$4B  Code for E2
       JSR TONE
       LDA #$38  Code for D2
       JSR TONE
       LDA #$4B  Code for E2
       JSR TONE
       CLC
       RTS
```

Its operation is straightforward.

The NUMKEY routine will save the code corresponding to the note in the memory. As in the case of a Teletype program, the computer will echo the character which has been pressed in the form of an audible sound. In other words, every time a key has been pressed, the program will play the corresponding note. This is performed by the next two instructions:

```
NUMKEY  STA TEMP
       JSR PLAYIT
```

The list length is then checked for overflow. If an overflow situation is encountered, the player is advised through the use of the three-tone sequence of BEEP3:

```
LDA PILLEN  Get length of list
CMP #$FF  Overflow?
BNE OK  No: add note to list
JSR BEEP3  Yes: warn player
BCC NXKEY  Read next key
```

Otherwise, the new nibble (4 bits) corresponding to the note identification number is shifted into the list:

```
OK       LSR A  Shift low bit into nibble pointer
TAY      Use as byte index
LDA TEMP  Restore key #
```

Note that the nibble-pointer is divided by two and becomes a byte index. It is then stored in register Y, which will be used later to perform
an indexed access to the appropriate byte location within the table (STA TABELG, Y).

Depending on the value which has been shifted into the carry bit, the nibble is stored either in the high end or in the low end of the table’s entry. Whenever the nibble must be saved in the high-order position of the byte, a 4-bit shift to the left is necessary, which requires four instructions:

- **BCS FINBYT** Test if byte has a nibble
- **AND #00000111** Mask high nibble
- **STA TABEG, Y** Save
- **INC PILEN** Next nibble
- **BCC NXKEY**

**FINBYT**

- **ASL A**
- **ASL A**
- **ASL A**
- **ASL A**

Finally, it can be saved in the appropriate table address,

- **ORA TABEG, Y**
- **STA TABEG, Y**

The pointer is incremented and the next key is examined:

- **INC PILEN**
- **BCC NXKEY**

Let us look at this technique with an example. Assume:

- **PILEN = 9** (length of list)
- **TEMP = 6** (key pressed)

The effect of the instructions is:

- **OK**
- **LSR A** A will contain 4, C will contain 1
- **TAY** Y = 4
- **LDA TEMP** A = 6
- **BCS FINBYT** C is 1 and the branch occurs

The situation in the list is:

<table>
<thead>
<tr>
<th>NOTE 1</th>
<th>NOTE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOTE 3</th>
<th>NOTE 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PILEN</th>
<th>EMPTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DISPLACEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Fig. 2.10: Entering a Note in the List

Shift “6” into the high-order position of A:

- **FINBYT**
- **ASL A**
- **ASL A**
- **ASL A**
- **ASL A** A = 60 (hex)

Write A into table:

- **ORA TABEG, Y** A = 16X (where X is the previous nibble in the table)
- **STA TABEG, Y** Restore old nibble with new nibble

The Subroutines

**PLAYEM Subroutine**

The PLAYEM routine is also straightforward. The PTR memory location is used as the running nibble-pointer for the note table. As before, the contents of the running nibble-pointer are shifted to the right and become a byte pointer. The corresponding table entry is then loaded using an indexed addressing method:
Depending upon the value of the bit which has been shifted into the carry, either the high-order nibble or the low-order nibble will be extracted and left-justified in the accumulator. The subroutine PLAYIT described below is used to obtain the appropriate constants and to play the note:

```
FINISH JSR PLAYIT Play note
```

A delay is then implemented between two consecutive notes, the running pointer is incremented, a check occurs for a possible end of list, and the loop is reentered:

```
PLAYEM LDX #0
STX PTR PTR = 0
LDA PTR
LOOP LSR A
TAX
LDA TABEG,X
BCS ENDBYT
AND #%00001111
BCS FINISH
ENDBYT AND #%11110000
LSR A
LSR A
LSR A
LSR A
```

**PLAYIT Subroutine**

The PLAYIT subroutine plays the note or implements a rest, as specified by the nibble passed to it in the accumulator. This subroutine is called "PLAYNOTE" on the program flowchart. It merely looks up the appropriate duration for the note from table DURTAB, and saves it at address DUR (at memory location 4). It then loads the appropriate half-period value from the table at address NOTAB into the 38

```
A register, using indexed addressing, and calls subroutine TONE to play it:

```
PLAYIT CMP #13 Check for a rest
BNZ SOUND No
LDX #$54 Delay = .21 sec (note duration)
JSR DELAY If rest was specified
RTS
SOUND TAX Use key # as index
LDA DURTAB,X To look up duration
STA DUR
LDA NOTAB,X
JSR TONE
RTS
```

**TONE Subroutine**

The TONE subroutine implements the appropriate wave form generation procedure described above, and toggles the speaker at the appropriate frequency to play the specified note. It implements a traditional two-level, nested loop delay, and toggles the speaker by complementing the output port after each specified delay has elapsed:

```
TONE STA FREQ
```

A contains the half-cycle time on entry. It is stored in FREQ. The loop timing will result in an output wave-length of:

```
(20 × A + 26) µs
```

Port B is configured as output:

```
LDA #$FF
STA DDRB
```

Registers are then initialized. A is set to contain the pattern to be output. X is the outer loop counter. It is set to the value DUR which contains the number of half cycles at the time the subroutine is called:

```
LDA #$00
LDX DUR
```

```
38
```

```
MUSIC PLAYER
```

```
39
```
The inner loop counter Y is then initialized to FREQ, the frequency constant:

```
FL2      LDY FREQ
```

and the inner loop delay is generated as usual:

```
FL1      DEY
CLC
BCC .+2
BNE FL1
```

Then the output port is toggled by complementing it:

```
EOR #$FF
STA OPB
```

and the outer loop is completed:

```
DEX
BNE FL2
RTS
```

The DELAY subroutine is shown in Figure 2.9 at memory location 29C and is left as an exercise.

**SUMMARY**

This program uses a simple algorithm to remember and play tunes. All data and constants are stored in tables. Timing is implemented by nested loops. Indexed addressing techniques are used to store and retrieve data. Sound is generated by a square wave.

**EXERCISES**

**Exercise 2-4:** Change the note constants to implement a different range of notes.

**Exercise 2-5:** Store a tune in memory in advance. Trigger it by pressing key "0."

**Exercise 2-6:** Rewrite the program so that it will store the note and duration constants in memory when they are entered, and will not need to look them up when the tune is played. What are the disadvantages of this method?

---

**TRANSLATE**

**THE RULES**

This is a game designed for two competing players. Each player tries to quickly decipher the computer’s coded numbers. The players are alternately given a turn to guess. Each player attempts to press the hexadecimal key corresponding to a 4-bit binary number displayed by the program. The program keeps track of the total guessing time for each player, up to a limit of about 17 seconds. When each player has correctly decoded a number, the players’ response times are compared to determine who wins the turn. The first player to win ten turns wins the match.

The program signals each player’s turn by displaying an arrow pointing either to the left or to the right. The player on the right will be signaled first to initiate the game. The program’s “prompt” is shown in Figure 3.1.

A random period of time will elapse after this prompt, then the bottom row of LEDs on the Games Board will light up. The left-most LED (LED #10) signals to the player to proceed. The four right-most LEDs (LEDs 12, 13, 14, and 15) display the coded binary number. This is shown in Figure 3.2. In this case, player 1 should clearly press key number 5. If the player guesses correctly, the program switches to player 2. Otherwise, player 1 will be given another chance until his or her turn (17 seconds) is up. It should be noted here that for each number presented to the player, the total guessing time is accumulated to a maximum of about 17 seconds. When the maximum is reached, the bottom row will go blank and a new number will be displayed.

The program signals player 2’s turn (the player on the left) by displaying a left arrow on the LEDs as shown in Figure 3.3. Once both players have had a turn to guess a binary digit, the program will signal...
**A TYPICAL GAME**

The right arrow lights up. The following LED pattern appears at the bottom: 10, 13, 14, 15. The player on the right (player 1) pushes key “C,” and the bottom row of LEDs goes blank, as the answer is incorrect. Because player 1 did not guess correctly and he or she still has time left in this turn, a new number is offered to player 1. LEDs 10, 13, 14, and 15 light up and the player pushes key ‘7.’ He or she wins and now the left arrow lights up, indicating that it is player 2’s turn. This time the number proposed is 10, 12, 15. The left player pushes key “9.” At this point, LEDs 10, 11, and 12 light up, indicating that the player is the winner for this turn as he/she has used less total time to make a correct guess than player 1.

Let us try again. The right arrow lights up; the number to translate appears in LEDs 10, 13, 14, and 15. Player 1 pushes key “7,” and a left arrow appears. The next number lights LEDs 10 and 14. Player 2 pushes key “2.” Again, the left-most three LEDs light up at the bottom, as player 2 was faster than player 1 at providing the correct answer.

**THE ALGORITHM**

The flowchart corresponding to the program is shown in Figure 3.4. A first waiting loop is implemented to measure the time that it takes for player 1 to guess correctly. Once player 1 has achieved a correct guess, his or her total time is accumulated in a variable called TEMP. It is then player 2’s turn, and a similar waiting loop is implemented. Once both players have submitted their guesses, their respective guessing times are compared. The player with the least amount of time wins, and control flows either to the left or to the right, as shown by labels 1 and 2 on the flowchart in Figure 3.4. A secondary variable called PLYR1 or PLYR2 is used to count the number of games won by a specific player. This variable is incremented for the player who has won and tested against the value 10. If the value 10 has not been reached, a new game is started. If the value 10 has been reached, the player with this score is declared the winner of the match.

**THE PROGRAM**

The corresponding program uses only one significant data structure. It is called NUMTAB and is used to facilitate the display of the random binary numbers on the LEDs. Remember that LED #10 must always be lit (it is the “proceed” LED). LED #11 must always be off. LEDs 12, 13, 14, and 15 are used to display the binary number. Remember also that bit position 6 of Port 1B is not used. As a result, displaying a “0” will be accomplished by outputting the pattern...
Fig. 3.4: Translate Flowchart

Outputting a “1” will be accomplished with the pattern “00000010.” Outputting “2” will be accomplished with the pattern “10000010.” Outputting “3” will be accomplished with the pattern “00100010.” Outputting “4” will be accomplished with the pattern “10100010,” etc. (See Figure 3.5)

The complete patterns corresponding to all sixteen possibilities are stored in the NUMTAB table of the program. (See Figure 3.6.) Let us examine, for example, entry 14 in the NUMTAB (see line 0060 of the program). It is “00111010.” The corresponding binary number to be displayed is, therefore: “00111.”

It is “1110” or 14. Remember that bit 6 on this port is always “0.”

Low Memory Area

Memory locations 0 to 1D are used to store the temporary variables and the NUMTAB table. The functions of the variables are:

- **TEMP** Storage for random delay-length
- **CNTHI,CNTLO** Time used by a player to make his or her move
- **CNT1H,CNT1L** Time used by player 1 to make his or her move (permanent storage)
- **PLYR1** Score for Player 1 (number of games won so far, up to a maximum of ten)
- **PLYR2** Same for player 2
- **NUMBER** Random number to be guessed
- **SCR and following** Scratch area used by the random number generator

In the assembler listing, the method used to reserve memory locations in this program is different from the method used in the program in Chapter 2. In the MUSIC program, memory was reserved for the variables by simply declaring the value of the symbols representing the
variable locations with the statement:

\[
\text{<VARIABLE NAME>} = \text{<MEMORY ADDRESS>}
\]

In this program, the location counter of the assembler is incremented with expressions of the form:

\[
* = * + n
\]

Thus, the symbols for the variable locations in this program are declared as "labels," while, in the MUSIC program, they are "symbols" or "constant symbols."

The program in this chapter consists of one main routine, called MOVE, and five subroutines: PLAY, COUNTER, BLINK, DELAY, RANDOM. Let us examine them. The data direction registers A and B for the VIA's #1 and #3 of the board must first be initialized. DDR1A, DDR1B, and DDR3B are configured as outputs:

\[
\begin{align*}
\text{START} & \quad \text{LDA} \ #\text{FF} \\
& \quad \text{STA} \ \text{DDR1A} \\
& \quad \text{STA} \ \text{DDR1B} \\
& \quad \text{STA} \ \text{DDR3B}
\end{align*}
\]

DDR3A is conditioned as input:

\[
\begin{align*}
\text{LDA} & \ #0 \\
& \quad \text{STA} \ \text{DDR3A}
\end{align*}
\]

Finally, the variables PLYR1 and PLYR2, used to accumulate the number of wins by each player, are initialized to zero:

\[
\begin{align*}
\text{STA} & \ \text{PLYR1} \\
& \quad \text{STA} \ \text{PLYR2}
\end{align*}
\]

The main body of MOVE is then entered. A right arrow will be displayed to indicate that it is player 2's turn. A reminder of the LEDs connections is shown in Figure 3.5. In order to display a right arrow, LEDs 1, 4, 5, 6, and 7 must be lit (refer also to Figure 3.1). This is accomplished by outputting the appropriate code to Port 1A:

\[
\begin{align*}
\text{MOVE} & \quad \text{LDA} \ #\%01111001 \\
& \quad \text{STA} \ \text{PORT1A} \quad \text{Display right arrow}
\end{align*}
\]
The bottom line of LEDs must be cleared:

```
LDA #0
STA PORT1B
```

Finally, the counters measuring elapsed time must be cleared:

```
STA CNTLO
STA CNTHI
```

We are ready to play:

```
JSR PLAY
```

The PLAY routine will be described below. It returns to the calling routine with a time-elapsed measurement in locations CNTLO and CNTHI.

Let us return to the main program (line 0082 in Figure 3.6). The time-elapsed duration which has been accumulated at locations CNTLO and CNTHI by the PLAY routine is saved in a set of permanent locations reserved for player 1, called CNTIL, CNTHI:

```
LDA CNTLO
STA CNTIL
LDA CNTHI
STA CNTHI
```

It is then player 2’s turn, and a left arrow is displayed. This is accomplished by turning on LEDs 3, 4, 5, and 6:

```
LDA #%000111100  Display left arrow
STA PORT1A
```

Then LED #9 is turned on to complete the left arrow:

```
LDA #1
STA PORT1B
```

As before, the time-elapsed counter is reset to zero:

```
LDA #0
STA CNTLO
STA CNTHI
```
TRANSLATE

译文如下：

程序翻译（续）

Figure 3.6: Translate Program (Continued)
and player 2 can play:

JSR PLAY

The time elapsed for player 2 is then compared to the time elapsed for player 1. If player 2 wins, a branch occurs to PLR2. If player 1 wins, a branch occurs to PLR1. The high bytes are compared first. If they are equal, the low bytes are compared in turn:

```assembly
EQUAL
LDA CNTHI
CMP CNT1H
BEQ EQUAL
BCC PLR2
BCS PLR1
LDA CNTLO
CMP CNT1L
BCC PLR2
BCS PLR1
```

Compare high bytes
Player 2 has lower time?
Player 1 does
Compare low bytes

Once the winner has been identified, the bottom row of LEDs on his or her side will light up, pointing to the winner. Let us follow what happens when PLR1 wins, for example. Player 1's right-most three LEDs (LEDs 13 through 15) are lit up:

PLR1
LDA #11110000
STA PORT1B

The other LEDs on the Games Board are cleared:

LDA #0
STA PORT1A

A DELAY is then implemented, and we get ready to play another game, up to a total of 10:

LDA #$40
JSR DELAY

The score for player 1 is incremented:

INC PLYR1

It is compared to 10. If it is less than 10, a return occurs to the main MOVE routine:

LDA #10
CMP PLYR1
BNE MOVE

Otherwise, the maximum score of 10 has been reached and the game is over. The LEDs on the winner's side will blink:

LDA #11110000  Blink pattern
JSR BLINK
RTS

The corresponding sequence for player 2 is listed at address PLR2 (line 117 on Figure 3.6):

PLR2
LDA #01110
STA PORT1B
LDA #0
STA PORT1A
LDA #$40
JSR DELAY
INC PLYR2
LDA #10
CMP PLYR2
BNE MOVE
LDA #11110
JSR BLINK
RTS

The Subroutines

**PLAY Subroutine**

The PLAY subroutine will first wait for a random period of time before displaying the binary number. This is accomplished by calling the RANDOM subroutine to obtain the random number, then the DELAY subroutine to implement the delay:

PLAY  JSR RANDOM
      JSR DELAY
The RANDOM subroutine will be described below. Another random number is then obtained. It is trimmed down to a value between 0 and 15, inclusive. This will be the binary number displayed on the LEDs. It is stored at location NUMBER:

```
JSR RANDOM
AND #0F
STA NUMBER
```

The NUMTAB table, described at the beginning of this section, is then accessed to obtain the correct pattern for lighting the LEDs using indexed addressing. Register X contains the number between 0 and 15 to be displayed:

```
JSR NUMTAB
```

The pattern in the accumulator is then stored in the output register in order to light the LEDs. Note that the pattern is OR'ed with the previous contents of the output register so that the status of LED 9 is not changed:

```
ORA PORT1B
STA PORT1B
```

Once the random number has been displayed in binary form on the LEDs, the subroutine waits until the player presses a key. The CNTSUB subroutine is used for this purpose:

```
JSR CNTSUB
```

It will be described below. The value returned in register Y by this subroutine is compared to the number to be guessed, which is stored at memory address NUMBER. If the comparison succeeds, exit occurs. Otherwise, all LEDs are cleared using an AND, to prevent changing the status of LED 9, and the subroutine is reentered. Note that the remaining time for the player will be decremented every time the CNTSUB subroutine is called. It will eventually decrement to 0, and this player will be given another number to guess:

```
CPY NUMBER
BEQ DONE
LDA #01
AND PORT1B
STA PORT1B
JMP PLAY
DONE
RTS
```

**Exercise 3-1:** Modify PLAY and/or CNTSUB so that, upon timeout, the player loses the current round, as if the maximum amount of time had been taken to make the guess.

**CNTSUB Subroutine**

The CNTSUB subroutine is used by the PLAY subroutine previously described. It monitors a player's keystroke and records the amount of time elapsed until the key is pressed. The key scanning is performed in the usual way:

```
LDY #$F
STY PORT3B
BIT PORT3A
BPL FINISH
DEY
BPL KEYLP
FINISH
BNE CNTSUB
```

Each time that all keys have been scanned unsuccessfully, the time elapsed counter is incremented (CNTLO,CNTHI):

```
INC CNTLO
BNE CNTSUB
INC CNTHI
BNE CNTSUB
FINISH
RTS
```

Upon return of the subroutine, the number corresponding to the key which has been pressed is contained in index register Y.

**Exercise 3-2:** Insert some "do-nothing" instructions into the CNTSUB subroutine so that the guessing time is longer.
BLINK Subroutine

The LEDs specified by the accumulator contents are blinked (turned on and off) ten times by this subroutine. It uses memory location CNTHI and CNTLO as scratch registers, and destroys their previous contents. Since the LEDs must alternately be turned on and off, an exclusive-OR instruction is used to provide the automatic on/off feature by performing a complementation. Because two complementations of the LED status must be done to blink the LEDs once, the loop is executed 20 times. Note also that LEDs must be kept lit for a minimum amount of time. If the “on” delay was too short, the LEDs would appear to be continuously lit. The program is shown below:

```
BLINK   LDX #20       20 blinks
        STX CNTHI     Blink counter
        STA CNTLO     Blink register
BLOOP   LDA CNTLO     Get blink pattern
        EOR PORT1B    Blink LEDs
        STA PORT1B
        LDA #10       Short delay
        JSR DELAY
        DEC CNTHI     Loop if not done
        BNE BLOOP
        RTS
```

DELAY Subroutine

The DELAY subroutine implements a classic three-level, nested loop design. Register X is set to a maximum value of FF (hexadecimal), and used as the inner loop counter. Register Y is set to the value of 10 (hexadecimal) and used as the level-2 loop counter. Location TEMP contains the number used to adjust the delay and is the counter for the outermost loop. The subroutine design is straightforward:

```
DELAY   STA TEMP
        LDX #$FF
        STA CNTLO
DL1     LDA CNTLO     Blink counter
        STA CNTLO     Blink register
        LDA CNTLO     Get blink pattern
        EOR PORT1B    Blink LEDs
        STA PORT1B
        LDA #10       Short delay
        JSR DELAY
        DEC CNTHI     Loop if not done
        BNE BLOOP
        RTS
```

Exercise 3-3: Compute the exact duration of the delay implemented by this subroutine as a function of the number contained in location TEMP.

RANDOM Subroutine

This simple random number generator returns a semi-random number into the accumulator. A set of six locations from memory address 0008 ("SCR") have been set aside as a scratch-pad for this generator. The random number is computed as 1 plus the contents of the number in location SCR + 1, plus the contents of the number in location SCR + 4, plus the contents of the number in location SCR + 5:

```
RANDOM   SEC
        LDA SCR + 1
        ADC SCR + 4
        ADC SCR + 5
        STA SCR
```

The contents of the scratch area (SCR and following locations) are then shifted down in anticipation of the next random number generation:

```
RNDLP   LDX #4
        LDA SCR,X
        STA SCR + 1,X
        DEX
        BPL RNDLP
        RTS
```

The process is illustrated in Figure 3.7. Note that it implements a seven-location circular shift. The random number which has been computed is written back in location SCR, and all previous values at memory locations SCR and following are pushed down by one position. The previous contents of SCR + 5 are lost. This ensures that the numbers will be reasonably random.
SUMMARY

This game involved two players competing with each other. The time was kept with nested loops. The random number to be guessed was generated by a pseudo-random number generator. A special table was used to display the binary number. LEDs were used on the board to indicate each player's turn to display the binary number, and to indicate the winner.

Exercise 3-4: What happens in the case in which all memory locations from SCR to SCR + 5 were initially zero?

4

HEXGUESS

THE RULES

The object of this game is to guess a secret 2-digit number generated by the computer. This is done by guessing a number, then submitting this number to the computer and using the computer's response (indicating the proximity of the guessed number to the secret number) to narrow down a range of numbers in which the secret number resides. The program begins by generating a high-pitched beep which signals to the player that it is ready for a number to be typed. The player must then type in a two-digit hexadecimal number. The program responds by signaling a win if the player has guessed the right number. If the player has guessed incorrectly, the program responds by lighting up one to nine LEDs, indicating the distance between the player's guess and the correct number. One lit LED indicates that the number guessed is a great distance away from the secret number, and nine lit LEDs indicate that the number guessed is very close to the secret number.

If the guess was correct, the program generates a warbling tone and flashes the LEDs on the board. The player is allowed a maximum of ten guesses. If he or she fails to guess the correct number in ten tries, a low tone is heard and a new game is started.

A TYPICAL GAME

The computer beeps, notifying us that we should type in a guess.

Our guess is: ‘‘40’’
The computer lights 4 LEDs We are somewhat off
Next guess: "C0"
Computer's answer: 3 LEDs We are going further away
Next guess: "20"
Computer's response: 3 The number must be between C0 and 20
Next guess: "80"
Response: 5 We are getting closer
Next guess: "75"
Response: 5 It's not just below 80
Next guess: "90"
Response: 4 We're wandering away
Next guess: "65"
Response: 7 Now we're closing in
Next guess: "60"
Response: 9
Next guess: "5F"
Response: 8
Next guess: "61"
We win!!! All the LEDs flash and a high warbling tone is heard.

THE ALGORITHM

The flowchart for Hexguess is shown in Figure 4.1. The algorithm is straightforward:

— a random number is generated
— a guess is entered
— the closeness of the number guessed to the secret number is evaluated. Nine levels of proximity are available and are displayed by an LED on the board. A closeness or proximity table is used for this purpose.
— a win or a loss is signaled
— more guesses are allowed, up to a maximum of ten.

THE PROGRAM

Data Structures

The program consists of one main routine called GETGES, and two subroutines called LITE and TONE. It uses one simple data structure
— a table called LIMITS. The flowchart is shown in Figure 4.1, and the program listing appears in Figure 4.2.

The LIMITS table contains a set of nine values against which the proximity of the guess to the computer's secret number will be tested. It is essentially exponential and contains the sequence: 1, 2, 4, 8, 16, 32, 64, 128, 200.

Program Implementation

Let us examine the program itself. It resides at memory address 200 and may not be relocated. Five variables reside in page zero:

GUESS is used to store the current guess

GUESS# is the number of the current guess

NUMBER is the secret computer number

As usual, the data direction registers VIA #1 and VIA #3 are conditioned in order to drive the LED display and read the keyboard:

LDA $FF
STA DDR1A OUTPUT
STA DDR1B OUTPUT
STA DDR3B OUTPUT

Memory location DUR is used to store the duration of the tone to be generated by the TONE subroutine. It is initialized to “FF” (hex):

STA DUR

The memory location GUESS# is used to store the number of guesses. It is initialized to 10:

START LDA #$0A
STA GUESS#

The LEDs on the Games Board are turned off:

LDA #00
STA PORT1A
STA PORT1B
GET CLOSENESS COUNTER TO DISTANT

LDA $00

SET CLOSENESS COUNTER TO DISTANT

0246: B0 27

BEC SIGNAL

FRAWMME CLOSENESS OF GUESS TO TABLE OF LIMITS TO SEE HOW MANY LEVELS TO LIGHT.

0248: EB

INDEX

LOOK AT NEXT CLOSENESS LEVEL.

0249: F9 09

CPX $00

WALL NEXT LEVELS TRIED?

024B: F0 06

BNE LOOP 06, TRY NEXT LEVEL.

024D: AP 0B

WIN

LOAD NUMBER OF BLINKS

024F: BS 00

STA GUESS

USE GUESS AS TEMP

0251: AF FF

STA $FF

LIGHT LEDS

0253: BD 01

STA PORTA

0254: BD 00

STA PORTB

0257: A9 32

NOW

LOAD $00

0258: 20 20 02

JSR TONE

MAKE WIN SIGNAL

025E: A9 FF

STA $FF

0260: AD 01

EDR PORTA

0261: AD 00

STA PORTA

0264: BD 00

STA PORTB

0266: C6 00

DEC GUESS

IF BLINKS/TONES DONE?

0268: DD 0E

BNE NOW

AND, DO AGAIN

026B: FF 0E

BEQ START

YES, START NEW GAME.

026F: EE

INDEX

INCREMENT CLOSENESS LEVEL

IF COUNTER TO AT LEAST 1 LED IS LIGHT,

0271: A9 00

LDA $00

CLEAR HIGH LED PORI.

0272: 7D 00

STA PORTB

0275: 20 0E 02

JSR LITE

SET LED PATTERN

0278: B0 01 A0

STA PORTA

0279: 90 05

BCC C6

IF CARRY SET PBO = 1

027B: A9 01

LDA $01

027F: DD 00

STA PORTB

0282: C6 01

CC

DEC GUESS

IF GUESS USED

0284: 10 99

BNE RETIES

LOAD FROM LITE

GET NEXT.

0286: A9 $BE

LDA $BE

LOAD TONE SIGNALS LOST.

0288: 20 96 02

JSR TONE

028A: 4C 06 02

JMP START

NEW GAME.

; ROUTINE TO MAKE PATTERN OF LED LEIS BY SHIFTING A STRING OF ONES TO THE LEFT IN THE ACCUMULATOR UNTIL THE BIT POSITION CORRESPONDING TO THE NUMBER IN X IS REACHED.

; LDA $00

CLEAR ACCUMULATOR FOR PATTERN

; SHFT SEC

MAKE LOW BIT HIGH.

; ROL A

SHIFT IT IN

; BEX

TONE BIT DONE

; BNE SHIFT

If NOT DONE.

; RTS

RETURN

; TONE GENERATION ROUTINE.

; TONE STA FRED

; LDA $400

0290: BB 03

0291: A9 00

0294: A6 02

FL2 LDY FRED

0291: 80

FL1 DEY

029F: 10

02A0: 90

BCC $2

02A2: DD 00

BNE FL1

02A3: 49 FF

02A4: AD 00

02A7: DD 00

02A9: DB 00

02AC: 60

Rts

; TABLE OF LIMITS FOR CLOSENESS LEVELS.

; Fig. 4.2: Hexguess Program (Continued)

The program will generate a random number which must be guessed by the player. A reasonably random number is obtained here by reading the value of timer1 of VIA #1. It is then stored in memory address NUMBER:

LDA TIMER

Low latch of timer 1

STA NUMBER

A random number generator is not required because requests for random numbers occur at random time intervals, unlike the situation in most of the other games that will be described. An important observation on the use of TI CL of a 6522 VIA is that it is often called a "latch" but it is a "counter" when performing a read operation! Its contents are not frozen during a read as they would be with a latch. They are continuously decremented. When they decrement to 0, the counter is reloaded from the "real" latch.

Note that in Figure 4.3 T1L-L is shown twice — at addresses 04 and 06. This is a possible source of confusion and should be clearly understood. Location 4 corresponds to the counter; location 6 corresponds to the latch. Location 4 is read here.

We are ready to go. A high-pitched tone is generated to signal the player that a guess may be entered. The note duration is stored at

HEXGUESS

02AF: AB

LIMITS BYTE 200,128,64,32,16,8,4,2,1

02A9: 80

02AF: 00

02B1: 20

02B0: 10

02B2: 08

02B3: 04

02BA: 02

02B5: 01

SYMBOL TABLE:

0100 TOWER A004 DOLLAR A00X

DORI A002 PORTA A001 PORTB A000

GUES$ 0400 SINK 000C FRE$ 0003

NUMBER 0004 START 0200 NETE$ 021E

ALRIGHT 0241 LOOP 0247 WIN 024B

LOW 0259 SIGNAL 026F CC 0279

LITE 02BE SHIFT 0290 TONE 029A

FL2 029C FL1 029E LIMITS 02A0

X
memory location DUR while the note frequency is set by the contents of the accumulator:

GETGES LDA #$20 High pitch
JSR TONE

Two key strokes must be accumulated for each guess. The GETKEY subroutine is used to obtain the number of the key being pressed, which is then stored in the accumulator. Once the first character has been obtained, it is shifted left by four positions into the high nibble position, and the next character is obtained. (See Figure 4.4.)

Once the second character has been transferred into the accumulator, the previous character, which had been saved in memory location GUESS, is retrieved and OR'ed back into the accumulator:

AND $00001111
ORA GUESS

It is stored back at memory location GUESS:
Now that the guess has been obtained, it must be compared against the random number stored by the computer at memory location NUMBER. A subtraction is performed:

```
LDA NUMBER
SEC
SBC GUESS
```

Note that if the difference is negative, it must be complemented:

```
BCS ALRIGHT
EOR %1111111
SEC
ADC #0
```

Positive?

It is negative: complement

Make it two’s complement

Add one

Once the “distance” from the guess to the actual number has been computed, the “closeness-counter” must be set to a value between 1 and 9 (only nine LEDs are used). This is done by a loop which compares the absolute “distance” of the guess from the correct number to a bracket value in the LIMITS table. The number of the appropriate bracket value becomes the value assigned to the proximity or closeness of the guessed number to the secret number. Index register X is initially set to 0, and the indexed addressing mode is used to retrieve bracket values. Comparisons are performed as long as the “distance” is less than the bracket value, or until X exceeds 9, i.e., until the highest table value is looked up.

```
ALRIGHT
LOOP
LDX #0
CMP LIMITS,X
BCS SIGNAL
INX
CPX #9
BNE LOOP
```

At this point, unless a branch has occurred to SIGNAL, the distance between the guess and the actual number is 0: it is a win. This is signaled by blinking the LEDs and by generating a special win tone:

```
WIN
LDA #11
STA GUESS
LDA #FF
```

The blinking is generated by complementing the LEDs repeatedly:

```
LDA #$FF
EOR PORT1A
STA PORT1A
STA PORT1B
```

The loop is executed again:

```
DEC GUESS
BNE WOW
```

Finally, when the loop index (GUESS) reaches zero, a branch occurs back to the beginning of the main program: START:

```
BEQ START
```

If, however, the current guess is not correct, a branch to SIGNAL occurs during bracket comparison, with the contents of the X register being the proximity value: i.e., the number of LEDs to light. Depending on the closeness of the guess to the secret number, LEDs #1 to #9 will be turned on:

```
SIGNAL
INX
LDA #0
STA PORT1B
JSR LITE
STA PORT1A
BCC CC
LDA #01
STA PORT1B
```

The number of LEDs to turn on is in X. It must be converted into the appropriate pattern to put on the output port. This is done by the LITE subroutine, described below.

If LED #9 is to be turned on, the carry bit is set by LITE. An ex-
Explicit test of the carry for this case is done above (the pattern 01 is then sent to PORT1B). The number of the current guess is decremented next. If it is 0, the player has lost; the lose signal is generated and a new game is started; otherwise, the next guess is obtained:

<table>
<thead>
<tr>
<th></th>
<th>DEC GUESS#</th>
<th>BNE GETGES</th>
<th>Any guesses left?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDA #$BE</td>
<td>JSR TONE</td>
<td>Low tone</td>
</tr>
<tr>
<td></td>
<td>JMP START</td>
<td></td>
<td>New game</td>
</tr>
</tbody>
</table>

### The Subroutines

#### LITE Subroutine

The LITE subroutine will generate the pattern required to light up LEDs #1 to #8, depending on the number contained in register X. The required "1" bits are merely shifted right in the accumulator as register X is being decremented. An example is given in Figure 4.5.

Upon exit from the subroutine, the accumulator contains the correct pattern required to light up the specified LEDs. If LED #9 is included, the pattern would consist of all ones, and the carry bit would be set:

#### TONE Subroutine

The TONE subroutine will generate a tone for a duration specified by a constant in memory location DUR, at the frequency specified by the contents of the accumulator. Index register Y is used as the inner loop counter. The tone is generated, as usual, by turning the speaker connected to PORT3B on and off successively during the appropriate period of time:

<table>
<thead>
<tr>
<th></th>
<th>STA FREQ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDA #$00</td>
</tr>
<tr>
<td></td>
<td>LDX DUR</td>
</tr>
<tr>
<td>FL2</td>
<td>LDY FREQ</td>
</tr>
<tr>
<td>FL1</td>
<td>DEY</td>
</tr>
</tbody>
</table>
SUMMARY
This time, the program used the timer's latch (i.e., a hardware register) rather than a software routine as a random number generator. A simple "LITE" routine was used to display a value, and the usual TONE routine was used to generate a sound.

EXERCISES
Exercise 4-1: Improve the Hexguess program by adding the following feature to it. At the end of each game, if the player has lost, the program will display [the number which the player should have guessed] for approximately 3 seconds, before starting a new game.

Exercise 4-2: What would happen if the SEC at location 290 hexadecimal were left out?

Exercise 4-3: What are the advantages and disadvantages of using the timer's value to generate a random number? What about the successive numbers? Will they be related? Identical?

Exercise 4-4: How many times does the above program blink the lights when it signals a win?

Exercise 4-5: Examine the WIN routine (line 24D). Will the win tone be sounded once or several times?

Exercise 4-6: What is the purpose of the two instructions at addresses 29F and 2A0? (Hint: read Chapter 2.)

Exercise 4-7: Should the program start the timer?

Exercise 4-8: Is the number of LEDs lit in response to a guess linearly related to the closeness of a guess?

5
MAGIC SQUARE

THE RULES
The object of the game is to light up a perfect square on the board, i.e., to light LEDs 1, 2, 3, 6, 9, 8, 7, and 4 but not LED #5 in the center.

The game is started with a random pattern. The player may modify the LED pattern on the board through the use of the keyboard, since each of the keys complements a group of LEDs. For example, each of the keys corresponding to the corner LED positions (key numbers: 1, 3, 9, and 7) complements the pattern of the square to which it is attached. Key #1 will complement the pattern formed by LEDs 1, 2, 4, 5. Assuming that LEDs 1, 2, and 4 are lit, pressing key #1 will result in the following pattern: 1-off, 2-off, 4-off, 5-on.

The pattern formed by LEDs 1, 2, 4, and 5 has been complemented and only LED #5 is lit after pressing key #1. Pressing key #1 again will result in: 1, 2, and 4-on with 5-off. Pressing a key twice results in two
successive complementations, i.e., it cancels out the first action.

Similarly, key #9 complements the lower right-hand square formed by LEDs 5, 6, 8, and 9.

Key #3 complements the pattern formed by LEDs 2, 3, 5, and 6.
Key #7 complements the pattern formed by LEDs 4, 5, 7, and 8.

The “edge keys” corresponding to LEDs 2, 4, 6, and 8 complement the pattern formed by the three LEDs of the outer edge of which they are a part. For example, pressing key #2 will complement the pattern for LEDs 1, 2, and 3. Assume an initial pattern with LEDs 1, 2, and 3 lit. Pressing key #2 will result in obtaining the complemented pattern, i.e., turning off all three LEDs. Similarly, assume an initial pattern on the left vertical edge where LEDs 4 and 7 are lit.

Pressing key #4 will result in a pattern where LED #1 is lit and LEDs 4 and 7 are turned off.

Pressing key #5 will result in lighting up LEDs 2, 4, and 5:

The winning combination in which all LEDs on the edge of the square are lit is obtained by pressing the appropriate sequence of keys.

Finally, pressing key #5 (the center LED position) will result in complementing the pattern formed by LEDs 2, 4, 5, 6, and 8. For example, assume the following initial pattern where only LEDs 6 and 8 are lit:

Pressing key #5 will result in lighting up LEDs 2, 4, and 5:

The winning combination in which all LEDs on the edge of the square are lit is obtained by pressing the appropriate sequence of keys.
The mathematical proof that it is always possible to achieve a "win" is left as an exercise for the reader. The program confirms that the player has achieved the winning pattern by flashing the LEDs on and off.

Key "0" must be used to start a new game. A new random pattern of lit LEDs will be displayed on the board. The other keys are ignored.

A TYPICAL GAME

Here is a typical sequence:
The initial pattern is: 1-3-4-6-9.

Move: press key #8.
The resulting pattern is: 1-3-4-6-7-8.

Next move: press key #3.
The resulting pattern is: 3-4-5-7-8.

Next move: press key #2.
The resulting pattern is: 1-2-4-5-7-8.

Next move: press key #2.
The resulting pattern is: 2-4-6-7-8.
Next move: press key #6.
The resulting pattern is 1-2-3-4-5-6-7-8-9.

Note that this is a "classic" pattern in which all LEDs on the board are lit. It is not a winning situation, as LED #5 should be off. Let us proceed.

Next move: the end of this game is left to the mathematical talent of the reader. The main purpose was to demonstrate the effect of the various moves.

Hint: a possible winning sequence is 2-4-6-8-5!

General advice: in order to win this game, try to arrive quickly at a symmetrical pattern on the board. Once a symmetrical pattern is obtained, it becomes a reasonably simple matter to obtain the perfect square. Generally speaking, a symmetrical pattern is obtained by hitting the keys corresponding to the LEDs which are off on the board but which should be "on" to complete the pattern.

THE ALGORITHM

A pattern is generated on the board using random numbers. The key corresponding to the player's move is then identified, and the appropriate group of LEDs on the board is complemented.

A table must be used to specify the LEDs forming a group for each key.

The new pattern is tested against a perfect square. If one exists, the player wins. Otherwise, the process begins anew.

The detailed flowchart is shown in Figure 5.1.
THE PROGRAM

Data Structures

The main problem here is to devise an efficient way to complement the correct LED pattern whenever a key is pressed. The complementation itself may be performed by an Exclusive-OR instruction. In this case, the pattern used with the EOR instruction should contain a "1" in each LED position which is to be complemented, and "0"s elsewhere. The solution is quite simple: a nine-entry table, called TABLE, is used. Each table entry corresponds to a key and has 16 bits of which only nine are used inasmuch as only nine LEDs are used. Each of the nine bits contains a "1" in the appropriate position, indicating the LED which will be affected by the key.

For example, we have seen that key number 1 will result in complementing LEDs 1, 2, 4, and 5. The corresponding table entry is therefore: 0, 0, 0, 1, 1, 0, 1, 1, 0, where bits 1, 2, 4, and 5 (starting the numbering at 1, as with the keys) have been set to "1." Or, more precisely, using a 16-bit pattern:

0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1

The complete table appears below in Figure 5.2.

<table>
<thead>
<tr>
<th>KEY</th>
<th>PATTERN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00011011</td>
</tr>
<tr>
<td>2</td>
<td>00000111</td>
</tr>
<tr>
<td>3</td>
<td>00110110</td>
</tr>
<tr>
<td>4</td>
<td>01000101</td>
</tr>
<tr>
<td>5</td>
<td>10111000</td>
</tr>
<tr>
<td>6</td>
<td>00010001</td>
</tr>
<tr>
<td>7</td>
<td>01100000</td>
</tr>
<tr>
<td>8</td>
<td>11000000</td>
</tr>
<tr>
<td>9</td>
<td>10110000</td>
</tr>
</tbody>
</table>

Fig. 5.2: Complementaton Table

Program Implementation

A random pattern of LEDs must be lit on the board at the beginning of the game. This is done, as in the previous chapter, by reading the value of the VIA #1 timer. If a timer were not available, a random number-generating routine could be substituted.
The data direction registers for Ports A and B of the VIA are configured for output to drive the LEDs:

```
LDA #$FF
STA DDRA
STA DDDB
```

The “random” numbers are then obtained by reading the value of timer 1 of the VIA and are used to provide a random pattern for the LEDs. (Two numbers provide 16 bits, of which 9 are kept.)

```
START LDA TICL
STA PORT1
LDA TICL
AND #01
STA PORT2
```

An explanation of the use of TICL has been presented in the previous chapter. The program then monitors the keyboard for the key stroke of the player. It will accept only inputs “0” through “9” and will reject all others:

```
KEY JSR GETKEY
CMP #0
BEQ START
CMP #10
BPL KEY
```

If the player has pressed key “0,” the program is restarted with a new LED display. If it is a value between “1” and “9” that is pressed, the appropriate change must be performed on the LED pattern. The key number will be used as an index to the table of complementation codes. Since the keys are labeled 1 through 9, the key number must first be decremented by 1 in order to be used as an index. Since the table contains double-byte entries, the index number must also be multiplied by 2. This is performed by the following three instructions:

```
SEC
SBC #1
ASLA
```

---

Fig. 5.3: Magic Square Program (Continued)
Remember that a shift left is equivalent to a multiplication by 2 in the binary system. The resulting value is used as an index and stored in index register X:

\[ \text{TAX} \]

The LED pattern is stored in the Port A data registers. It will be complemented by executing an EOR instruction on Port 1, then repeating the process for Port 2:

\[ \text{LDA \ PORT1} \]
\[ \text{EOR \ TABLE,} \ X \text{ Complement Port1} \]
\[ \text{STA \ PORT1} \]
\[ \text{LDA \ PORT2} \]
\[ \text{EOR \ TABLE + 1,} \ X \text{ Same for Port2} \]
\[ \text{AND \ #01} \]
\[ \text{Mask out unused bits} \]
\[ \text{STA \ PORT2} \]

Note that assembly-time arithmetic is used to specify the second byte in the table:

\[ \text{EOR \ TABLE + 1,} \ X \]

Once the pattern has been complemented, the program checks for a winning pattern. To do so, the contents of Port 2 and Port 1 must be matched against the correct LED pattern. For Port 2, this is "0, 0, 0, 0, 0, 0, 0, 1." For Port 1, this is "1, 1, 1, 0, 1, 1, 1, 1." Bit 0 of Port 2 happens presently to be contained in the accumulator and can be tested immediately after a right shift:

\[ \text{LSR \ A} \]
\[ \text{Shift bit 0 of Port 2} \]
\[ \text{BCC \ KEY} \]

The contents of Port 1 must be explicitly compared to the appropriate pattern:

\[ \text{LDA \ PORT1} \]
\[ \text{CMP \ #} \text{111101111} \]
\[ \text{BNE \ KEY} \]

To confirm the win, LEDs are now blinked on the board. TEMP is used as a counter variable; X is used to set the fixed delay duration. Y is used as a counter for the innermost loop. Each port is complemented after the delay has elapsed.

\[ \text{LDA \ #14} \]
\[ \text{STA \ TEMP} \]
\[ \text{BLINK} \]
\[ \text{LDX \ #$20} \]
\[ \text{DELAY} \]
\[ \text{LDY \ #$FF} \]
\[ \text{DLY} \]
\[ \text{NOP} \]
\[ \text{BNE \ .+2} \]
\[ \text{DEY} \]
\[ \text{BNE \ DLY} \]
\[ \text{DEX} \]
\[ \text{BNE \ DELAY} \]
\[ \text{LDA \ PORT1} \]
\[ \text{EOR \ #$FF} \]
\[ \text{STA \ PORT1} \]
\[ \text{LDA \ PORT2} \]
\[ \text{EOR \ #1} \]
\[ \text{STA \ PORT2} \]
\[ \text{DEC \ TEMP} \]
\[ \text{BNE \ BLINK} \]
\[ \text{BEQ \ KEY} \]

**SUMMARY**

This game of skill required a special table to perform the various complementations. The timer is used directly to provide a pseudo-random number, rather than a program. The LED pattern is stored directly in the I/O chip's registers.

**EXERCISES**

**Exercise 5-1:** Rewrite the end of the program using a delay subroutine.

**Exercise 5-2:** Will the starting pattern be reasonably random?
Exercise 5-3: Provide sound effects.

Exercise 5-4: Allow the use of key "A" to perform a different change such as a total complementation.

Exercise 5-5 (more difficult): Write a program which allows the computer to play and win.

Exercise 5-6: Add to the previous exercise the following feature: record the number of moves played by the computer, then play against the computer. You must win in fewer moves. You may specify an identical starting pattern for yourself and the computer. In this case, you should start, then let the computer "show you." If the computer requires more moves than you do, you are either an excellent player, a lucky player, or you are a poor programmer. Perhaps you are using the wrong algorithm!

6

SPINNER

THE RULES

A light spins around the square formed by LEDs 1, 2, 3, 6, 9, 8, 7, and 4, in a counterclockwise fashion.

The object of the game is to stop the light by hitting the key corresponding to the LED at the exact time that the LED lights up. Every time that the spinning light is stopped successfully, it will start spinning at a faster rate. Every time that the player fails to stop the LED within 32 spins, the light will stop briefly on LED #4, then resume spinning at a slower pace. The expert player will be able to make the light spin faster and faster, until the maximum speed is reached. At this point, all the LEDs on the Games Board (LEDs 1 through 15) light up simultaneously. It is a win, and a new game is started.

Each win is indicated to the player by a hesitation of the light on the LED corresponding to the key pressed. When a complete game is won, all LEDs on the Games Board will be lit.
This game can also be used to sharpen a player's reflexes, or to test his or her reaction time. In some cases, a player's reaction may be too slow to catch the rotating LED even at its slowest speed. In such a case, the player may be authorized to press two or even three consecutive keys at once. This extends the player's response time. For example, with this program, if the player would press keys 7, 8, and 9 simultaneously, the light would stop if it was at any one of those positions (7, 8, or 9).

THE ALGORITHM

The flowchart is presented in Figure 6.1. The game may operate at eight levels of difficulty, corresponding to the successive speeds of the "blip" traveling with increased rapidity around the LED square. An 8-bit counter register is used for two functions simultaneously. (See Figure 6.2.) The lower 3 bits of this register are used as the "blip-counter" and point to the current position of the light on the LED square. Three bits will select one of eight LEDs. The left-most 5 bits of this register are used as a "loop-counter" to indicate how many times the blip traverses the loop. Five bits allow up to 32 repetitions. LEDs are lit in succession by incrementing this counter. Whenever the blip-counter goes from "8" to "0," a carry will propagate into the loop-counter, incrementing it automatically. Allocating the 8 bits of register Y to two different conceptual counters facilitates programming. Another convention could be used.

Every time that an LED is lit, the keyboard is scanned to determine whether the corresponding key has been pressed. Note that if the key was pressed prior to the LED being lit, it will be ignored. This is accomplished with an "invalid flag." Thus, the algorithm checks to see whether or not a key was initially depressed and then ignores any further closures if it was. A delay constant is obtained by multiplying the difficulty level by four. Then, during the delay while the LED is lit, a new check is performed for a key closure if no key had been pressed at the beginning of this routine. If a key had been pressed at the beginning it will be treated as a miss, and the program will not check again to see if the key was pressed as the "invalid flag" will have been set.

Every time the correct key is pressed during the delay while the LED is on (left branch of the flowchart in the middle section of Figure 6.1), the value of the difficulty level is decremented (a lower difficulty number results in a higher rotation speed). For every miss on the part of the player, the difficulty value is incremented up to 15, resulting in a slower spin of the light. Once a difficulty level of 0 has been reached, if a hit is recorded, all LEDs on the board will light to acknowledge the situation.

THE PROGRAM

Data Structures

The program uses two tables. The KYTBL table stores the key numbers corresponding to the circular LED sequence: 1, 2, 3, 6, 9, 8, 7, 4. It is located at memory addresses 0B through 12. See the program listing in Figure 6.3.

The second table, LTABLE, contains the required bit patterns which must be sent to the VIA's port to illuminate the LEDs in sequence. For example, to illuminate LED #1, bit pattern "00000001" or 01 hexadecimal, must be sent. For LED #2, the bit pattern "00000100" must be sent, or 02 hexadecimal. Similarly, for the other LEDs, the required pattern is: 04, 20, 00, 80, 40; 0B in hexadecimal.

Note that there is an exception for LED #9. The corresponding pattern is "0" for Port 1, and bit 0 of Port 2 must also be turned on. We will need to check for this special situation later on.

Program Implementation

Three variables are stored in memory page 0:

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DURAT</td>
<td>Is the delay between two successive LED illuminations</td>
</tr>
<tr>
<td>DIFCLT</td>
<td>Is the &quot;difficulty level&quot; (reversed)</td>
</tr>
<tr>
<td>DNTST</td>
<td>Is a flag used to detect an illegal key closure when scanning the keys</td>
</tr>
</tbody>
</table>

As usual, the program initializes the three required data direction registers: DDR1 on both Port A and Port B for the LEDs, and DDR3B for the keyboard:

START:

LDA #$FF
STA DDR1A
STA DDR1B
STA DDR3B
Fig. 6.1: Spinner Flowchart

1. **START**
   - **DIFFICULTY = 8**
   - **COUNTER = 0**
   - Use bits 0 - 2 of counter to look up LED pattern in table, then display pattern.

2. **KEY DOWN?**
   - **NO**
     - **DISPLAY PATTERN**
     - **OUTPUT NUMBER OF KEY TO LOOK FOR (BITS 0 - 2 OF COUNTER) TO KEYBOARD**
   - **YES**
     - **CLEAR INVALID FLAG**
     - **SET INVALID FLAG**
     - **DURAT = 128**
     - **DELAY CONST = 4 x DIFFICULTY**
     - **DELAY ACCORDING TO DELAY CONST**

3. **PAUSE**
   - **LIGHT ALL LEDs**
   - **COUNTER = COUNTER + 1**
   - **DURAT = DURAT + 1**
   - **DIFFICULTY = DIFFICULTY - 1**
   - **COUNTDOWN = 0 (FROM OVERFLOW?)**

4. **PAUSE**
   - **INCREMENT DIFFICULTY, MAKING SURE IT DOES NOT EXCEED 15**

5. **PAUSE**
   - **INVALID FLAG SET?**
   - **YES**
     - **DIFFICULTY = DIFFICULTY - 1**
     - **DURAT = DURAT - 1**
   - **NO**
     - **DIFFICULTY = 07**

Fig. 6.1: Spinner Flowchart (Continued)
The difficulty level is set to 8, an average value:

```
LDA #8
STA DFCLT
```

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```

Fig. 6.2: Dual Counter

The difficulty level is set to 8, an average value:

```
LDA #8
STA DFCLT
```

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```

Fig. 6.2: Dual Counter

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```

Fig. 6.2: Dual Counter

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```

Fig. 6.2: Dual Counter

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```

Fig. 6.2: Dual Counter

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```

Fig. 6.2: Dual Counter

The keystrope port is conditioned for input:

```
STA DDR3A
```

The Y register, to be used as our generalized loop-plus-blip-counter, is set to ‘0’:

```
NWGME LDY #0
```

The key-down indicator is also set to ‘0’:

```
LOOP LDA #0
STA DNTST
```

LED #9 is cleared:

```
STA PORT1B
```

The lower 3 bits of the counter are extracted. They contain the blip-counter and are used as an index into the LED pattern table:

```
TYA AND #$07
```

The pattern is obtained from LTABLE, using an indexed addressing:

```
LDA LTABLE, X
```
As we indicated in the previous section, an explicit check must be made for the pattern “0,” which requires that bit 0 of Port B be turned on. This corresponds to LED #9:

```
BNE CHECK
LDA #1
STA PORT1B
```

Once the correct LED has been lit, the keyboard must be inspected to determine whether the player has already pressed the correct key. The program only checks the key number corresponding to the LED being lit:

```
CHECK
LDA KTBL,X
STA PORT1A
```

If the corresponding key is down (a strobe high on Port 3A is detected), the key-down flag, DNTST, is set to “1”:

```
INVALID
STA DTNST
```

This is an illegal key closure. It will be ignored. A delay to keep the LED lit is implemented by loading a value in memory location DURAT. This location is used as a loop-counter. It will be decremented later on and will cause a branch back to location DL1 to occur:

```
DELAY
LDA #80
STA DURAT
```

The difficulty counter, DIFCLT, is then multiplied by four. This is accomplished by two successive left shifts:
The result is saved in index register X. It will determine the delay length. The lower the “difficulty-level,” the shorter the delay will be.

The delay loop is then implemented:

```
DL1  LDA DIFCLT
     ASL A
     ASL A
     TAX
```

The key-down flag, DNTST, is then retrieved from memory and tested. If the key was down at the beginning of this routine, the program branches to location NOTST. Otherwise, if a closure is detected, a hit is reported and a branch occurs to location HIT:

```
LDA DNTST
BNE NOTST
BIT PORT3A
BPL HIT
```

At NOTST, the external delay loop proceeds: the value of DURAT is decremented and a branch back to location DL1 occurs, unless DURAT decrements to “0.” Whenever the delay decrements to “0” without a hit, the main counter (register Y) is incremented by 1. This results in advancing the blip-counter (lower three bits of register Y) to the next LED. However, if the blip-counter was pointing to LED #4 (the last one in our sequence), the loop-counter (upper 5 bits of register Y) will automatically be incremented by 1 when the blip-counter advances. If the value 32 is reached for the loop-counter, the value of register Y after incrementation will be “0” (in fact, an overflow will have occurred into the carry bit). This condition is tested explicitly:

```
NOTST  DEC DURAT
       BNE DL1
       INY
       BNE LOOP
```
6502 GAMES

JSR WAIT
JMP START

The pause is achieved with the usual delay subroutine called "WAIT."
It is a classic, two-level nested loop delay subroutine, with additional
do-nothing instructions inserted at address 0286 to make it last longer:

WAIT
LP1
LP2
LDY #$FF
LDX #$FF
ROR DURAT
ROL DURAT
ROR DURAT
ROL DURAT
DEX
BNE LP2
DEY
BNE LP1
RTS

SUMMARY

This program implemented a game of skill. Multiple levels of difficulty were provided in order to challenge the player. Since human reaction time is slow, all delays were implemented as delay loops. For efficiency, a special double-counter was implemented in a single register: the blip counter—loop counter.

EXERCISES

Exercise 6-1: There are several ways to "cheat" with this program. Any given key can be vibrated rapidly. Also, it is possible to press any number of keys simultaneously, thereby massively increasing the odds. Modify the above program to prevent these two possibilities.

Exercise 6-2: Change the rotation speed of the light around the LEDs by modifying the appropriate memory location. (Hint: this memory location has a name indicated at the beginning of the program.)

Exercise 6-3: Add sound effects.

7

SLOT MACHINE

THE RULES

This program simulates a Las Vegas-type slot machine. The rotation of the wheels on a slot machine is simulated by three vertical rows of lights on LED columns 1-4-7, 2-5-8, and 3-6-9. The lights "rotate" around these three columns, and eventually stop. (See Figure 7.1.) The final light combination representing the player's score is formed by LEDs 4-5-6, i.e., the middle horizontal row.

At the beginning of each game, the player is given eight points. The player's score is displayed by the corresponding LED on the Games Board. At the start of each game, LED #8 is lit, indicating this initial score of 8.

The player starts the slot machine by pressing any key. The lights start spinning on the three vertical rows of LEDs. Once they stop, the combination of lights in LEDs 4, 5, and 6 determines the new score. If either zero or one LED is lit in this middle row, it is a lose situation, and the player loses one point. If two LEDs are lit in the middle row, the player's score is increased by one point. If three LEDs are lit in the middle row, three points are added to the player's score.

Whenever a total score of zero is obtained, the player has lost the game. The player wins the game when his or her score reaches 16 points. Everything that happens while the game is being played produces tones from the machine. While the LEDs are spinning, the speaker crackles, reinforcing the feeling of motion. Whenever the lights stop rotating, a tone sounds in the speaker, at a high pitch if it is a win situation, or at a low pitch if it is a lose situation. In particular, after a player takes his or her turn, if there are three lights in the mid-
dle row (a win situation), the speaker will go beep-beep-beep in a high pitch, to draw attention to the fact that the score is being incremented by three points. Whenever the maximum of 16 points is reached, the player has obtained a "jackpot." At this point all the LEDs on the board will light up simultaneously, and a siren sound will be generated (in ascending tones). Conversely, whenever a null score is reached a siren will be sounded in descending tones.

Note that, unlike the Las Vegas model, this machine will let you win frequently! Good luck. However, as you know, it is not as much a matter of luck as it is a matter of programming (as in Las Vegas machines). You will find that both the scoring and the probabilities can be easily modified through programming.

A TYPICAL GAME

The Games Board initially displays a lit LED in position 8, indicating a starting score of 8. At this point the player should select and press a key. For this example let's press key 0. The lights start spinning. At the end of this spin, LEDs 4, 5, and 9 are lit. (See Figure 7.2.) This is a win situation and one point will be added to the score. The high-pitch tone sounds. LED #9 is then lit to indicate the total of the 8 previous points plus the one point obtained on this spin.

Key 0 is pressed again. This time only LED 5 in the middle row is lit after the spin. The score reverts back to 8. (Remember, the player loses 1 point from his or her score if either zero or only one LED in the middle row is lit after the spin.)

Key 0 is pressed again; this time LEDs 5 and 6 light up resulting in a score of nine.

Key 0 is pressed again. LED 4 is lit at the end of the spin, and LED 8 lights up again.

Key 0 is pressed. LED 6 is lit. The score is now 7, etc.

THE ALGORITHM

The basic sequencing for the slot machine program is shown in the flowchart in Figure 7.3. First, the score is displayed, then the game is started by the player's key stroke and the LEDs are spun. After this, the results are evaluated: the score is correspondingly updated and a win or lose situation is indicated.

The LED positions in a column are labeled 0, 1, 2, from the top to bottom. LEDs are spun by sequentially lighting positions 0, 1, 2, and then returning to position 0. The LEDs continue to spin in this manner and their speed of rotation diminishes until they finally come to a stop. This effect is achieved by incrementing the delay between each successive actuation of an LED within a given column. A counter-register is associated with each "wheel," or column of three LEDs. The initial contents of the three counters for wheels 1, 2, and 3 are obtained from a random number generator. In order to influence the odds, the random number must fit within a programmable bracket called (LOUM, HIUM). The value of this counter is transferred to a temporary memory location. This location is regularly decremented until it reaches the value "0." When the value 0 is reached, the next LED on
the "wheel" is lit. In addition, the original counter contents are incremented by one, resulting in a longer delay before lighting up the next LED. Whenever the counter overflows to 0, the process for that wheel stops. Thus, by using synchronous updating of the temporary memory locations, the effect of asynchronously moving LED "blips" is achieved. When all LEDs have stopped, the resulting position is evaluated.

The flowchart corresponding to this DISPLAY routine is shown in Figure 7.4. Let us analyze it. In steps 1, 2, and 3 the LED pointers are initialized to the top row of LEDs (position 0). The three counters used to supply the timing interval for each wheel are filled with numbers from a random number generator. The random number is selected between set limits. Finally, the three counters are copied into the temporary locations reserved for decrementing the delay constants.

Let us examine the next steps presented in Figure 7.4:

4. The wheel pointer X is set at the right-most column: \( X = 3 \).
5. The corresponding counter for the current column (column 3 this time) is tested for the value 0 to see if the wheel has stopped. It is not 0 the first time around.
6,7. The delay constant for the column of LEDs determined by the wheel pointer is decremented, then it is tested against the value 0. If the delay is not 0, nothing else happens for this column, and we move to the left by one column position:
16. The column pointer X is decremented: \( X = X - 1 \)
17. X is tested against zero. If X is zero, a branch occurs to step 5. Every time that X reaches the value zero, the same situation may have occurred in all three columns. All wheel counters are, therefore, tested for the value zero.
18. If all counters are zero, the spin is finished and exit occurs. If all counters are not zero, a delay is implemented, and a branch back to (4) occurs.

Back to step 7:

7. If the delay constant has reached the value zero, the next LED down in the column must be lit.
8. The LED pointer for the wheel whose number is in the wheel pointer is incremented.
9. The LED pointer is tested against the value 4. If 4 has not been reached, we proceed; otherwise, it is reset to the value 1. (LEDs are designated externally by positions 1, 2, and 3 from...
Fig. 7.4: DISPLAY Flowchart

1. LED POINTERS = 0
2. FILL COUNTERS WITH RANDOM NUMBERS BETWEEN LOLIM & HILIM
3. COPY COUNTERS TO CORRESPONDING DELAY CONSTANT LOCATIONS
4. X = 3
5. COUNTER (X) = 0?
6. DECREMENT DELAY CONSTANT (X)
7. DELAY CONSTANT (X) = 0?
8. INCREMENT LED POINTER (X)
9. YES
10. TEM" = 2 X LED POINTER (X)
11. LTMSK (X) = LIGHTABLE (TEMP, X)
12. INCREMENT COUNTER (X)
13. COPY COUNTER (X) INTO DELAY CONSTANT (X)
14. OUTPUT (LTMSK1 OR LTMSK2 OR LTMSK3) TO LEDs
15. TOGGLE SPEAKER
16. YES

Fig. 7.4: DISPLAY Flowchart (Continued)

16. YES
17. X = 0?
18. NO
19. ALL COUNTERS = 0?
20. YES
21. DELAY A BIT
22. DONE: RETURN
23. NO

Fig. 7.4: DISPLAY Flowchart (Continued)
top to bottom. The next LED to be lit after LED #3 is LED #1.

10, 11. The LED must be lit on the board, and a table LIGHTABLE is utilized to obtain the proper pattern.

12. The counter for the appropriate wheel is incremented. Note that it is not tested against the value zero. This will occur only when the program moves to the left of wheel 1. This is done at location 18 in the flowchart, where the counters are tested for the value zero.

13. The new value of the counter is copied into the delay constant location, resulting in an increased delay before the next LED actuation.

14. The current lighting patterns of each column are combined and displayed.

15. As each LED is lit in sequence, the speaker is toggled (actuated).

16. As usual, we move to the column on the left and proceed as before.

Let us go back to the test at step 5 in the flowchart:

5. Note that whenever the counter value for a column is zero, the LED in that column has stopped moving. No further action is required. This is accounted for in the flowchart by the arrow to the right of the decision box at 5: the branch occurs to 16 and the column pointer is decremented, resulting in no change for the column whose counter was zero.

Next, the evaluation algorithm must evaluate the results once all LEDs have stopped and then it must signal the results to the player. Let us examine it.

The Evaluation Process

The flowchart for the EVAL algorithm is shown in Figure 7.5. The evaluation process is also illustrated in Figure 7.6, which shows the nine LEDs and the corresponding entities associated with them. Referring to Figure 7.6, X is a row-pointer and Y is a column- or wheel-pointer. A value counter is associated with each row. It contains the total number of LEDs lit in that row. This value counter will be converted into a score according to specific rules for each row. So far, we have only used row 2 and have defined a winning situation as being one in which two or three LEDs were lit in that row. However, many other combinations are possible and are allowed by this mechanism.

Exercises will be suggested later for other winning patterns.

The total for all of the scores in each row is added into a total called SCORE, shown at the bottom right-hand corner of Figure 7.6.

Let us now refer to the flowchart in Figure 7.5. The wheel- or column pointer Y is set initially to the right-most column: Y = 3.

2. The temporary counters are initialized to the value zero.

3. Within the current column (3), we need only look at the row which has a lit LED. This row is pointed to by LED-POINTER. The corresponding row value is stored in:

   \[ X = \text{LED-POINTER}(Y) \]

4. Since an LED is lit in the row pointed to by X, the value counter for that row is incremented by one.

Assuming the LED situation of Figure 7.7, the second value counter has been set to the value 1.

5. The next column is examined: Y = Y - 1.

   If Y is not 0, we go back to (3); otherwise the evaluation process may proceed to its next phase.

Exercise 7-1: Using the flowchart of Figure 7.5, and using the example of Figure 7.7, show the resulting values contained in the value counters when we finally exit from the test at (6) in the flowchart of Figure 7.5.

The actual number of LEDs lit in each row must now be transformed into a score. The SCORETABL is used for that purpose. If the scoring rules contained in this table are changed, they will completely modify the way the game is played.

The score table contains four byte-long numbers per row. Each number corresponds to the score to be earned by the player when 0, 1, 2, or 3 LEDs are lit in that row. The logical organization of the score table is shown in Figure 7.8. The entries in the table correspond to the score values which have been selected for the program presented at the beginning of this chapter. Any combination of LEDs in rows 1 or 3 scores 0. Any combination of 2 LEDs in row 2 scores 1, but, three LEDs score 3. Practically, this means that the score value of row 1 is obtained by merely using an indexed access technique with the number of LEDs lit as the index. For row 2, a displacement of four must be added for table access. In row 3, an additional displacement of four must be added. Mathematically, this translates to:

   \[ \text{SCORE} = \text{SCORETABL}[(X - 1) \times 4 + 1 + Y] \]
Fig. 7.5: EVAL Flowchart

1. Y = 3
2. VALUE COUNTERS, SCORE TEMP = 0
3. X = LED POINTER (Y)
4. INCREMENT VALUE COUNTER (X)
5. Y = Y - 1
6. Y = 0?
   NO
   7. X = 3
   YES
   8. TEMP = (X - 1) * 4 + 1
9. Q = SCORTABL (VALUE COUNTER (X), TEMP)
10. SCORE TEMP = SCORE TEMP + Q
11. X = X - 1
12. X = 0?
   NO
   13. RETURN: NEXT SPIN
   YES
14. INCREMENT SCORE
15. SCORE = 167
   NO
16. SIGNAL GAME WON WITH RISING WAVE, LT LEDs
17. PLAY HIGH TONE, SHOW NEW SCORE
18. SCORTEMP = SCORTEMP - 1
19. SCORTEMP = 0?
   NO
   20. DECREMENT SCORE
   YES
21. SHOW NEW SCORE, SOUND LOW TONE
22. SCORE = 0?
   NO
   23. SOUND FALLING TONE
   YES
   RETURN: NEXT SPIN
   NO
   RETURN: NEW GAME

Fig. 7.5: EVAL Flowchart (Continued)
where X is the row number and Y is the number of LEDs lit for that row. Since this technique allows each of the three rows to generate a score, the program must test the value counter in each row to obtain the total score.

This is accomplished by steps 7 and 8: the row pointer is initialized to 3, and a score table displacement pointer is set up:

\[ \text{TEMP} = (X - 1) \times 4 + 1 \]

9. Next, the value of the score is obtained from the table:

\[ Q = \text{SCORTABL}(\text{value counter (X)}, \text{TEMP}) \]

The value of that row's score is obtained by accessing the score table indexed by the number of LEDs lit, contained in the value counter for that row, plus a displacement equal to TEMP. The intermediate score is obtained by adding this partial score to any previous value:

10. \[ \text{SCORTMP} = \text{SCORTMP} + Q \]
11. Finally, the row number is decremented, and the process is repeated until X reaches the value 0.
12. Whenever X reaches the value 0, the score for this spin has been computed and stored in location SCORTMP.
13. At this point, the score computed above (SCORTMP) is examined by the program, and two possibilities exist: if the SCORTMP is 0, a branch occurs to 20, where the game score is decremented. If SCORTMP is not 0, the game score will be increased by the score for this spin — SCORTMP. Let us follow this path first.
14. The total game score is incremented by one.
15. It is then tested for the maximum value of 16.
16. If the maximum score of 16 is reached in step 15, a special audible and visual signal is generated to reward the player. A new game may be started.
17. If 16 is not reached in step 15, the updated game score is shown to the player, accompanied by a high-pitched tone.
18. The amount by which the game score must be increased, SCORTMP, is decremented.
19. If SCORTMP is not zero, more points must be added to the game score, and a branch occurs to 14. Otherwise, the player may enter the next spin.

Let us now follow the other path from position thirteen on the flowchart, where the total score had been tested:
20. The score for this spin is 0, so the game score is decremented.
21. It is displayed to the player along with a low tone.
22. The new score is tested for the minimum value 0. If this minimum value has been reached, the player has lost. Otherwise, the player may keep playing.
23. A descending siren-type tone is generated to indicate the loss, and the game ends.

THE PROGRAM

Data Structures

Two tables are used by this program: 1) the score table is used to compute a score from the number of LEDs lit in each row — this has already been described; 2) the LTABLE is used to generate the appropriate code on the I/O port to light the specified LED. Each entry within this table contains a pattern to be OR’d into the I/O register to light the specified LED.

Vertically, in the memory, the table entries correspond to the first column, the second column, and then the third column of LEDs. Looking at the program on lines 39, 40, and 41, the rows of digits correspond respectively to the columns of LEDs. For example, the third entry in the table, i.e., 64 decimal, or 40 hexadecimal (at address 001C) corresponds to the third LED in the first column on the Games Board, or LED 7.

Page Zero Variables

The following variables are stored in memory:
— TEMP is a scratch location

---

```
LINE # LOC  CODE   LINE
0002  0000  .BLOT MACHINE SIMULATOR PROGRAM.
0003  0000  !PRESS ANY KEY TO START 'SPIN'.
0004  0000  ■SCORE DETERMINED BY ARRAY 'SCORPB'.
0005  0000  •8 POINTS INITIAL SCORE. ONE POINT PENALTY
0006  0000  •FOR EACH BAD SPIN.
0007  0000  •
0008  0000  # = 01
0009  0000  TEMP = 01; TEMPORARY STORAGE.
0100  0001  SCORPB = 01; TEMPORARY SCORE STORAGE.
0101  0002  SCORPB = 01; SCORE.
0102  0003  DUR = 01; DURATION OF TONES.
0103  0004  FRE = 01; FREQUENCY OF TONES.
0104  0005  SPEED = 01; SPEEDS OF REVOLUTION FOR LEDs
0105  0006  INDEX = 01; I N COLUMNS.
0106  0007  INDEX = 01; INDEX OF LED 1.
0107  0008  INDEX = 01; INDEX OF LED 2.
0108  0009  INDEX = 01; INDEX OF LED 3.
0109  000A  PORT = 01; PORT 1/0.
0110  000B  PORT = 01; PORT 1/0.
0111  000C  PORT = 01; PORT 1/0.
0112  000D  PORT = 01; PORT 1/0.
0113  000E  PORT = 01; PORT 1/0.
0114  000F  PORT = 01; PORT 1/0.
0115  0010  PORT = 01; PORT 1/0.
0116  0011  PORT = 01; PORT 1/0.
0117  0012  PORT = 01; PORT 1/0.
0118  0013  PORT = 01; PORT 1/0.
0119  0014  PORT = 01; PORT 1/0.
011A  0015  PORT = 01; PORT 1/0.
011B  0016  PORT = 01; PORT 1/0.
011C  0017  PORT = 01; PORT 1/0.
011D  0018  PORT = 01; PORT 1/0.
011E  0019  PORT = 01; PORT 1/0.
011F  001A  PORT = 01; PORT 1/0.
```

---

Fig. 7.9: Slot Machine Program
Fig. 7.9: Slot Machine Program (Continued)
--- Fig. 7.9: Slot Machine Program (Continued) ---

- **SCORTP** is used as a temporary storage for the score gained or lost on each spin.
- **SCORE** is the game score.
- **DUR** and **FREQ** specify the usual constants for tone generation.
- **SPEEDS** (3 locations) specify the revolution speeds for the three columns.
- **INDX** (3 locations): delay counters for LED revolutions.
- **INCR** (3 locations): pointers to the LED positions in each column used to fetch patterns out of tables.
- **LTMSK** (3 locations): patterns indicating lit LEDs.
- **VALUES** (3 locations): number of LEDs lit in each column.
- **RND** (6 locations): scratch-pad for random number generator.
Program Implementation

The program consists of a main program and two main subroutines: DISPLY and EVAL. It also contains some utility subroutines: DELAY for a variable length delay, LIGHT to light the appropriate LED, TONE to generate a tone, and RANDOM to generate a random number.

The main program is stored at memory locations 200 and up. As usual, the three data-direction registers for Ports A and B of VIA#1 and for Port B of VIA#3 must be conditioned as outputs:

```
LDA #$FF
STA DDR1A
STA DDR1B
STA DDR3B
```

As in previous chapters, the counter register of timer 1 is used to provide an initial random number (a seed for the random number generator). This seed is stored at memory location RND + 1, where it will be used later by the random number generation subroutine:

```
LDA TICL
STA RND + 1
```

On starting a new game, the initial score is set to 8. It is established:

```
START LDA #8
STA SCORE
```

and displayed:

```
TAY Y must contain it
JSR LIGHT
```

The LIGHT subroutine is used to display the score by lighting up the LED corresponding to the contents of register Y. It will be described later.

The slot machine program is now ready to respond to the player. Any key may be pressed:

```
KEY JSR GETKEY
```

As soon as a key has been pressed, the wheels must be spun:

```
JSR DISPLY
```

Once the wheels have stopped, the score must be evaluated and displayed with the accompanying sound:

```
JSR EVAL
```

If the final score is not "0," the process is restarted:

```
LDA SCORE
BNE KEY
```

and the user may spin the wheels again. Otherwise, if the score was "0," a new game is started:

```
BEQ START
```

This completes the body of the main program. It is quite simple because it has been structured with subroutines.

The Subroutines

The algorithms corresponding to the two main subroutines DISPLY and EVAL have been described in the previous section. Let us now consider their program implementation.

DISPLY Subroutine

Three essential subroutine parameters are LOLIM, HILIM, and SPDPRM. For example, lowering LOLIM will result in a longer spinning time for the LEDs. Various other effects can be obtained by varying these three parameters. One might be to include a win almost every time! Here LOLIM = 90, HILIM = 134, SPDPRM = 80.

Memory location INCR is used as a pointer to the current LED position. It will be used later to fetch the appropriate bit pattern from the table, and may have the value 0, 1, or 2 (pointing to LED positions 1, 2, or 3). The three pointers for the LEDs in each column are stored respectively at memory locations INCR, INCR + 1, and INCR + 2. They are initialized to 0:
Note that in the previous examples (such as Figure 7.7), in order to simplify the explanations, we have used pointers X and Y to represent the values between 1 and 3. Here, X and Y will have values ranging between 0 and 2 to facilitate indexing. The wheel pointer is set to the right-most wheel:

```
LDRND    LDY #2
```

An initial random number is obtained with the RANDOM subroutine:

```
GETRND    JSR RANDOM
```

The number returned by the subroutine is compared with the acceptable low limit and the acceptable high limit. If it does not fit within the specified interval, it is rejected, and a new number is obtained until one is found which fits the required interval.

```
CMP #HILIM    Too large?
BCS GETRND    If so, get another
CMP #LOLIM    Too small?
BCC GETRND    If so, get another
```

The valid random number is then stored in the index location INDX and in the SPEEDS location for the current column. (See Figure 7.10.)

```
STA INDX,Y
STA SPEEDS,Y
```

The same process is carried out for column 1 and column 0:

```
DEY
BPL GETRND    Get next random #
```

Once all three columns have obtained their index and speed, a new iteration loop is started, using register X as a wheel counter:

```
UPDATE    LDX #2
```

```
UPDTLP    LDY SPEEDS,X    Is speed (X) = 0?
BEQ NXTUPD    If so, update next column
```

As long as the speed is not 0, the next LED in that column will have to be lit. The delay count is decremented:

```
DEC INDX,X    Decrement loop, index (X)
```
If the delay has not decremented to 0, a branch occurs to NXTUPD which will be described below. Otherwise, if the delay counter INDX is decremented to 0, the next LED should be lit. The LED pointer is incremented with a possible wrap-around if it reaches the value 3:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE NXTUPD</td>
<td>If loop index(X) $\not\equiv 0$, do next update</td>
</tr>
<tr>
<td>LDY INCR,X</td>
<td>Inc pointer</td>
</tr>
<tr>
<td>INY</td>
<td></td>
</tr>
<tr>
<td>CPY #3</td>
<td>Pointer = 3?</td>
</tr>
<tr>
<td>BNE NORST</td>
<td>If not, skip</td>
</tr>
<tr>
<td>LDY #0</td>
<td>Reset to 0</td>
</tr>
<tr>
<td>NORST</td>
<td>Restore pointer(INDX)</td>
</tr>
</tbody>
</table>

The new value of the LED pointer is stored back into INCR for the appropriate column. (Remember that within the UPDATE routine, X points at the column.) In order to light the appropriate LED, a bit pattern must be obtained from LTABLE. Note that LTABLE (and also SCORTB) is treated conceptually, as if it was a two-dimensional array, i.e., having rows and columns. However, both LTABLE and SCORTB appear in memory as a contiguous series of numbers. Thus, in order to obtain the address of a particular element, the row number must be multiplied by the number of columns and then added to the column number.

The table will be accessed using the indexed addressing mode, with register Y used as the index register. In order to access the table, X must first be multiplied by 3, then the value of INCR (i.e., the LED pointer) must be added to it.

Multiplication by 3 is accomplished through a left shift followed by an addition, since a left shift is equivalent to multiplication by 2:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STX TEMP</td>
<td>Multiply X by 3</td>
</tr>
<tr>
<td>TXA</td>
<td>Left shift</td>
</tr>
<tr>
<td>ASL A</td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td>ADC TEMP</td>
<td>Plus one</td>
</tr>
</tbody>
</table>

The value of INCR is added, and the total is transferred into register Y so that indexed addressing may be used. Finally, the entry may be retrieved from LTABLE:

ADC INCR,X
TAY
LDA LTABLE,Y   Get pattern for LED

Once the pattern has been obtained, it is stored in one of three memory locations at address LTMSK and following. The pattern is stored at the memory location corresponding to the column currently being updated, where the LED has "moved." The lights will be turned on only after the complete pattern for all three columns has been implemented. As a result of the LED having moved one position within that column, the speed constant must be incremented:

STA LTMSK,X
LDA LTABLE,Y Get pattern for LED

The index is set so that it is equal to the new speed:

STY INDX,X

Note that special handling will now be necessary for LED #9. The pattern to be displayed on the first eight LEDs was stored in the LTABLE. The fact that LED #9 must be lit is easily recognized by the fact that the pattern for column #3 shows all zeroes; since one LED must be lit at all times within that column, it implies that LED #9 will be lit:

LEDUPD   LDA #0
         STA PORT1B   Reset LED 9

Next, the pattern for the third column is obtained from the location where it had been saved at LTMSK + 2. It is tested for the value of 0:

LDA LTMSK + 2
BNE OFFLD9

If this pattern is 0, then LED #9 must be turned on:

LDA #01
STA PORTIB

Otherwise, a branch occurs to location OFFLD9, and the remaining LEDs will be turned on. The pattern contained in the accumulator which was obtained from LTMSK + 2, is successively OR’ed with the patterns for the second and first columns:

```
OFFLD9
  LDA #0
  ORA LTMSK
  ORA LTMSK + 1
```

At this point, A contains the final pattern which must be sent out in the output port to turn on the required LED pattern. This is exactly what happens:

STA PORTIA

At the same time, the speaker is toggled:

```
LDA PORT3B
  EOR #$FF
  STA PORT3B
```

It is important to understand that even though only the LED for one of the three columns has been moved, it is necessary to simultaneously turn on LEDs in all of the columns or the first and second columns would go blank!

Once the third column has been taken care of, the next one must be examined. The column pointer X is therefore decremented, and the process is continued:

```
NDTUPD     DEX      BPL UPDTLP   If X >= 0 do next update
```

Once the second and the first columns have been handled, a delay is implemented to avoid flashing the LEDs too fast. This delay is controlled by the speed parameter SPDPRM:

```
    LDY #SPDPRM
    WAIT
    DEY
    BNE WAIT
```

Once this complete cycle has been executed, the speed location for each column is checked for the value 0. If all columns are 0, the spin is finished:

```
LDA SPEEDS
  ORA SPEEDS + 1
  ORA SPEEDS + 2
  BNE UPDATE
```

Otherwise, a branch occurs at the location UPDATE. If all LEDs have stopped, a pause must be generated so that the user may see the pattern:

```
LDA #$FF
STA DUR
JSR DELAY
```

and exit occurs:

```
RTS
```
Exercise 7-2: Note that the contents of the three SPEEDS locations have been OR'ed to test for three zeroes. Would it have been equivalent to add them together?

**EVAL Subroutine**

This subroutine is the user output interface. It computes the score achieved by the player and generates the visual and audio effects. The constants for frequencies for the high tone generated by a win situation and the low tone generated by a lose situation are specified at the beginning of this subroutine:

```
HITONE = $20
LOTONE = $F0
```

The method used to compute the number of LEDs lit per row has been discussed and shown in Figure 7.7. The number of LEDs lit for each row is initially reset to 0:

```
EVAL LDA #0 STA VALUES
       STA VALUES + 1
       STA VALUES + 2
```

The temporary score is also set to 0:

```
STA SCORTP
```

Index register Y will be used as a column pointer, and the number of LEDs lit per row will be computed. The number of the LED lit for the current column is obtained by reading the appropriate INCR entry. See the example in Figure 7.11. The value contained in each of the three locations reserved for INCR is a row number. This row number is stored in register X, and is used as an index to increment the appropriate value in the VALUES table. Notice how this is accomplished in just two instructions, by cleverly using the indexed addressing feature of the 6502 twice:

```
CNTLP LDY #2 3 iterations
       LDX INCR,Y
       INC VALUES,X
```

Once this is done for column 2, the process is repeated for columns 1 and 0:

```
DEY
BPL CNTLP
```

Now, another iteration will be performed to convert the final numbers entered in the VALUES table into the actual scores as per the specifications of the score table, SCORTB. Index register X is used as a row-pointer for VALUES and SCORTB.

```
LDX #2
```

Since the SCORTB table has four one-byte entries per row level, in order to access the correct byte within the table the row number must first be multiplied by 4, then the corresponding “value” (number of LEDs lit) for that row must be added to it. This provides the correct displacement. The multiplication by 4 is implemented by two successive left shifts:

```
SCORLP TXA
       ASLA
       ASLA
```

The number presently contained in the accumulator is equal to 4 times the value contained in X, i.e., 4 times the value of the row-pointer. To obtain the final offset within the SCORTB table, we must add to that the number of LEDs lit for that row, i.e., the number contained in the VALUES tables. This number is retrieved, as usual, by performing an indexed addressing operation:

```
CLC
       ADC VALUES,X Column address in array
```

This results in the correct final offset for accessing SCORTB.

The indexed access of the SCORTB table can now be performed. Index register Y is used for that purpose, and the contents of the accumulator are transferred to it:

```
TAY
```
The access is performed:

```
LDA SCORTB,Y Get score for this spin
```

The correct score for the number of LEDs lit within the row pointed to by index register X is now contained in the accumulator. The partial score obtained for the current row is added to the running total for all rows:

```
CLC
ADC SCORTP Total the scores
STA SCORTP Save
```

The row number is then decremented so that the next row can be examined. If X decrements from the value 0, i.e., becomes negative, we are done; otherwise, we loop:

```
DEX
BPL SCORLP
```

At this point, a total score has been obtained for the current spin. Either a win or a lose must be signaled to the player, both visually and audibly. In anticipation of activating the speaker, the memory location DUR is set to the correct tone duration:

```
LDA #$60
STA DUR
```

The score is then examined: if 0, a branch occurs to the LOSE routine:

```
LDA SCORTP
BEQ LOSE
```

Otherwise, it is a win. Let us examine these two routines.

**WIN Routine**

The final score for the user (for all spins so far) is contained in memory location SCORE. This memory location will be incremented one point at a time and checked every time against the maximum value 16. Let us do it:

```
WIN
INC SCORE
LDY SCORE
CPY #16
```

If the maximum value of 16 has been reached, it is the end of the game and a branch occurs to location WINEND:

```
BEQ WINEND
```

Otherwise, the score display must be updated and a beep must be sounded:

```
JSR LIGHT
```

The LIGHT routine will be described below. It displays the score to the player. Next, a beep must be sounded.

```
LDA #HITONE
JSR TONE
```

The TONE routine will be described later. A delay is then implemented:

```
JSR DELAY
```

then the score for that spin is decremented:

```
DEC SCORTP
```

and checked against the value 0. If it is 0, the scoring operation is complete; otherwise, the loop is reentered:

```
BNE WIN
RTS
```

**WINEND Routine**

This routine is entered whenever a total score of 16 has been reached. It is the end of the game. All LEDs are turned on simultaneously, and a siren sound with rising frequencies is activated. Finally, a restart of the game occurs.
All LEDs are turned on by loading the appropriate pattern into Port 1A and Port 1B:

```
LDA #$FF
STA PORT1A
STA PORT1B
```

Turn on all LEDs

Variables are reinitialized: the total score becomes 0, which signals to the main program that a new game must be started, the DUR memory location is set to 4 to control the duration of time for which the beeps will be sounded, and the frequency parameter is set to "FF" at location TEMP:

```
STA TEMP
LDA #0
STA SCORE
LDA #4
STA DUR
```

Freq. parameter
Clear for restart
Beep duration

The TONE subroutine is used to generate a beep:

```
RISE LDA TEMP
JSR TONE
```

Get frequency
Generate beep

The beep frequency constant is then decremented, and the next beep is sounded at a slightly higher pitch:

```
DEC TEMP
BNE RISE
```

Whenever the frequency constant has been decremented to 0, the siren is complete and the routine exits:

```
RTS
```

**LOSE Routine**

Now let us examine what happens in the case of a lose situation. The events are essentially symmetrical to those that have been described for the win.

In the case of a loss, the score needs to be updated only once. It is decremented by 1:

```
130 DEC SCORE
```

The lowered score is displayed to the user:

```
LDY SCORE
JSR LIGHT
```

An audible tone is generated:

```
LDA #LOTONE
JSR TONE
```

The final value of the score is checked to see whether a "0" score has been reached. If so, the game is over; otherwise, the next spin is started:

```
LDY SCORE
BEQ LOSEND
RTS
```

Let us look at what happens when a "0" score is reached (LOSEND). A siren of decreasing frequencies will be generated. All LEDs will go blank on the board:

```
LOSEND LDA #0
STA TEMP
STA PORT1A
```

Clear LED #1

The beep duration for each frequency is set to a value of 4, stored at memory location DUR:

```
LDA #4
STA DUR
```

The beep for the correct frequency is then generated:

```
FALL LDA TEMP
JSR TONE
```

Play beep

A siren of decreasing frequencies will be generated. All LEDs will go blank on the board:

```
LOSEND LDA #0
STA TEMP
STA PORT1A
```

Clear LED #1

The beep for the correct frequency is then generated:

```
FALL LDA TEMP
JSR TONE
```

Play beep

Next, the frequency constant is increased by 1, and the process is restarted until the TMP register overflows.
This completes our description of the main program. Let us now examine the four subroutines that are used. They are: DELAY, LIGHT, TONE, and RANDOM.

**DELAY Subroutine**

This subroutine implements a delay; the duration of the delay is set by the contents of memory location DUR. The resulting delay length will be equal to \((2046 \times DUR + 10)\) microseconds. The delay is implemented using a traditional two-level, nested loop structure. The inner-loop delay is controlled by index register X, while the outer-loop delay is controlled by index register Y, which is initialized from the contents of memory location DUR. Y is therefore initialized:

```
DELAY LDY DUR
```

The inner loop delay is then implemented:

```
DL1 LDX #$FF
DL2 BNE * + 2
    DEX
    BNE DL2
```

And, finally, the outer loop is implemented:

```
DEY
BNE DL1
RTS
```

**Exercise 7-3:** Verify the exact duration of the delay implemented by the DELAY subroutine.

**LIGHT Subroutine**

This subroutine lights the LED corresponding to the number contained in register Y. Remember that the fifteen LEDs on the Games Board are numbered externally from 1 to 15 but are connected to bits 0 to 7 of Port 1A and 0 to 7 of Port 1B. Thus, if a score of 1 must be displayed, bit 0 of Port 1A must be turned on. Generally, bit N of Port 1A must be turned on when N is equal to the score minus one. However, there is one exception. To see this, refer to Figure 1.4 showing the LED connections. Notice that bit 6 of Port 1B is not connected to any LEDs. Whenever a score of fifteen must be displayed, bit 7 of Port 1B must be turned on. This exception will be handled in the routine by simply not decrementing the score when it adds up to fifteen.

The correct pattern for lighting the appropriate LED will be created by shifting a ‘1’ into the accumulator at the correct position. Other methods will be suggested in the exercise below. Let us first initialize:

```
LIGHT LDA #0
    STA TEMP
    STA PORT1A
    STA PORT1B
```

We must first look at the situation where the score contained in Y is 15 and where we do nothing (no shift):

```
CPY #15
    BEQ *
```

Code for uncorrected bit?

If so, no change

For any other score, it is first decremented, then the shift is performed:

```
DEY
SEC
    ROL A
```

Decrement to internal code

Set bit to be shifted

The contents of the accumulator were zeroed in the first instruction of this subroutine. The carry is set to the value 1, then shifted into the right-most position of A. (See Figure 7.12.) This process will be repeated as many times as necessary. Since we must count from 1 to 14, or 0 to 13, an overflow will occur whenever the ‘1’ that is rotated in the accumulator “falls off” the left end. As long as this does not happen, the shifting process continues, and a branch to location LTCC is implemented:

```
BCC LTCC
```
The shifting process continues. The counter is decremented, and, if it reaches the value “0,” we are done; otherwise, the process is repeated:

```
ROL A
LTCC
DEY
BPL LTSHFT
```

Once the process is completed, the value of memory location TEMP is examined. If this value is “0,” it indicates that no overflow has occurred and Port 1A must be used. If this value is not “0,” i.e., it is “FF,” then Port 1B must be used:

```
LDX TEMP
BNE HIBYTE
GET OVERFLOW FLAG
```

```
LOBYTE STA PORT1A
RTS
A SENT TO LOW LEDS
```

```
HIBYTE STA PORT1B
RTS
A SENT TO HIGH LEDS
```

**TONE Subroutine**

This subroutine generates a beep. The frequency of the beep is determined by the contents of the accumulator on entry; the duration of the beep is set by the contents of the memory location DUR. This has already been described in Chapter 2.

**RANDOM Subroutine**

This is a simple random number generator. The subroutine has already been described in Chapter 3.

**Exercise 7-4:** Suggest another way to generate the correct LED pattern in the accumulator, without using a sequence of rotations.

**Game Variations**

The three rows of LEDs supplied on the Games Board may be interpreted in a way that is different from the one used at the beginning of this chapter. Row 1 could be interpreted as, say, cherries. Row 2 could be interpreted as stars, and row 3 could be interpreted as oranges. Thus, an LED lit in row 1 at the end of a spin shows a cherry, while...
two LEDs in row 3 show two oranges. The resulting combination is one cherry and two oranges. The scoring table used in this program can be altered to score a different number of points for each combination, depending upon the number of cherries, oranges, or stars present at the end of the spin. It becomes simply a matter of modifying the values entered into the scoring table. When new values are entered into the scoring table a completely different scoring result will be implemented. No other alterations to the program will be needed.

SUMMARY

This program, although simple in appearance, is relatively complex and can lead to many different games, depending upon the evaluation formula used once the lights stop. For clarity, it has been organized into separate routines that can be studied individually.

8

ECHO

THE RULES

The object of this game is to recognize and duplicate a sequence of lights and sounds which are generated by the computer. Several variations of this game, such as “Simon” and “Follow Me” (manufacturer trademarks*), are sold by toy manufacturers. In this version, the player must specify, before starting the game, the length of the sequence to be recognized. The player indicates his or her length preference by pressing the appropriate key between 1 and 9. At this point the computer generates a random sequence of the desired length. It may then be heard and seen by pressing any of the alphabetic keys (A through F).

When one of the alphabetic keys is pressed, the sequence generated by the program is displayed on the corresponding LEDs (labeled 1 through 9) on the Games Board, while it is simultaneously played through the loudspeaker as a sequence of notes. While this is happening, the player should pay close attention to the sounds and/or lights, and then enter the sequence of numbers corresponding to the sequence he or she has identified. Every time that the player presses a correct key, the corresponding LED on the Games Board lights up, indicating a success. Every time a mistake is made, a low-pitched tone is heard.

At the end of the game, if the player has guessed successfully, all LEDs on the board will light up and a rising scale (succession of notes) is played. If the player has failed to guess correctly, a single LED will light up on the Games Board indicating the number of errors made, and a descending scale will be played.

If the player guessed the series correctly, the game will be restarted. Otherwise, the number of errors will be cleared and the player will be given another chance to guess the series.

*“Follow Me” is a trademark of Atari, Inc., “Simon” is a trademark of Milton Bradley Co.
At any time during a game, the player may press one of the alphabetic keys that will allow him or her to hear the sequence again. All previous guesses are then erased, and the player starts guessing again from the beginning.

Two LEDs on the bottom row of the LED matrix are used to communicate with the player:

LED 10 (the left-most LED) indicates “computer ready — enter the length of the sequence desired.”

LED 11 lights up immediately after the player has specified the length of the sequence. It will remain lit throughout the game and it means that you should “enter your guess.”

At this point, the player has three options:
1. To press a key corresponding to the number in the sequence that he or she is attempting to recognize.
2. To press key 0. This will result in restarting the game.
3. To press keys A through F. This will cause the computer to play the sequence again, and will restart the guessing sequence.

Variations

The program provides a good test for your musical abilities. It is suggested that you start each new game by just listening to the sequence as it is played on the loudspeaker, without looking at the LEDs. This is because the LEDs on the Games Board are numbered, and it is fairly easy to remember the light sequence simply by memorizing the numbers. This would be too simple. The way you should play it is to start with a one-note sequence. If you are successful, continue with a two-note sequence, and then with a three-note sequence. Match your skills with other players. The player able to recognize the longest sequence is the winner. Note that some players are capable of recognizing a nine-note sequence fairly easily.

After a certain number of notes are played (e.g., when more than five notes are played), in order to facilitate the guessing you may allow the player to look at the LEDs on the Games Board. Another approach might be to allow the player to press one of the alphabetic keys at any time in order to listen to the sequence again. However, you may want to require that the player pay a penalty for doing this. This could be achieved by requiring that the player recognize a second sequence of the same length before trying a longer one. This means that if, for example, a player attempts to recognize a five-note sequence but becomes nervous after making a mistake and forgets the sequence, that player will be allowed to press one of the alphabetic keys and hear the sequence again. However, if the player is successful on the second attempt, he or she must then recognize another five-note sequence before proceeding to a six-note one.

You can be even tougher and specify that any player is allowed a replay of the stored pattern a maximum of two, three, or five times per game. In other words, throughout the games a player may replay the sequence he or she is attempting to guess by pressing one of the alphabetic keys, but this resource may be used no more than n times.

An ESP Tester

Another variation of this game is to attempt to recognize the sequence without listening to it or seeing it! Clearly, in such a case you can rely only on your ESP (Extra Sensory Perception) powers to facilitate guessing. In order to determine whether you have ESP or not, set the length of the initial sequence to “1.” Then, hit the key in an attempt to guess the note selected by the program. Try this a number of times. If you do not have ESP your results should be random. Statistically, you should win one out of nine times which is only one-ninth of the time, or 11.11% of the time. Note that this percentage is valid only for a large number of guesses.

If you win more than 11% of the time, you may have ESP! If your score is higher than 50%, you should definitely run for political office or immediately apply for a top management position in business. If your score is less than 11%, you have “negative ESP” and you should consider looking both ways before crossing the street.

The following is an exercise for readers who have a background in statistics.

Exercise 8-1: Compute the statistical probability of guessing a correct two-number sequence, and a correct four-number sequence.

A TYPICAL GAME

The program starts at location 200. As usual, LED 10 lights up as shown in Figure 8.1. We specify a series of length two by pushing key “2” on the keyboard. The LED display as it appears in Figure 8.2, means “enter your guess.”

We want to hear the tunes so we push key “F.” In response, LEDs 5 and 2 light up briefly on the Games Board and corresponding tones
are heard through the speaker. This is illustrated in Figure 8.3. We must now enter the sequence we have recognized. We push "5" on the keyboard. In response, LED 11 goes blank and LED 5 lights up briefly. Simultaneously, the corresponding note is played through the speaker. It is a successful guess!

Next, we press key "2." LED 2 lights up, and the speaker produces the matching tone indicating that our second guess has also been successful. A moment later, all LEDs on the board light up to congratulate us and the rising scale is sounded. It is a sequence of notes of increasing frequencies meant to confirm that we have guessed successfully. The game is then restarted, and LED 10 lights up, as shown in Figure 8.1.

Let us now follow a losing sequence: LED 10 is lit at the beginning of the game, as in Figure 8.1. This time we press key "1" in order to specify a one-note sequence. Led 11 lights up, as shown in Figure 8.2. We press key "F," and the note is played on the speaker. (We do not look at the Games Board to see which LED lights up, as that would be too easy.) We press key "3." A "lose" sound is heard, and LED 1 lights up indicating that one mistake has been made. A decreasing scale is then played (notes of decreasing frequencies) to confirm to the unfortunate player that he or she has guessed the sequence incorrectly. The game is then continued with the same sequence and length, i.e., the situation is once again the one indicated in Figure 8.2.

If at this point the player wants to change the length of the sequence, or enter a new sequence, he or she must explicitly restart the game by pressing key 0. After pressing key 0, the situation will be the one indicated in Figure 8.2, where the length of the sequence can be specified again.

THE ALGORITHM

The flowchart for this program is shown in Figure 8.4. Let us examine it, step-by-step:

1. The program tells the player to select a sequence length by lighting LED 10 on the Games Board.
2. The sequence length is read from the keyboard. (Keys 0 and A-F are ignored at this point.)
3. The two main variables are initialized to "0," i.e., the number of guesses and the number of errors are cleared.
4. A sequence table of the appropriate length must then be generated using random numbers whose values are between 1 and 9.
5. Next, LED 11 is lit, and the player's keystroke is read.
6. If it is "0," the game is restarted. Otherwise, we proceed.
7. If the keystroke value is greater than or equal to 10, it is an alphabetic character and we branch off to the right part of the flowchart into steps 8 and 9. The recorded sequence is displayed to the player, all variables are reinitialized to 0, and the guessing process is restarted. If the keystroke was a number between 1 and 9, it must be matched against the stored value. We go to 10 on the flowchart.
Fig. 8.4: Echo Flowchart

1. Signal player to enter sequence length.
2. Read sequence length from keyboard.
3. Guess number = 0, errors = 0.
4. Fill sequence table with random numbers between 1 and 9.
5. Read player's keystroke.
6. If keystroke = 01, yes; if not, no.
7. Start

8. Play sequence.
9. If guess number = 0, errors = 0, yes.
10. Key number = stored value.
11. Light LED (key number) and play tone (key number).
12. Increment guess number.
13. Key number = sequence length.
14. If errors = 01, yes; if not, no.
15. Light all LEDs and play ascending tones.
16. Display number of errors, play descending tones.
17. If guess number = 0, errors = 0, yes.
18. Increment errors.
19. Play low tone.

Fig. 8.4: Echo Flowchart (Continued)
10. If the guess was correct, we branch right on the flowchart to step 11.

11. Since the key pressed matches the value stored in memory, the corresponding LED on the Games Board is lit, and the tone corresponding to the key that has been pressed is played.

12. The guessed number is incremented, and then it is compared to the maximum length of the sequence to be guessed.

13. A check is made to see if the maximum length of the sequence has been reached. If it has not, a branch occurs back to step 5 on the flowchart, and the next keystroke is obtained. If the maximum length of the sequence has been reached, we proceed down the flowchart to the box labeled 14.

14. The total number of errors made by the player is checked. The variable ERRORS is tested against the value "0." If it is "0" it is a winning situation and a branch occurs to the box labeled 15.

15. All LEDs on the board are lit, a sequence of ascending tones is played, and a branch occurs back to the beginning of the game.

Let us now go back to box 14. If the number of errors was greater than zero, this is a "lose" situation and a branch occurs to box 16.

16. The number of errors is displayed, and a sequence of descending tones is played.

17. All variables are reset to 0, and a branch occurs to box 5, giving the player another chance to guess the series.

Now we shall turn our attention back to box 10 on the flowchart, where the value of the key was being tested against the stored value. We will assume this time that the guess was wrong, and branch to the left of box 10.

18. The number of errors made by the player is incremented by one.

19. A low tone is played to indicate the losing situation. The program then branches back to box 12 and proceeds as before.

THE PROGRAM

The complete program appears in Figure 5.1. The program uses two tables, and several variables. The two tables are NOTAB used to specify the note frequencies, and DURTAB used to specify the note durations. Both of these tables were introduced in Chapter 2, and will not be described here. Essentially, they provide the delay constants required to implement a note of the appropriate frequency and to play it for the appropriate length of time. Note that it is possible to modify

---

**Fig. 8.5: Echo Program**
ECHO

0142 02AC 3E
0143 02AD E9 01
0144 02AF D0 F0
0145 02B1 85 01
0146 02B3 BD 02
0147 02B5 F0 BD
0148 02B7 AF FF
0149 02B9 BD 00 A0
014A 02BC BD 00 A0
014B 02DF AF 01
0152 02C1 48
0153 02C2 20 FA 02
0154 02C5 48
0155 02C6 1B
0156 02C7 49 01
0157 02CA CF 0A
0158 02CB D0 F4
0159 02C9 F0 B4
0160 02CF
0161 02CF
0162 02CF
0163 02CF
0164 02CF
0165 024F 4B
0166 024E 0A 00
0167 024F 0A 00
0168 0250 BD 00 A0
0169 0251 3B
0170 0252 0A
0171 0256 0B
0172 0257 20 22
0173 0259 0A 00
0174 025B 49 04
0175 025D 84 02
0176 025F B5 0A
0177 0261 49 04
0178 0263 20 CF 02
0179 0265 A0 FF
017A 0267 A6 03
017B 0269 BB
017C 026B 00 F9
017D 026D 00 F9
017E 026F 00 F9
017F 0271 A6 01
0180 0273 A6 01
0181 0275 A6 04
0182 0277 F0 00
0183 0279 FF 00
0184 027B 00 F9
0185 027D FF 00
0186 027F 00 FF
0187 0281 E6 02
0188 0283 AV 00
0189 0285 BB 03
018A 0287 00 FF
018B 0289 20 04 03
018C 028B FF 0C
018D 028D 00 CF 02
018E 028F 0B 00
018F 0291 20 FA 02
0190 0293 0A 01
0191 0293 A5 00
0192 0299 CE 05
0193 029A 00 A0
0194 029C A5 02
0195 029E 0B
0196 029F 15
0197 02A0 0B 02
0198 02A3 A9 09
0199 02A4 4B
019A 02A8 20 FA 02
019B 02A9 0A 00
019C 02AB 68

Fig. 8.5: Echo Program (Continued)
the difficulty of the game by increasing or decreasing the duration during which each note is played. Clearly, reducing the duration makes the game more difficult. Increasing the duration will usually make it easier, up to a point. You are encouraged to try variations.

The main variables used by the program are the following:

DIGITS contains the number of digits in the sequence to be recognized.

GESNO indicates the number of the current guess, i.e., which of the notes in the series the user is attempting to recognize.

ERRS indicates the number of errors made by the player so far.

TABLE is the table containing the sequence to be recognized.

A few other memory locations are reserved for passing parameters to subroutines or as scratch-pad storage. They will be described within the context of the associated routines.

As usual, the program starts by setting the data direction registers for Port 1A, Port 1B and Port 3B to an output configuration:

```
START LDA #$FF
STA DDR1A
STA DDR1B
STA DDR3B
```

Next, all LEDs on the board are turned off:

```
LDA #0
STAPORT1A
```

and the two variables, ERRS and GESNO, are set to 0:

```
STA ERRS
STA GESNO
```

The random number generator is primed by obtaining a seed and storing it at locations RND + 1 and RND + 4:

```
LDA T1CL
STA RND + 1
STA RND + 4
```

The game is now ready to start. LED 10 must be turned on to indicate to the player that the game is ready:

```
LDA #0/0010
STAPORT1B
```

The keyboard is scanned for the player input using the usual GETKEY subroutine (described in Chapter 1):

```
DIGKEY JSR GETKEY
```

It is checked for the value "0":

```
LDA #$%010
STA PORT1B
```

and the random number generator is primed by obtaining a seed and storing it at locations RND + 1 and RND + 4:
CMP #0
BEQ DIGKEY  If = 0, get another one

If the entry was "0," the program waits for another keystroke. Otherwise, it is compared to the value 10:

CMP #10
BPL DIGKEY  If sequence longer than 9

If the sequence length is greater than 9, it is also rejected. Accepting only valid inputs, using a bracket is known as "reasonableness testing" or "bracket-filtering."

If all is fine, the length of the sequence to be recognized is stored at memory location DIGITS:

STA DIGITS  Length of sequence

A running pointer is then computed and stored at location TEMP. It is equal to the previous length minus 1:

TAX  Use X for computation
DEX  Decrement
FILL  STX TEMP

The RANDOM subroutine is then called to provide a first random number:

JSR RANDOM

The position pointer in the series of notes now being generated is retrieved from TEMP, and stored in index register X in anticipation of storing the new random number in TABLE:

LDX TEMP

The value of the random number contained in the accumulator is then converted to a decimal value between 0 and 9. This process can be performed in various ways. Here, we take advantage of the special decimal mode available on the 6502. The decimal mode is set by specifying:

SED  Set decimal mode

Note that the carry flag must be cleared, prior to an addition:

CLC  Clear carry

The trick used here is to add "0" to the random number contained in the accumulator. The result in the right part of A is guaranteed to be a digit between 0 and 9, since we are operating in the decimal mode. Naturally, any other number could also be added to A to make its contents "decimal"; however, this would change the distribution of the random numbers, and some numbers in the series such as 0, 1, and 2 might never appear. Once this conversion has been performed, the decimal mode is simply turned off:

ADC #0  Add "0" in decimal mode
CLD  Clear decimal mode

This is a powerful 6502 facility used to a great advantage in this instance. In order to guarantee that the result left in A be a decimal number between 0 and 9, the upper nibble of the byte is removed by masking it off:

AND $#OF

Finally, a value of "0" is not allowed, and a new number must be obtained if this is the current value of the accumulator:

BEQ FILL

Exercise 8-2: Could we avoid this special case for "0" by adding a value other than "0" to A above?

If this is not the current value of the accumulator, we have a decimal number between 1 and 9 that is reasonably random, which can now be stored in the table. Remember that index register X has been preloaded with the current number's position in the sequence (retrieved from memory location TEMP). It can be used, as is, as an index:

STA TABLE,X  Store # in table

The number pointer is then decremented in anticipation of the next iteration:
and the loop is reentered until the table of random numbers becomes full:

**BPL FILL**

We are now ready to play. LED 12 will be turned on, signaling to the player that he or she may enter a guess:

**KEY**

<table>
<thead>
<tr>
<th>LDA #0</th>
<th>Get guess</th>
</tr>
</thead>
</table>

The player's guess is then read from the keyboard:

| JSR GETKEY | Get guess |

It must be tested for "0" or for an alphabetic value. Let us test for "0":

<table>
<thead>
<tr>
<th>CMP #0</th>
<th>Is it 0?</th>
</tr>
</thead>
</table>

If it is "0," the game is restarted, and a branch occurs to location **START**. If it is not "0," we must check for an alphabetic character:

<table>
<thead>
<tr>
<th>CMP #10</th>
<th>Number &lt;10?</th>
</tr>
</thead>
</table>

| BMI EVAL | If yes, evaluate correctness |

If the value of the input keystroke is less than ten, it is a guess and is evaluated with the **EVAL** routine. Otherwise, the program executes the **SHOW** routine to display the series.

**SHOW Routine**

We will assume here that an alphabetic key has been pressed. **BMI** fails, and we enter the **SHOW** routine. This routine plays the computer-generated tune and lights up the corresponding sequence of LEDs. Also, whenever this routine is entered, the guessing sequence is restarted and the temporary variables are reset to 0:

**SHOW**

<table>
<thead>
<tr>
<th>LDX #0</th>
<th>Get Xth entry in table</th>
</tr>
</thead>
<tbody>
<tr>
<td>STX GESNO</td>
<td></td>
</tr>
<tr>
<td>STX ERRS</td>
<td>Reset all variables</td>
</tr>
</tbody>
</table>

The first table entry is obtained, the corresponding LED is lit, and the corresponding tone is played:

**SHOWLP**

<table>
<thead>
<tr>
<th>LDA TABLE,X</th>
<th>Get Xth entry in table</th>
</tr>
</thead>
<tbody>
<tr>
<td>STX TEMP</td>
<td>Save X</td>
</tr>
<tr>
<td>JSR LIGHT</td>
<td>Light LED # TABLE (X)</td>
</tr>
<tr>
<td>JSR PLAY</td>
<td>Play tone # TABLE (X)</td>
</tr>
</tbody>
</table>

An internote delay is then implemented using **Y** as the loop counter and two dummy instructions to extend the delay:

**DELAY**

<table>
<thead>
<tr>
<th>LDY #$FF</th>
<th>Dummy instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR DUR</td>
<td>Dummy</td>
</tr>
<tr>
<td>ROL DUR</td>
<td>Count down</td>
</tr>
<tr>
<td>DEY</td>
<td>End of loop test</td>
</tr>
</tbody>
</table>

We are now ready to perform the same operation for the next note in the current table. The index pointer is restored and incremented:

<table>
<thead>
<tr>
<th>LDX TEMP</th>
<th>Restore X</th>
</tr>
</thead>
<tbody>
<tr>
<td>INX</td>
<td>Increment it</td>
</tr>
</tbody>
</table>

It is then compared to the maximum number of digits stored in the table. If the maximum has been reached, the display operation is complete and we go back to label **KEY**. Otherwise, the next tone is sounded, and we go back to label **SHOWLP**:

<table>
<thead>
<tr>
<th>CPX DIGITS</th>
<th>All digits shown?</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE SHOWLP</td>
<td>Done, get next input</td>
</tr>
</tbody>
</table>

**EVAL Routine**

Let us now examine the routine which evaluates the guess of the
player. It is the EVAL routine. The value of the corresponding entry in TABLE is obtained and compared to the player's input:

```
EVAL  LDX GESNO Load guess number into X
       CMP TABLE,X Compare guess to number
       BEQ CORECT If correct, tell player
```

If there is a match, a branch occurs to location CORECT; otherwise, the program proceeds to label WRONG. Let us examine this case. If the guess is wrong, one more error is recorded:

```
WRONG  INC ERRS
```

A low tone is played:

```
LDA #$80
STA DUR
LDA #$FF
JSR PLYTON
```

A jump then occurs to location ENDCHK:

```
BEQ ENDCHK Check for end of game
```

Exercise 8-3: Examine the BEQ instruction above. Will it always result in a jump to label ENDCHK? (Hint: determine whether or not the Z bit will be set at this point.)

Exercise 8-4: What are the merits of using BEQ (above) versus JMP?

Now we shall consider what happens in the case of a correct guess. If the guess is correct, we light up the corresponding LED and play the corresponding tone. Both subroutines assume that the accumulator contains the specified number:

```
CORECT  JSR LIGHT Turn on LED
       JSR PLAY Play note to confirm
```

We must now determine whether we have reached the end of a sequence or not, and take the appropriate action. The number of guesses is incremented and compared to the maximum length of the stored tune:

```
ENDCHK  INC GESNO One more guess
       LDA DIGITS All digits guessed?
       CMP GESNO If not, get next key closure
       BNE KEY
```

If we are not done yet, a branch occurs back to label KEY. Otherwise, we have reached the end of a game and must signal either a "win" or a "lose" situation. The number of errors is checked to determine this:

```
LDA ERRS Get number of errors
       CMP #0 No error?
       BEQ WIN If not, player wins
```

If a "win" is identified, a branch occurs to label WIN. This will be described below. Let us examine now what happens in the case of a "lose":

```
LOSE  JSR LIGHT Show number of errors
```

The number of errors is displayed by lighting up the corresponding LED. Remember that the accumulator was conditioned prior to entering this routine and contained the value of ERRS, i.e., the number of errors so far.

Next, a sequence of eight descending tones is played. The top of the stack is used to contain the remaining number of tones to be played:

```
LOSELP  LDA #9 Play 8 descending tones
       PHA Save A on stack
       JSR PLAY Play tone
       PLA Restore A
```

Once a tone has been played, the remaining number of tones to be played is decremented by one and tested for "0":

```
SEC Set carry (for subtract)
       SBC #1 Subtract one
       BNE LOSELP
```

Exercise 8-5: Note how the top of the stack has been used as a tem-
porary scratch location. Can you suggest an alternative way to achieve the same result without using the stack?

Exercise 8-6: Discuss the relative merits of using the stack versus using other techniques to provide temporary working locations for the program. Are there potential dangers inherent in using the stack?

Eight successive tones are played. Then the two work variables, GESNO and ERRS, are reset to “0,” and a branch occurs back to the beginning of the program:

```
STA GESNO    Clear variables
STA ERRS
BEQ KEY
```

Let us examine now what happens in a “win” situation. All LEDs on the Games Board are turned on simultaneously:

```
WIN   LDA #$FF    It is a win: turn all LEDs on
      STA PORT1A
      STA PORT1B
```

Next, a sequence of eight ascending tones is played. The tone number is stored in the accumulator and will be used as an index by the PLAY subroutine to generate an appropriate note. As before, the top of the stack is used to provide working storage:

```
WINLP  LDA #1      A will be incremented to 9
      PHA
      JSR PLAY
      PLA
```

The number of tones which have been played is then incremented by 1 and compared to the maximum value of 9:

```
CLC    Clear carry for addition
ADC #01
CMP #10
```

As long as the maximum of 9 has not been reached, a branch occurs back to label WINLP:

```
BNE WINLP
```

Otherwise, a new game is started:

```
BEQ STRTJP    Double jump for restart
```

This completes the description of the main program. Three subroutines are used by this program. They will now be described.

The Subroutines

LIGHT Subroutine

This subroutine assumes that the accumulator contains the number of the LED to be lit. The subroutine will light up the appropriate LED on the Games Board. It will achieve this result by writing a “1” in the appropriate position in the accumulator and then sending it to the appropriate output port. Either Port 1A will be used (for LEDs 1 through 8) or Port 1B (for LED 9). The “1” bit is written in the appropriate position in the accumulator by performing a sequence of shifts. The number of shifts is equal to the position of the LED to be lit. Index register Y is used as a shift-counter. The number of the LED to be lit is saved in the stack at the beginning of the subroutine and will be restored upon exit. Note that this is a classic way to preserve the contents of an essential register during subroutine execution so that the contents of the accumulator will be unchanged upon subroutine exit. If this was not the case, the calling program would have to explicitly preserve the contents of the accumulator prior to calling the LIGHT subroutine. Then it might have to load it back into the accumulator prior to using another one of the routines, such as the PLAY routine. Because LIGHT and PLAY are normally used in sequence, it is more efficient to make it the subroutine's responsibility to save the contents of the accumulator. Let us do it:

```
LIGHT      PHA    Preserve A
```

The shift-counter is then set up:

```
TAY    Use Y as shift counter
```

and the accumulator is initialized to “0”:
LDA #0  Clear A

LED 9 is turned off in case it was lit:

STA PORT1B

The shifting loop is then implemented. The carry bit is initially set to "1," and it will be shifted left in the accumulator as many times as necessary:

SEC  Set carry
LTSHFT
ROL A
DEY
BNE LTSHFT

The correct bit pattern is now contained in the accumulator and displayed on the Games Board:

STA PORT1A

However, one special case may arise: if LED 9 has been specified, the contents of the accumulator are "0" at this point, but the carry bit has been set to "1" by the last shift. This case must be explicitly tested for:

BCC LTCC  Is bit 9 set?

If this situation exists, the accumulator must be set to the value "00000001," and output to Port 1B:

LDA #1
STA PORT1B  Turn LED 9 on

We finally exit from the routine without forgetting to restore the accumulator from the stack where it had been saved:

LTCC
PLA  Restore A
RTS

Exercise 8-7: List the registers destroyed or altered by this subroutine every time it is executed.

Exercise 8-8: Assume that register Y must be left unchanged upon leaving this subroutine. What are the required program changes, if any?

RANDOM Subroutine

This subroutine generates a new random number and returns its value in A. Its operation has been described in Chapter 4.

PLAY Subroutine

This subroutine will normally play the tone corresponding to the number contained in the accumulator. Optionally, it may be entered at location PLYTON and will then play the tone corresponding to the frequency set by the accumulator and corresponding to the length specified by the contents of memory location DUR. Let us examine it.

Index register Y is used as an index to the two tables required to determine the note duration and the note frequency. In this game, up to 9 notes may be played, corresponding to LEDs and keys 1 through 9. Index register Y is first conditioned:

PLYTON
TAY
DEY  Use tone # as index
Decrement to internal value

Note that the index register must be decremented by one. This is because key 1 corresponds to entry number 0 in the table, and so on. The duration and frequencies are obtained from tables DURTAB and NOTAB using the indexed addressing mode. They are stored respectively at locations DUR and FREQ:

PLYTON
LDA DURTAB,Y  Get duration
STA DUR  Save it
LDA NOTAB,Y  Get frequency
STA FREQ  Save it

The speaker is then turned off:

PLYTON
LDA #0
STA PORT3B  Set speaker Port 3B

Two loops will now be implemented. An inner loop will use register Y as the delay-counter to implement the correct frequency for the note.
Register X will be used in the outer loop and will generate the tone for the appropriate duration of time.

Let us condition the two counter registers:

\begin{verbatim}
       LDX DUR     Get duration in # of \( \frac{1}{2} \) cycles
       LDY FREQ    Get frequency
\end{verbatim}

Next, let us implement the inner loop delay:

\begin{verbatim}
       DEY          Waste time
       CLC
       BCC \(+2\)  
       BNE FL1     Delay loop
\end{verbatim}

Note that two "do-nothing" instructions have been placed inside the loop to generate a longer delay. At the end of this inner loop delay the contents of the output port connected to the loudspeaker are complemented in order to generate a square wave.

\begin{verbatim}
       EOR \#FF     Complement port
\end{verbatim}

Note that, once more, EOR \#FF is used to complement the contents of a register.

\begin{verbatim}
       STA PORT3B
\end{verbatim}

The outer loop can then be completed:

\begin{verbatim}
       DEX
       BNE FL2     Outer loop
       RTS
\end{verbatim}

**SUMMARY**

This program demonstrates how simple it is to implement electronic keyboard games that sound for input/output and that are challenging to adult players.

**Exercise 8-9:** The duration and frequency constants for the nine notes are shown in Figure 8.6. What are the actual frequencies generated by the program?
THE RULES

This game is inspired by the commercial game of MasterMind (trademarked by the manufacturer, Invicta Plastics, Ltd.). In this game, one or more players compete against the computer (and against each other). The computer generates a sequence of digits — for example, a sequence of five digits between “0” and “9” — and the player attempts to guess the sequence of five numbers in the correct order. The computer responds by telling the player how many of the digits have been guessed accurately, and how many were guessed in their correct location in the numerical sequence.

LEDs 1 through 9 on the Games Board are used to display the computer's response. A blinking LED is used to indicate that the player's guess contains a correct digit which is located in the right position in the sequence. A steadily lit LED is used to indicate a digit correctly guessed but appearing out of sequence. Several players can match their skills against each other. For a given complexity level — say, for guessing a sequence of seven digits — the player that can correctly guess the number sequence with the fewest guesses is the winner.

The game may also be played with a handicap whereby a given player has to guess a sequence of n digits while the other player has to guess a sequence of only n – 1 digits. This is a serious handicap, since increasing the level of difficulty by one is quite significant.

A TYPICAL GAME

Both audio and visual feedback are used to play this game.

The Audio Feedback

Every time that a player has entered his or her sequence of guesses, the computer responds by sounding a specific tone. A low tone indicates an incorrect guess; a high tone indicates that the sequence was guessed correctly.

The Visual Feedback

At the beginning of each game, LED #10 is lit, requesting the length of the sequence to be guessed. This is shown in Figure 9.1. The player then specifies the sequence length as a number from 1 through 9. Any other input will be ignored.

As soon as the length has been specified, for example, let's say the length “2” has been selected, LED #11 lights up. This means “Enter your guess.” (See Figure 9.2.) At this point the player enters his or her guess as a sequence of two digits. Let us now play a game.

The player types in the sequence “1,2.” A low tone sounds, LEDs 10 and 11 go out briefly, but nothing else happens. The situation is indicated in Figure 9.3. Since LEDs 1 through 9 are blank, there is no correct digit in the guess. Digits “1” and “2” must be eliminated. Let us try another guess.

We type “3,4.” A low tone sounds, but this time LED #1 is steadily on, as indicated in Figure 9.4. From this we know that either “3” or
"4" is one of the digits and that it belongs in the other position. Conversely, the sequence "4,3," must have one good digit in the right position. Just to be sure let us perform a test.

We now type "4,3." A low tone sounds, indicating that the sequence is not correct, but this time LED #1 is on and blinking. This proves that our reasoning is correct, and we proceed.

We now try "4,5." A high-pitched sound is heard and LEDs 1 and 2 light up briefly, indicating that those digits have been guessed correctly and that we have won our first game.

At the end of the game, the situation reverts to the one at the beginning, as indicated in Figure 9.1. Note that typing in a value other than "1" through "9" as a guess will restart the game.

There is a peculiarity to the game: if the number to be guessed contains two identical digits, and the player enters this particular digit in one of its two correct locations, the computer response will indicate this digit as being both the right digit in the right place and the right digit in the wrong place!

THE ALGORITHM

The flowchart for Mindbender is shown in Figure 9.5. Interrupts are used to blink the LEDs. Interrupts will be generated automatically by the programmable interval timer of VIA #1 at approximately 1/15th-of-a-second intervals.

Referring to Figure 9.5, all of the required registers and memory locations will be initialized first. Next (box 2 on the flowchart), the length of the sequence to be guessed is read from the keyboard. The validity bracket "1" to "9" is used to "filter" the player's input.

Next, a random sequence must be generated. In box 3 of the flowchart, a sequence of random numbers is generated and stored in a digit table, starting at address D1G0.

In box 5, the computer's sequence of numbers is compared — one number at a time — with the player's guess. The algorithm takes one digit from the computer sequence and matches it in order against every digit of the player sequence. As we have already indicated, this may result in lighting up two LEDs, if ever there are two or more identical digits in the number to be guessed and the player has specified only one digit. One digit may be flagged as being in the right place, and also as being correct but in the wrong location(s).

Note that, alternatively, another comparison algorithm could be used in which each digit of the player's sequence is compared in turn with each digit of the computer's sequence.

Once the digits have been compared, the resulting score is displayed on the LEDs (box 6). Finally, a test is made for a win situation (box 7), and the appropriate sound is generated (box 8).
6502 GAMES

166

Fig. 9.5: Mindbender Flowchart

THE PROGRAM

Data Structures

Two tables of nine entries are used to store, respectively, the computer’s sequence and the player’s sequence. They are stored starting at addresses DIG0 and ENTRY0. (See Figure 9.6.)

The Variables

Page 0 is used, as usual, to provide additional working registers, i.e., to store the working variables. The use of page 0 is indicated as a “memory map” in Figure 9.6. The first nine locations are used for the program variables. The function of each variable is indicated in the illustration and will be described in detail as we examine the program below. Locations “09” through “0E” are reserved for the random table used to generate the random numbers. Locations “1F” through “17” are used for the DIG0 table used to store the computer-generated sequence of random numbers. Finally, locations “18” and following are used to contain the sequence of digits typed by the user.

The memory locations used for addressing input/output and for interrupt vectoring are shown in Figure 9.7. Locations “A000” through “A005” are used to address Ports A and B of VIA #1 as well as timer T1. The memory map for a 6522 VIA is shown in Figure 9.8.

Location “A00B” is used to access the auxiliary control register, while location “A00E” accesses the interrupt-enable register. For a detailed description of these registers the reader is referred to the 6502 Applications Book (reference D302).

Memory locations “A67E” and “A67F” are used to set up the interrupt vector. The starting address of the interrupt-handling routine will be stored at this memory location. In our program, this will be address “03EA.” This is the routine in charge of blinking the LEDs. It will be described below. Finally, Port 3 is addressed at memory locations “AC00” and “AC02.”

Program Implementation

A detailed flowchart for the Mindbender program is shown in Figure 9.9. Let us now examine the program itself. (See Figure 9.13.)

The initialization block resides at memory addresses 0200-0239 hexadecimal and conditions interrupts and I/O. First, interrupts are conditioned. Prior to modifying the interrupt vector which resides at ad-
LENGTH OF SEQUENCE

DUR

TONE DURATION CONSTANT

XTEMP

TEMPORARY X REGISTER

YTEMP

TEMPORARY Y REGISTER

CNT

NUMBER OF MATCHES

MASKA

PATTERN FOR BLINKING LEDS ON A

MASKB

PATTERN FOR BLINKING LEDS ON B

FREQ

TONE FREQUENCY CONSTANT

CNT1

CORRECT DIGITS CORRECT PLACE

RND

RANDOM NUMBERS

DIG0

UP TO 9 DIGITS OF NUMBERS TO GUESS

ENTRY0

UP TO 9 DIGITS

Fig. 9.6: Low Memory Map

Fig. 9.7: High Memory Map
addresses "A67E" and "A67F" (see Figure 9.7) access to this protected area of memory must be authorized. This is performed by the ACCESS subroutine, which is part of the SYM monitor:

`JSR ACCESS`

Next, the new interrupt vector can be loaded at the specified location. The value "03EA" is entered at address IRQVEC:

```
LDA #$EA
STA IRQVEC
LDA #$03
STA IRQVEC
```

Now the internal registers of the 6522 VIA #1 must be conditioned to set up the interrupts. The interrupt-enable register (IER) will enable or disable interrupts. Each bit position in the IER matches the corresponding one in the interrupt flag register (IFR). Whenever a bit position is "0," the corresponding interrupt is disabled. Bit 7 of IER plays a special role. (See Figure 9.10.) When IER bit 7 is "0," each "1" in the remaining bit positions of IER will clear the corresponding enable flag. When IER bit 7 is "1," each "1" written in IER will play its normal role and set an enable. All interrupts are, therefore, disabled by setting bit 7 to "0" and all remaining bits in the IER to ones:

```
LDA #$7F
STA IER
```

Next, bit 6, which corresponds to the timer 1 interrupt, is enabled. In order to do this, bit 7 of IER is set to "1," as is bit 6:

```
LDA #$C0
STA IER
```

Next, timer 1 will be set in the "free-running mode." Remember that, with the 6522, the timer can be used in either the "one-shot" mode or the "free-running mode." Bits 6 and 7 of the auxiliary control register are used to select timer 1 operating modes. (See Figure 9.11.) In this instance, bit 7 is set to "0" and bit 6 is set to "1":

```
LDA #$40
STA ACR
```

Prior to using the timer in the output mode, its counter-register must be loaded with a 16-bit value. This value specifies the duration of the square pulse to be generated. The maximum value "FFFF" is used here:

```
LDA #$FF
STA T1LL
STA T1CH
```

The actual wave form from timer 1 is shown in Figure 9.12. In order to compute the exact duration of the pulse, note that the pulse dura-
1. 

START

2. 

GET KEY NUMBER

3. 

NUMBER DIGITS = KEY NUMBER

Fig. 9.9: Detailed Mindbender Flowchart

Fig. 9.9: Detailed Mindbender Flowchart (Continued)
tion will alternate between $n + 1.5$ cycles and $n + 2$ cycles, where $n$ is the initial value loaded in the counter register.

Next, interrupts are enabled:

CLI

and the three ports used by this program are configured in the appropriate direction:

| STA DDR1A | Output |
| STA DDR1B | Output |
| STA DDR3B | Output |

All LEDs are then cleared:

<table>
<thead>
<tr>
<th>ACR7</th>
<th>ACR6</th>
<th>MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT ENABLE</td>
<td>INPUT ENABLE</td>
<td>(ONE-SHOT)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>GENERATE TIME OUT INT WHEN TI LOADED PB7 DISABLED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>GENERATE CONTINUOUS INT PB7 DISABLED</td>
</tr>
<tr>
<td>(FREE RUN)</td>
<td>(FREE RUN)</td>
<td>(FREE RUN)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>GENERATE INT AND OUTPUT PULSE ON PB7 EVERYTIME TI IS LOADED = ONE-SHOT AND PROGRAMMABLE WIDTH PULSE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>GENERATE CONTINUOUS INT AND SQUARE WAVE OUTPUT ON PB7</td>
</tr>
</tbody>
</table>

LED 10 is now turned on in order to signal to the player that he or she should specify the number of digits to be guessed:

LDA #0
STA PORTIA
STA PORTIB

and the blink masks are initially set to all 0's:

STA MASKA
STA MASKB

The key pressed is read using the usual GETKEY routine:

JSR GETKEY Get # digits

A software filter is implemented at this point. The value of the key read from the keyboard is validated as falling within the range "1" through "9." If it is greater than 9, or less than 1, the entry is ignored:

CMP #10
BPL KEY1
CMP #0
BEQ KEY1
Once validated, the length specified for the sequence is stored at memory location DIGITS:

```
STA DIGITS
```

A sequence of random numbers must now be generated.

**Generating a Sequence of Random Numbers**

The initial random number is obtained from the counter and used to start the random number generator. The theory behind this technique has been described before.

Locations RND + 1, RND + 4, and RND + 5 are seeded with the same number:

```
LDA TILL
STA RND+1
STA RND+4
STA RND+5
```

Then a random number is obtained using the RANDOM subroutine:

```
LDY DIGITS
DEY
JSR RANDOM
```

Get # of digits to guess
Count to 0
Filling them with values

The resulting random number is set to a BCD value which guarantees that the last digit will be between 0 and 9:

```
SED
ADC #00
CLD
```

Decimal Adjust

It is then truncated to the lower 4 bits:

```
AND #$00001111
```

Once the appropriate random digit has been obtained, it is saved at the next location of the digit table, using index register Y as a running pointer:

```
STA DIG0,Y
```

The counter Y is then decremented, and the loop executed until all required digits have been generated:

```
DEY
BPL RAND
```

**Collecting the Player's Guesses**

Index register X will serve as a running pointer for the ENTRY table used to collect the player's guess. It is initialized to the value "0," and stored at memory location XTEMP:

```
EXTRA
LDA #0
Clear pointer
STA XTEMP
```

LEDs 10 and 11 are then turned on to signal the player that he or she may enter his or her sequence:

```
LDA #$00000110
STA PORT1B
```

The key pressed by the player is read with the usual GETKEY routine:

```
KEY2 JSR GETKEY
```

If the key pressed is greater than 9, it is interpreted as a request to restart the game:

```
CMP #10
BPL KEY1
```

Otherwise, the value of the index register X is retrieved from memory location XTEMP and is used to perform an indexed store of the accumulator to the appropriate location in the ENTRY table:

```
LDX XTEMP
STA ENTRY0,X Store guess in table
```

The running pointer is then incremented, and stored back in memory:
INX
STX XTEMP

Then, the value of the running pointer is compared to the maximum number of digits to be fetched from the keyboard and, as long as this number is not reached, a loop occurs back to location KEY2:

CPX DIGITS  All numbers fetched?
BNE KEY2    If not, get another

Once the player has entered his or her sequence, the digits must be compared to the computer-generated sequence. In anticipation of the display of a possible win the LEDs on the board are blanked and the masks are cleared:

LDX #0
STX PORTIA
STX PORTIB
STX MASKA
STX MASKB

Two locations in memory will be used to contain the number of correct digits and the number of correct digits in the correct location. They are initially cleared:

STX CNT    Number of matches
STX CNT1   Number of correct digits

Each entry of the DIG0 table will now be compared in turn to all entries of the ENTRY0 table. Each digit is loaded from the DIGIT table and immediately compared to the corresponding ENTRY contents:

DIGLP    LDA DIG0,X
        CMP ENTRY0,X

If it is not the right digit at the right place, there is no exact match. We will then check to see if the digit appears at any other place within the ENTRY table:

BNE ENTRYCMP

Otherwise, one more exact match is recorded by incrementing location CNT1, and the next digit is examined:

INC CNT1
BNE NEXTDIG

Let us examine now what happens when no match has occurred. The digit (of the number to be guessed) which has just been read and is contained in the accumulator should be compared to every digit within the ENTRY table. Index register Y is used as a running pointer, and the contents of the accumulator are compared in turn to each of the digits in ENTRY:

ENTRYCMP    LDY #0
ENTRYLP   CMP ENTRY0,Y
        BNE NEXTENT

If a match is found, memory location CNT is incremented and the next digit is examined:

INC CNT
BNE NEXTDIG

Otherwise, index register Y is incremented. If the end of the sequence is reached, exit occurs to NEXTDIG. Otherwise a branch back occurs to the beginning of the loop at location ENTRYLP:

NEXTENT    INY
        CPY DIGITS
        BNE ENTRYLP

Increment guess # pointer
All tested?
No: try next one

The next digit in table DIG must then be examined. The running pointer for DIG is contained in index register X. It is incremented and compared to its maximum value:

NEXTDIG    INX
        CPX DIGITS

Increment digit # pointer
All digits checked

If the limit has not been reached, a branch occurs back to the beginning of the outer loop at location DIGLP:
At this point, we are ready to turn on the LEDs to display the results to the player.

Displaying the Results to the Player

The total number of LEDs which must be turned on is obtained by adding the contents of CNT to CNT1:

```
CLC
LDA CNT
ADC CNT1
```

The total is contained in the accumulator and transferred into index register Y where it will be used by the LITE routine:

```
TAY
JSR LITE
```

The operation of the LITE routine will be described below. Its effect is to fill the accumulator with the appropriate number of ones in order to turn on the appropriate LEDs.

The pattern created by the LITE subroutine is then stored in the mask:

```
STA PORT1A
```

For the special case in which the result is 9, the carry bit will have been set. This case is explicitly tested:

```
BCC CC If carry 0, don’t light PB0.
```

and if the carry had been set to 1, Port B will be set appropriately so that LED #9 is turned on:

```
LDA #1 Turn PB0 on
STA PORT1B
```

Recall that once masks A and B have been set up, they will automatically be used by the interrupt handling routine which will cause the appropriate LEDs to blink.

```
CC LDY CNT1
JSR LITE
STA MASKA
BCC TEST
LDA #01
STA MASKB
```

The program must now test for a win or lose situation.

Testing for a Win or Lose Situation

The number of correct digits in the right places is contained in CNT1. We will simply compare it to the length of the sequence to be guessed:

```
TEST LDX CNT1
CPX DIGITS
```

If these numbers are equal, the player has won:

```
BEQ WIN
```

Otherwise, a low tone will be sounded. The tone duration constant is set to “72,” and its frequency value to “BE”:

```
BAD LDA #$72
STA DUR
LDA #$BE
```

The TONE subroutine is then used to generate the tone, as usual:

```
JSR TONE
```

Then a return occurs to the beginning of the program:

```
BEQ ENTER
```

If a win has occurred, a high-pitched tone will be generated. Its duration constant is set to “FF” and its pitch is controlled by setting the
frequency constant to "54":

WIN         LDA #$FF
            STA DUR
            LDA #$54

As usual, the TONE subroutine is used to generate the tone:

JSR TONE

The game is then restarted:

JMP KEY1

The Subroutines

Four routines are used by this program. They are: LITE, RANDOM, TONE, and INTERRUPT HANDLER. The RANDOM and TONE routines have been described in previous chapters and will not be described again here.

LITE Subroutine

When entering this subroutine, index register Y contains the number of LEDs which should blink. In order to make them blink it is necessary to load the appropriate pattern into the mask patterns called MASKA and MASKB. The appropriate number of 1's has to be set in these two locations. A test is first made for the value "0" in Y. If that value is found, the accumulator is cleared, as well as the carry bit (the carry bit will be used as an indicator for the fact that Y contained the value "9"):

LITE BNE STRTSH       Test Y for zero
            LDA #0
            CLC
            RTS

Otherwise, the accumulator is initially cleared, and the appropriate number of 1's is shifted left into the accumulator through the carry bit. They are introduced one at a time by setting the carry bit, then performing a left shift into A. Each time, index register Y is decremented and the loop is executed again as long as Y is not "0":

Note that a rotation to the left is used rather than a shift. If Y did contain the value "9," the accumulator A would be filled with 1's and the carry bit would also contain the value "1" upon leaving the subroutine.

The Interrupt Handler

This subroutine complements the LEDs each time an interrupt is received, i.e., every time timer 1 runs out. It is located at memory addresses "03EA" and following. Since the accumulator is used as a working register by the subroutine, it must be preserved upon entry and pushed into the stack:

PHA

The contents of Ports 1A and 1B will be read and then complemented. Recall that there is no complementation instruction on the 6502, so an exclusive OR will be used instead. MASKA and MASKB specify the bits to be complemented:

LDA PORT1A
EOR MASKA
STA PORT1A
LDA PORT1B
EOR MASKB
STA PORT1B

Also recall that the interrupt bit in the 6522 has to be cleared explicitly after every interrupt. This is done by reading the latch:

LDA TILL

Finally, the accumulator is restored, and a return occurs to the main program:
PLA
RTI

SUMMARY

In this program, we have used two new hardware resources in the 6522 I/O chip: the interrupt control and the programmable interval timer. Interrupts have been used to implement simultaneous processing by blinking the LEDs while the program proceeds, testing for a win or lose situation.

Exercise 9.1: Could you implement the same without using interrupts?

---

**Fig. 9.13: Mindbender Program**

---

**Fig. 9.13: Mindbender Program (Continued)**
; Routine to fill guess table/ users' guesses
02671 AP 00 ENTER LDA #0 ; Clear entry table pointer.
02679 BS 02 STA XTEMP
02681 A9 06 LDA $50000010 Iset User know that guesses
02682 00 00 ORA PORTB "Should be input.
02684 00 00 STA PORTB ...Without changing array.
02687 BD 00 AO JSR DKEY "Get guess.
0268A 09 00 CMP $10 "Is it greater than 9?
0268D 10 04 BPL KEY1 "If yes, restart game.
0268F A6 02 LDX XTEMP Get pointer for indexing
02692 00 10 STA ENTRY0+X "Store guess in table.
02695 00 00 INX Increment pointer.
02698 02 BS STA XTEMP
0269B 00 00 CPA DIGITS "Correct # of guesses fetched?
02831 D0 EE BNE KEY2 "If not, get another

; This routine compares users' guesses with digits
; for number to guess. For each correct digit in the
; incorrect place, a blinking led is lit. And for each
; incorrect digit in the wrong place, a nonblinking
; led is lit.
026A1 02 BS 00 LSC X;$ Clear following storages.
026A6 02 BS 00 STX PORTA 8 leds
026A8 02 BS 00 STX PORTB
026AD 02 BS 00 STA MARK
026B2 02 BS 00 STA CNT Count of matches.
026B7 02 BS 00 STA CNT1 8count of right digits.
026B9 02 BS 00 LDY X;$ Load list digit 1 for compares
026C0 02 BS 00 CMP ENTRY0+Y 8right guess/ right digit?
026C5 02 BS 00 BNE ENTRYEMP 8not is guess right digit/
; wrong place?
026CD 02 BS 00 INC CNT1 8one more right digits/right place
026D1 02 BS 00 BNE NEXTBIG 8examine next digit of number
026D7 02 BS 00 ENTRYCMP LDY $0 ; preset guess? for comparison
026D9 02 BS 00 BNE NEXTBIG 8Try next guess.
026DE 02 BS 00 entryMp CMP ENTRY0+Y 8right digit/wrong place
026F2 02 BS 00 BNE NEXTST 8examine next digit of number
026F8 02 BS 00 INC Ngext INY Increment guess pointer.
026F9 02 BS 00 CPA DIGITS 8all guesses tested?
026FB 02 BS 00 BNE NEXTBIG 8Try next guess.
02700 02 BS 00 CPA DIGITS 8all digits evaluated.
0270B 02 BS 00 BNE DIGIT CMP NEXTST 8check next digit.
02714 02 BS 00 BNE ENTRYMP ; Not ready for add.
0271D 02 BS 00 ACC CNT 8number of digits to add.
02724 02 BS 00 ACC CNT 8number of digits to subtract.
0272F 02 BS 00 JSR LITE 8Get pattern to light LED.
02736 02 BS 00 STA PORTA 8Turn LEDs on
0273D 02 BS 00 BCC CC 8If carry=0, don't light LED.
02740 02 BS 00 0A LDY #0 8LDA #1.
02743 02 BS 00 STA PORTB 8PT on.
02745 02 BS 00 CC LDY CNT 8Load # of LEDs to blink.
0274E 02 BS 00 BPL LED 8发生在无电路中.
02752 02 BS 00 STA MARK 8Start blink LED.
02755 02 BS 00 04 BCC TEST 8If carry=0, PRT won't blink.
02758 02 BS 00 A1 01 LDA #1 8LDA #1.
0275B 02 BS 00 0A STA PORTB 8PT on.
02760 02 BS 00 CC LDY CNT 8Load # of LEDs to blink.
02763 02 BS 00 BPL LED 8发生在无电路中.
02769 02 BS 00 STA MARK 8Start blink LED.
0276C 02 BS 00 04 BCC TEST 8If carry=0, PRT won't blink.
0276F 02 BS 00 A1 01 LDA #1 8LDA #1.
02772 02 BS 00 06 STA MARK

; Routine to test for win by checking if # of correct
; digits in correct places = number of digits. If win,
; a high pitched sound is generated. If any
; digit is wrong, a low sound is generated.
02781 02 BS 00 0D LDX CNT Load number of correct
digits
0278D 02 BS 00 0E CPA DIGITS "All guesses correct?
02790 02 BS 00 0F JSR DKEY "Get win.
02793 02 BS 00 07 8BAD LDA $72 8set up length of low tone.
02796 02 BS 00 08 STA DUR 8LDA $08 8value tone for low tone.
02799 02 BS 00 09 JSR TONE 8FGNAL BAD GUESSES & TONE
0279C 02 BS 00 0A STA ENTRY0+X 8Next digits fetched
0279F 02 BS 00 0F JSR FF 8High Tone
027A2 02 BS 00 0F 8LDA $95 8Tone value for high tone.
027A5 02 BS 00 00 JSR TONE 8Signal Win
027A8 02 BS 00 0E STA ENTRY0+X 8Next digits fetched
027AF 02 BS 00 0F JSR FF 8High Tone

; Routine to fill accumulator with '1' bits starting
; past the low order end, up to and including the
; first position corresponding to the # of leds to
; be lit or set to blinking.
027BA 02 BS 00 0D LITE BNE STRTSH 8If Y not zero, shift digits in
027BF 02 BS 00 0A LDA #0 8Special case: result is no ones.
027C3 02 BS 00 18 LDA 8Low sound
027C6 02 BS 00 60 RTS
027CA 02 BS 00 07 STRTSH LDA #0 8Clear a so pattern will show
027CB 02 BS 00 38 SHIFT SEC ;Mane a bit high
027C8 02 BS 00 24 RPL A 8Shift it to correct position
027D2 02 BS 00 88 LDA $88 8By looping to # of guess/digit
027D5 02 BS 00 18 HLT LDA $18 8Matches: AG passed in Y
027DB 02 BS 00 00 BNE SHIFT 8loop 'til done.
027DE 02 BS 00 60 RTS

; Random number generator
uses numbers A+B+C+D+E+ stored as random through
random: add B+C+D+ and places result in A+B, then
shifts a to B to C etc. The new random number
which is between 0 and 255 inclusive is in the
accumulator on exit
027E1 02 BS 00 38 random SEC 8Carry adds value 1
027E4 02 BS 00 5A LDA RND+1 8Add A+B+E and carry
027E7 02 BS 00 6C RND+4 8Acc RND+4
027EE 02 BS 00 0E RND+5 8Acc RND+5
027F1 02 BS 00 09 STA RND 8Acc RND+4
027F4 02 BS 00 24 LDX #4 8Shift numbers over
027F7 02 BS 00 09 RPL RDA RND+X
027FC 02 BS 00 0A STA RND+1 X
027FE 02 BS 00 85 EXE
02800 02 BS 00 10 F9 RPL RPL
03111 02 BS 00 60 RTS

; Tone generator routine.
; Duration of tone (number of cycles to create)
; Should be in 'our' on ENTRY, and the note value
; (frequency) in the accumulator.
0311B 02 BS 00 07 TONE STA FREQ
0311D 02 BS 00 07 TONE STA FREQ
0311F 02 BS 00 07 TONE STA FREQ

Fig. 9.13: Mindbender Program (Continued)
THE RULES

The standard game of Blackjack or “21,” is played in the following way. A player attempts to beat the dealer by acquiring cards which, when their face values are added together, total more points than those in the dealer’s hand but not more than a maximum of 21 points. If at any time the total of 21 is achieved after only two cards are played, a win is automatically declared for the player; this is called a Blackjack (the name of the game). Card values range from 1 through 11. In the standard version of Blackjack the house rules require the dealer to “hit” (take a card) if his/her hand equals 16 or fewer points, but prohibits him/her from taking a “hit” when his or her hand totals 17 or more points.

The version of Blackjack played on the Games Board differs slightly from the standard game of Blackjack. The single “deck” of cards used here contains cards with values from 1 through 10 (rather than 1 through 11), and the number of points cannot exceed 13 (as opposed to 21). The dealer in this variation of the game is the computer.

At the beginning of each hand, one card is dealt to the dealer and one to the player. A steady LED on the Games Board represents the value of the card dealt to the dealer (the computer). A flashing LED represents the card dealt to the player. If the player wants to be “hit” (i.e., receive another card) he/she must press key “C.” The player may hit several times. However, if the total of the player’s cards ever exceeds 13, the player has lost the round (“busted”) and he/she can no longer play. It is then the dealer’s turn. Similarly, if the player decides to pass (“stay”), it becomes the dealer’s turn. The dealer plays in the following manner: if the dealer’s hand totals fewer than 10
points, the computer deals itself one more card. As long as the hand does not exceed 13, the computer will check to see if it needs another card. Like the situation with the player, once the total of the computer's cards exceeds 13, it loses. No provision has been made for a bonus or an automatic win, which occurs whenever the player or the dealer gets exactly 13 points with only two cards (a Blackjack). This is left as an exercise for the reader. Once the dealer finishes its turn, assuming that it does not bust, the values of both hands are compared. If the dealer's total is greater than the player's, the player loses. Otherwise, the player wins. At the beginning of each series the player is allocated 5 chips (5 points). Each loss decreases this total by one chip; each win increases it by one. The game is over when the player goes broke and loses, or reaches a score of 10 and wins. After each play the resulting score is displayed as a number between 0 and 10 on the appropriate LED. Each time a player wins a hand, the left-most three LEDs of the bottom row light up. If the dealer wins the hand, the right-most LEDs light up. (See Figure 10.1.)

LEDs will light up in succession on the board as the computer deals itself additional cards until it goes over ten, reaches 13 exactly, or busts. Once the computer has stopped playing, any key may be pressed; the player's score will be displayed and the winner will be indicated through lit LEDs on the winner's side. The display will appear for approximately one second, then a new hand will be dealt.

Note that once the value of the computer's hand has reached a total greater than or equal to 10, it will do nothing further until a key is pressed. Let us follow this "typical game."

The initial display is shown in Figure 10.2. A steady LED is shown as a black dot, while a blinking LED is shown as a half dot. In the initial hand the computer has dealt itself a 1 and the player a 4. The player presses key "A" and receives an additional card. It is a 9. The situation is shown in Figure 10.3. It's a Blackjack and the player has won. The best the dealer can hope for at this point is to also reach 13.

A TYPICAL GAME

When playing a game against the dealer, the player will press key "A" to be "hit" (receive an additional card) until either a total of 13 is exceeded (a "bust"), or until the player decides that his or her total is close enough to 13 that he or she might beat the dealer. When the player makes this decision to stay, he or she must press key "C." This will start the dealer's turn, and all other keys will then be ignored.
Let us examine its response. To do this we must pass by hitting "C."
A moment later LED #3 lights up. The total of the computer's hand
now is $1 + 3 = 4$. It will deal itself another card. A moment later,
LED #7 lights up. The computer's total is now $4 + 7 = 11$. It stops.
Having a lower total than the player, it has lost. Let us verify it. We
press any key on the keyboard (for example, "0"). The result appears
on the display: LEDs 10, 11 and 12 light up indicating a player win,
and LED #6 lights up, indicating that the player's score has been in­
crease from 5 to 6 points. This information is shown in Figure 10.4. The

LED display then goes blank and a new hand is displayed. When there
is a draw, none of the LEDs in the bottom row light up and the score
is not changed. A new hand is dealt. (If the player busts, the dealer
wins immediately and a computer win is displayed.)

Let us play one more game. At the beginning of this hand the com­
puter has dealt itself a 5, and the player has a 6. The situation is shown
in Figure 10.5. Let us ask for another card. We hit key "A" and are
given a 7. This is almost unbelievable. We have thirteen again!! The
situation is shown in Figure 10.6. It is now the computer's turn. Let us
hit "C." LED #10 lights up. The computer has 15. It has busted. The
situation is shown in Figure 10.7. Let us verify it. We press any key on
the keyboard. The three left-most LEDs on the bottom row (LED 10,
11, and 12) light up and a score of 7 is displayed. This is shown in
Figure 10.8. A moment later the display goes blank and a new hand is
started.

Fig. 10.4: End of Turn: Dealer Loses

Fig. 10.5: Second Hand

Fig. 10.6: Blackjack Again

Fig. 10.7: Dealer Busts
THE PROGRAM

The detailed flowchart for the Blackjack program is shown in Figure 10.9, and the program is listed at the end of the chapter. As usual, a portion of page 0 has been reserved for the variables and flags which cannot be held in the internal registers of the 6502. This area is shown in Figure 10.10 as a “memory map.” These variables or flags are:

DONE: This flag is set to the value “0” at the beginning of the game. If the player goes broke, it will be set to the value “1111111.” If the player scores 10 (the maximum), it will be set to the value “1.” This flag will be tested at the end of the game by the ENDER routine which will display the final result of the game on the board and light up either a solid row of LEDs or a blinking square.

CHIPS: This variable is used to store the player’s score. It is initially set to the value “5.” Every time the player wins a hand it will be incremented by 1. Likewise, every time the player loses a hand, it will be decremented by 1. The game terminates whenever this variable reaches the value “0” or the value “10.”

MASKA, MASKB: These two variables are used to hold the masks or patterns used to blink the LEDs connected respectively to Port A and Port B on the Games Board.

PHAND: It holds the current hand total for the player. It is incremented every time the player hits (i.e., requests an additional card). card).

CHAND: This variable holds the current hand total for the computer (the dealer).

Fig. 10.9: Blackjack Flowchart
TEMP: This is a temporary variable used by the RANDOM routine to deal the next card to either player.

RND through RND + 5: These six locations are reserved for the random number generating routine called RANDER.

WHOWON: This status flag is used to indicate the current winner of the hand. It is initially set to "0," then decremented if the player loses or incremented if the player wins.

At the high end of memory the program uses VIA #1, the ACCESS subroutine provided by the SYM monitor, and the interrupt-vector at address A67E, as shown in Figure 10.11.

Let us now examine the program operation. For clarity it should be followed on the flowchart in Figure 10.9.

---

**Fig. 10.10: Low Memory Map**

- **C0**: DONE
- **C1**: CHIPS
- **C2**: MASKA
- **C3**: MASKB
- **C4**: PHAND
- **C5**: CHAND
- **C6**: TEMP
- **C7**: RND
- **C8**:
- **C9**:
- **CA**:
- **CB**:
- **CC**:
- **CD**: WHOWON

**Fig. 10.11: High Memory Map**

- **8886**: ACCESS
- **A000**: PORTB
- **A001**: PORTA
- **A002**: DDRB
- **A003**: DDBA
- **A004**: TILL
- **A005**: T1CH
- **A006**:
- **A007**:
- **A008**: UNUSED
- **A009**:
- **A00A**:
- **A00B**: ACR
- **A00C**: UNUSED
- **A00D**:
- **A00E**: IER
- **A00F**:
- **A67E**: INTVECH
- **A67F**: INTVECL
Program Initialization

The timer on 6522 VIA #1 will be used to generate the interrupts which blink the LEDs. These interrupts will cause a branch to location 03EA where the interrupt-handling routine is located. The first step is, therefore, to load the new value into the interrupt vector, i.e., “03EA,” at the appropriate memory location:

BLJACK JSR ACCESS Unprotect system memory
LDA #$EA Load low interrupt vector
STA INTVECL
LDA #$03 High vector
STA INTVECH

As described previously, the interrupt-enable register is first loaded with the value “01111111,” and then with the value “11000000” in order to enable the interrupt for timer 1:

LDA #$7F Clear timer interrupt-enable
STA IER
LDA #$C0 Enable timer 1 interrupt
STA IER

Loading the value “7F” clears bits 0 through 6, thereby disabling all interrupts. Then, loading the value “C0” sets bit 6, which is the interrupt-bit corresponding to timer 1. (See Figure 9.10.) As in the previous chapter, timer 1 is put in the free-running mode. It will then automatically generate interrupts which will be used to blink the LEDs. In order to set it to the free-running mode, bit 6 of the ACR must be set to “1”:

LDA #$40 Put timer 1
STA ACR In free run mode

The latches for timer 1 are initialized to the highest possible value, i.e., FFFF:

LDA #$FF Low latch of timer 1
STA T1LL
STA T1CH High latch and start timer

Finally, now that the timer has been correctly initialized, interrupts are enabled on the processor:

CLI Enable interrupts

LED Ports A and B configured as outputs (remember that the accumulator still contains the value “FF”):

STA DDRA
STA DDRB

As a precaution, the decimal flag is cleared:

CLD

The player’s score is initialized to the value 5:

LDA #5 Set player’s score to 5
STA CHIPS

The DONE flag is initialized to the value “0”:

LDA #0 Clear done flag
STA DONE

The LEDs on the board are cleared:

STA MASKA
STA MASKB
STA PORTA Clear LEDs
STA PORTB

And the WHOWON flag is also initialized to “0”:

STA WHOWON Clear flag

Dealing the First Hand

We are now ready to play. Let us deal one card to both the dealer and the player. The LIGHTR and the BLINKR subroutines will be used for that purpose. Each of these subroutines obtains a random
number and lights the corresponding LED. LIGHTR lights up a steady LED while BLINKR blinks the LED. These two subroutines will be described later. We set one LED blinking for the player:

```assembly
JSR BLINKR Set random blinking LED
```

and we save the first total for the current player’s hand:

```assembly
STA PHAND Store player’s hand
```

then we do the same for the computer:

```assembly
JSR LIGHTR
STA CHAND
```

**Hit or Stay?**

We will now read the keyboard. If the player presses “A,” this indicates a requested hit and one additional card must be dealt to the player. If “C” is pressed, the player “stays” (passes) and it becomes the computer’s turn to play. All other keys are ignored. Let us first obtain the key closure from the keyboard:

```assembly
ASK JSR GETKEY
```

The key value must now be compared to “A” and to “C”:

```assembly
CMP #$0A
BEQ HITPLR
CMP #$0C
BEQ DEALER
```

If any other key has been pressed, it will be ignored and a new key will be read:

```assembly
JMP ASK Invalid key, try again
```

At this point in the program, we will assume the situation warrants a “hit.” One more card must be dealt to the player. Let us set one more LED blinking. Naturally, the BLINKR subroutine, as well as the LIGHTR subroutine, are careful not to deal a card that has already been dealt. How this is achieved will be described later (this is the purpose of the SETBIT subroutine).

```assembly
HITPLR JSR BLINKR Set random LED
```

As soon as a new card has been dealt to the player, we compute the player’s new total for the current hand:

```assembly
CLC ADC PHAND Tally player’s hand
STA PHAND
```

The new total must be checked against the value “13.” As long as the player has 13 or less, he or she may play again, i.e., either be hit or stay. However, if the player’s score exceeds “13,” he or she busts and loses the play. Let us check:

```assembly
CMP #14 Check for 13
BCC ASK Ask if \( \leq 13 \)
JMP LOSE Busted
```

It is now the dealer’s turn. Since the computer is much faster than the player in deciding whether it wants to hit or to stay, we will first slow it down to provide more suspense to the game:

```assembly
DEALER JSR DELAY
```

The delay subroutine also extends the period of time between the successive decisions made by the computer to make the computer appear more “human-like.”

Before dealing another card to the computer (the dealer), let us examine its total. The house rule is that the dealer’s total cannot exceed “10.” (Naturally, other algorithms are available from Blackjack experts.) The computer hand is therefore checked against the value “10.” If this value is exceeded, a branch occurs to location WINNER where the winner will be decided. Otherwise, a new card will be dealt to the computer:

```assembly
LDA CHAND Check hand for limit
CMP #10
BCS WINNER Yes. Decide winner.
```
As long as the hand totals less than "10," the dealer requests a hit. A new card is dealt to the dealer in exactly the same way that it was dealt previously to the player:

```
JSR LIGHTR      Set random LED
```

The dealer's new total is computed:

```
CLC
ADC CHAND      Tally computer's hand
STA CHAND
```

Just as in the case of the player before, it is compared against the value "13" to determine whether or not the dealer has busted:

```
CMP #14       Is hand <= 13?
BCC DEALER    Yes: another hit?
JMP WIN       Busted: player wins
```

If the computer has busted, a jump occurs to location WIN which indicates a "win" by the player. Otherwise, a branch back to location DEALER occurs, where the computer will determine whether or not it wants to receive an additional card. Let us now determine the winner. Both hands are compared:

```
WINNER       LDA CHAND      Compare hands
CMP PHAND
```

There are three possible cases: equal scores, player wins, and player loses.

```
BEQ SCORER
BCC WIN
```

In the case that both scores are equal, a jump occurs to location SCORER which will display the current status. If the player wins, a branch occurs to location WIN and the sequence will be described below. First, let us examine what happens when the player loses.

**The Player Loses**

A special flag, called WHOWON, is used to store the status at the end of each play. It is decremented to indicate a loss by the player:

```
LOSE       DEC WHOWON
```

The player's score is decremented:

```
DEC CHIPS
```

The player's score must be compared to the value "0." If the player's score has reached "0," he or she is broke and has lost the game. In this case, the DONE flag is set to "11111111;" otherwise, it is not changed. Finally a jump occurs to SCORER where the final score will be displayed:

```
BNE SCORER   Player broke?
DEC DONE     Yes: set lose flag
JMP SCORER   Finish game
```

**Player Has Won**

Similarly, when the player wins, the WHOWON flag is set to "1":

```
WIN       INC WHOWON
```

The score is incremented:

```
INC CHIPS
```

It is then compared to the value "10":

```
LDA CHIPS
CMP #10    Chips = 10?
```

If the maximum score of "10" has been reached, the DONE flag is set.

```
BNE SCORER   Set done flag
INC DONE
```

Displaying the final status is accomplished by the SCORER routine. Remember that the final status will be displayed only at the player's request — when any key is pressed on the keyboard. Let us wait for
Before displaying the status, all LEDs on the board are turned off:

```
LDA #0
STA MASKA
STA MASKB
STA PORTA
STA PORTB
```

The player's score must now be displayed on the board. Let us read it:

```
LDX CHIPS
BEQ ENDER
```

If the player has no more chips, a branch occurs to location ENDER and the game will be terminated. Otherwise, the score is displayed. Unfortunately, LEDs are numbered internally "0" through "7," even though they are labeled externally "1" through "8." In order to light up the proper LED, the score must therefore first be decremented:

```
DEX
```

then a special subroutine called SETMASK is used to display the appropriate LED. On entry to the SETMASK routine, it is assumed that the accumulator contains the number of the LED to be displayed.

```
TXA
JSR SETMASK
```

Now that the proper mask has been created to display the score, we must indicate the winner. If the player won, the three left-most LEDs in the bottom row will be lit; if the computer won, the three right-most LEDs will be lit. If it was a tie, no LEDs will be lit on the bottom row. Let us see who won:

```
LDA WHOWON
BEQ ENDER
BMI SC
```

If the player lost, a branch occurs to address SC. If, on the other hand, the player won, the three left-most LEDs in the bottom row are lit:

```
LDA #$0E
JMP SC0
```

If the player lost, the three right-most LEDs are lit:

```
SC
LDA #$B0
```

Contained in the accumulator is the appropriate pattern to light the bottom row of LEDs, and this is sent to the Games Board:

```
SC0
ORA PORTB
STA PORTB
```

**End of a Play**

The ENDER routine is used to terminate each play. If the score was neither "0" nor "10," a new hand will be dealt:

```
ENDER
JSR DELAY2
LDA DONE
BNE EN0
JMP START
```

Otherwise, we check the DONE flag for either a player win or a player loss. If the player lost the game, the bottom row of LEDs is lit and the program ends:

```
EN0
BPL EN1
LDA #$BE
STA PORTB
RTS
```

In the case of a player win, a blinking square is displayed and the program is terminated:

```
EN1
LDA #$FF
STA MASKA
```
6502 GAMES

LDA #$01
STA MASKB
RTS

Subroutines

**SETHIT Subroutine**

The purpose of this subroutine is to create the pattern required to light a given LED. Upon entering the subroutine, the accumulator contains a number between “0” and “9” which specifies which LED must be lit. Upon exiting the subroutine, the correct bit is positioned in the accumulator. If the logical LED number was greater than “7,” the carry bit is set to indicate that output should occur on Port B rather than on Port A. Additionally, Y will contain the external value of the LED to be lit (1 to 10).

Let us examine the subroutine in detail. The LED number is saved in index register Y:

```assembly
SETHIT TAY  // Save logical number
```

It is then compared to the limit value “7.”

```assembly
CMP #8
BCC SBO
```

If the value was greater than 7, we subtract 8 from it:

```assembly
SBC #8  // Subtract if > 7
```

**Exercise 10-1:** Recall that SBC requires the carry to be set. Is this the case?

Now we can be assured that the number in the accumulator is between “0” and “7.” Let us save it in X:

```assembly
SBO TAX
```

A bit will now be shifted into the correct position of the accumulator. Let us first set the carry to “1”:

```assembly
SEC  // Prepare to roll
```

We clear the accumulator:

```assembly
LDA #0
```

then we roll in the bit to the correct position:

```assembly
SBLOOP ROL A
DEX
BPL SBLOOP
```

Note that index register X is used as a bit-counter. The accumulator is now correctly conditioned. The external number of the LED to be lit is equal to the initial value which was stored in the accumulator plus one:

```assembly
INY  // Make Y the external #
```

If LEDs 9 or 10 must be lit, the carry bit must be set to indicate this fact. Port B will have to be used rather than Port A:

```assembly
CPY #9  // Set carry for Port B
RTS
```

**Exercise 10-2:** Compare this subroutine to the LIGHT subroutine in the previous chapter.

**Exercise 10-3:** How was the carry set for LED #9 at the end?

**LIGHTR Subroutine**

This subroutine deals the next card to the dealer (computer). It must obtain a random number, then make sure that this card has not already been dealt, i.e., that it does not correspond to a card which has already been displayed on the board. If it has not already been displayed, the random number can be used as the value of the next card to be dealt. A steady LED will then be lit on the board.

Let us first get a random number:

```assembly
LIGHTR JSR RANDOM
```

It will be shown below that the RANDOM routine does not just ob-
tain a random number but also makes sure that it does not correspond to a card already used. All we have to do then is position the correct bit in the accumulator and display it. Let us use the SETBIT routine we have just described in order to position the bit in the accumulator:

JSR SETBIT

We must determine whether Port A or Port B must be used. This is done by testing the carry bit which has been conditioned by the SETBIT subroutine:

BCS LLO

We will assume that Port A must be used. The new bit will be added to the display by ORing it into Port A:

ORA PORTA
STA PORTA

The value of the card must be restored into the accumulator. It had been saved in the Y register by the SETBIT routine:

TYA
RTS

In case Port B is used, the sequence is identical:

LLO ORA PORTB
STA PORTB
TYA RTS

**BLINKER Subroutine**

This subroutine operates exactly like LIGHTR above except that it sets an LED flashing. Note that it contains the SETMASK subroutine which will set the proper LED flashing and exit with a numerical value of the LED in the accumulator:

```
BLINKR JSR RANDOM
SETMASK JSR SETBIT
```

**RANDOM Subroutine**

This subroutine will generate a random number between “0” and “9” which has not already been used, i.e., which does not correspond to the internal number of an LED that is already lit on the Games Board. The value of this number will be left in the accumulator upon exit. Let us obtain a random number:

```
RANDOM JSR RANDER Get 0-255 number
```

The RANDER subroutine is the usual random number generator which has been described in previous chapters. As usual, we must retain only a number between “0” and “9.” We will use a different strategy here by simply rejecting any number greater than “9” and asking for a new random number if this occurs:

```
AND #$0F
CMP #10
BCS RANDOM
```

**Exercise 10-4:** Can you suggest an alternative method for obtaining a number between “0” and “9”? (Hint: such a method has been described in previous chapters.)

A random number between “0” and “9” has now been obtained. Let us obtain the corresponding bit position which must be lit and save it in location TEMP:

```
JSR SETBIT Set bit in position
STA TEMP
```

We will now check to see if the corresponding bit is already lit on either
Port A or Port B. Let us first check to see if it is Port A or Port B:

BCS RN0  Determine Port A or B

Assuming that it is Port A, we must now find which LEDs in Port A are lit. This is done by combining the patterns for the blinking and steady LEDs, which are, respectively, in Mask A and Port A:

LDA MASKA  ORA PORTA  Combine Port and Mask

Then a check is made to see whether or not the bit we want to turn on is already on:

JMP RN1

If it is on, we must obtain a new random number between “0” and “9”:

RN1  AND TEMP
BNE RANDOM

If the bit was not already on, we simply exit with the internal value of the LED in the accumulator:

DEY
TYA
RTS

Similarly, if an LED on Port B had to be turned on, the sequence is:

RN0  LDA MASKB
ORA PORTB
AND TEMP
BNE RANDOM
DEY
TYA
RTS

RANDER Subroutine

This subroutine generates a random number between “0” and “255.” It has already been described in previous chapters.
used by the dealer. Most of the logical steps of the algorithm were accompanied by sound and light effects. Note how little memory is required to play an apparently complex game.

Exercise 10-6: Note that this program assumes that the contents of memory location RND are reasonably random at the beginning. If you would like to have a more random value in RND at the beginning of the game, you could suggest an additional instruction to be placed in the initialization phase of this program? (Hint: this has been done in previous programs.)

Exercise 10-7: In the ENDER routine are the instructions “BNE ENO” and “JMP START” both needed? If they are not, under what conditions would they be needed?

Exercise 10-8: “Recursion” describes a routine which calls itself. Is DELAY 2 recursive?

```assembly
; --- BLACKJACK PROGRAM ---
ACCESS = $9B66
INTECH = $A77E
INTECH = $A77F
IER = $A006
ACR = $A00B
TILL = $A004
TICH = $A005
DREX = $A003
DDBR = $A002
PORTA = $A001
PORTB = $A000
MASKA = $C2
MASKB = $C3
CHIPS = $C1
DONE = $C0
PHAND = $C4
CHAND = $C0
TEMP = $C5
RND = $C7
WHOSON = $C8
GETKEY = $100
, $200

; BLACKJACK GAME: USES A 'DECK' OF 10 CARDS, CARDS DEALT TO THE PLAYER ARE FLASHING LED'S. ONES IN THE COMPUTER'S HAND ARE STEADY. CARDS ARE DEALT BY A RANDOM NUMBER GENERATOR WHICH IS NON-REPEITIVE. NUMERICAL VALUES ARE KEPT IN ZERO PAGE LOCATION 'PHAND' AND 'CHAND'. PORTA AND PORTB ARE THE OUTPUT PORTS TO THE LED DISPLAY, MASKA AND MASKB ARE USED BY THE INTERRUPT ROUTINE TO FLASH SELECTED LED'S. 'DONE' AND 'WHOSON' ARE STATUS FLAGS TO DETERMINE END OF GAME AND WHO WON THE CURRENT HAND.
```

**Fig. 10.12: Blackjack Program**

---

**Fig. 10.12: Blackjack Program (Continued)**
Fig. 10.12: Blackjack Program (Continued)
Program (Continued)

; GAMES
0323:  20 47 03  RANDOM JSR RANDER GET 0-255 NUMBER
0326:  29 0F  AND $0F  IMASK HIGH NIBBLE
0321:  C9 0A  CMP $10  JBRACKET 0-9
0322:  B0 F7  BCS RANDOM
0326:  20 02  JSR SETBIT SET BIT IN POSITION
032C:  85 C4  STA TEMP SAVE IT
0331:  B0 08  BCD RNO DETERMINE PORT A OR B
0333:  A5 C2  LDA MASKA COMBINE PORT A AND MASK
0335:  D0 A0  ORA PORTA ORA PORTB
0339:  4C 40 03 JMP RN1
033B:  A5 C3 RNO LDA MASKB COMBINE PORT AND MASK
0340:  D0 A0  ORA PORTA
0340:  25 C6  RJN1 AND TEMP  FLOOK AT SPECIFIC BIT
0342:  D0 DF  BNE IF BIT SET ALREADY, TRY AGAIN
0344:  B8  DEY IMAGINE Y LOGICAL
0345:  9B  TYA EXIT WITH VALUE IN ACCUMULATOR
0346:  60  RTS

; GAMES
0347:  3B  RANDER SEC CARRY ADDS 1
034B:  A5 CB  LDA RND+1 ADD B++,F
034F:  A5 CC  ADC RND+1
0354:  A5 C7  SBC RND+2
0359:  A2 04  LDX $44
035D:  B5 C7  RLOOPE LDA RND+X
035F:  95 CB  STA RND+1,X
0364:  EA  DEY
0367:  10 F9  BPL RLOOPE
0359:  60  RTS

; IDELAY LOOP: DELAY2 IS SIMPLY TWICE THE TIME DELAY
; OF DELAY. GIVEN LOOP IS APPROX .5 SEC. DELAY.
035A:  20 5D 03  DELAY2 JSR DELAY
035D:  A9 FF  DELAY LDA #$FF /*SET VALUE FOR LOOPS*/
035F:  A0  TAY
0360:  AA  D0 TAX
0361:  C4  D1 DEX
0362:  A9 FF  LDA #$FF
0364:  D0 FB  BNE D1
0364:  D0 FF  BNE D0
0367:  E0 F7  BNE DO
0369:  60  RTS

; INTERRUPT ROUTINE: EXCLUSIVE OR'S THE OUTPUT
; PORTS WITH THE CORRESPONDING BLINKER MASKS EVERY
; TIME THE TIMER TIMES OUT TO FLASH SELECTED LED'S.
; AND REGISTERS ARE CHANGED, AND THE INTERRUPT
; FLAG IS CLEARED BEFORE EXIT.

=03EA
03EA:  4B  PHA SAVE ACCUMULATOR
03EB:  A0 01 A0  LDA PORTA COMPLEMENT PORTS WITH MASKS

Fig. 10.12: Blackjack Program (Continued)
11

TIC-TAC-TOE

THE RULES

Tic-Tac-Toe is played on a three-by-three sectioned square. An “O” symbol will be used to represent a move by the player and an “X” will be used to display a move by the computer. Each player moves in turn, and on every turn each player strategically places his or her symbol in a chosen section of the board. The first player to line up three symbols in a row (either horizontally, vertically or diagonally) is the winner. An example of the eight possible winning combinations is shown in Figure 11.1. Using our LED display, a continuously lit LED will be used to display an “X,” i.e., a computer move. A blinking LED will be used to display an “O,” i.e., the player’s move.

Either the player or the computer may make the first move. If the player decides to move first, he or she must press key “F.” If the computer is to move first, any other key should be pressed and the computer will start the game. At the end of each game a new game will start automatically. The computer is equipped with a variable IQ (intelligence) level ranging from one to fifteen. Every time the computer wins, its IQ level is reduced one unit. Every time the player wins, the computer’s IQ level is increased by one unit. This way, every player has a chance to win. A high tone is sounded every time the player wins and a low tone is sounded every time that the player loses.

A TYPICAL GAME

The display is initially blank. We will let the computer start. We do this by pressing any key but the key “F.” (If we press key “F,” then the player must go first.) Let us begin by pressing “0.” After a short pause the computer responds with a “chirp” and makes its move. (See Figure 11.2.)

Fig. 11.1: Tic-Tac-Toe Winning Combinations For a Player

Fig. 11.2: First Computer Move

An “X” is used to denote the computer’s moves. “O” will be used to denote our moves. Blank spaces are used to show unlit LEDs. Let
us move to the center and occupy position 5. (See Figure 11.3.) We press key “5.” A moment later, LED #1 lights up and a chirp is heard that indicates it is our turn to play. The board is shown in Figure 11.4.

![](image1)

Fig. 11.3: Our First Move

It is now our turn and we should block the computer to prevent it from completing a winning column: let us occupy position 4. We press key “4.” A moment later, LED #6 lights up and a chirp is heard. The situation is shown in Figure 11.5.

![](image2)

Fig. 11.5: After the Computer’s Third Move

We play in position 2. The computer reacts by playing in position 8. This is shown in Figure 11.6. We prevent the computer from completing a winning row by playing in position 9. The computer responds by occupying position 3. This is shown in Figure 11.7. This is a draw situation. Nobody wins, all the LEDs on the board blink for a moment, and then the board goes blank. We can start another game.

![](image3)

Fig. 11.6: After the Computer’s Fourth Move

![](image4)

(DRAW)

Fig. 11.7: After the Computer’s Fifth Move

Another Game

This time we are going to start and, hopefully, win! We press “F” to start the game. A chirp is heard, confirming that it is our turn to play. We play in position 5. The computer responds by occupying square 3. The chirp is heard, announcing that we can play again. The situation is shown in Figure 11.8. We play in position 4. The computer responds by occupying square 6. This is shown in Figure 11.9. This time we must block the computer from completing the column on the
Since the computer was "smart enough" to move into a diagonal position after we occupied the center position, we did not win. Note: if we keep trying, at some point the computer will play one of the side positions (2, 4, 6, or 8) rather than one of the corners and we will then have our chance to win. Here is an example.

We move to the center. The computer replies by moving into position 6. The situation is shown in Figure 11.12. We move to square 1; the computer moves to square 9. This is shown in Figure 11.13. We
move to square 3; the computer moves to square 7. This is shown in Figure 11.14. This time we make the winning move by playing into square 2. The situation is shown in Figure 11.15. Note that if we start playing and if we play well, the result will be either a draw or a win. With Tic-Tac-Toe, the player who starts the game cannot lose if he or she makes no mistakes.

![Figure 11.14: Move 3](image)

![Figure 11.15: "We Win!"](image)

THE ALGORITHM

The algorithm for the Tic-Tac-Toe program is the most complex of those we have had to devise so far. It belongs to the domain of so-called "artificial intelligence." This is a term used to denote the fact that the functions performed by the program duplicate the mental activity commonly called "intelligence." Designing a good algorithm for this game in a small amount of memory space is not a trivial problem. Historically, many algorithms have been proposed, and more can be found. Here, we will examine two strategies in detail, and then select and implement one of them. Additional exercises will suggest other possible strategies.

Strategy to Decide the Next Move

A number of strategies may be used to determine the next move to be made by the computer. The most straightforward approach would be to store all possible patterns and, the best response in each case. This is the best method to use from a mathematical point of view as it guarantees that the best possible move will be made every time. It is also a practical approach because the number of combinations on a 3 x 3 board is limited. However, since we have already learned to do table lookups for other games, such an approach would not teach us as much about programming. It might also not be considered "fair." We will, therefore, investigate other methods applicable to a wider number of games, or to a larger board.

Many strategies can be proposed. For example, it is possible to consider a heuristic strategy in which the computer learns by doing. In other words, the computer becomes a better player as it plays more games and learns from the mistakes it makes. With this strategy the moves made by the computer are random at the beginning of the game. However, provided that a sufficient amount of memory is available, the computer remembers every move that it has made. If it is led into a losing situation, the moves leading to it are thrown out by the computer as misjudged moves, and they will not be used again in that sequence. With time and a reasonable "learning" algorithm this approach will result in the construction of decision tables. However, this approach assumes that a very large amount of memory is available. This is not the case here. We want to design a program which will fit into 1K of memory. Let us look at another approach.

Another basic approach consists of evaluating the board after each move. The board should be examined from two standpoints: first, if there are two "O"s in a row, it is important to block them unless a win can be achieved with the current move. Also, the win potential of every board configuration should be examined each time: for example, if two "X"s are in a row, then the program must make a move in order to complete the row for a win. Naturally these two situations are easy to detect. The real problem lies in evaluating the potential of every square on the board in every situation.

An Analytical Algorithm

At this point, we will show the process used to design an algorithm along very general guidelines. After that, as we discover the weaknesses of the algorithm, we will improve upon it. This will serve as an ex-
ample of a possible approach to problem-solving in a game of strategy.

**General Concept**

The basic concept is to evaluate the potential of every square on the board from two standpoints: "win" and "threat." The **win potential** corresponds to the expectation of winning by playing into a particular square. The **threat potential** is the win potential for the opponent.

We must first devise a way to assign a numerical value to the combinations of "O"s and "X"s on the board. This must be done so that we can compute the strategic value, or "potential," of a given square.

**Value Computation**

For each row (or column or diagonal), four possible configurations may occur — that is, if we exclude the case in which all three positions are already taken and we cannot play in a row. These configurations are shown in Figure 11.16. Situation "A" corresponds to the case in which all three squares are empty. Clearly, the situation has some possibilities and we will start by assigning the value "one" to each square in that case. The next case is shown in row "B" of Figure 11.16; it corresponds to the situation in which there is already an "X" in that row. If we were to place a second "X" in that row, we would be very close to a win. This is a desirable situation that has greater value than the preceding one. Let us add "one" to the value of each free square because of the presence of the "X"; the value of each square in that instance will be "two."

Let us now consider case "C" in Figure 11.16, in which we have one "X" and one "O." The configuration has no value since we will never be able to win in that particular row. The presence of an "O" brings the value of the remaining square down to "zero."

Finally, let us examine the situation of row "D" in Figure 11.16, where there are already two "X"s. Clearly, this is a winning situation and it should have the highest value. Let us give it the value "three."

The next concept is that each square on the board belongs to a row, a column, and possibly a diagonal. Each square should, therefore, be evaluated in two or three directions. We will do this and then we will total the potentials in every direction. For convenience, we will use an evaluation grid as shown in Figure 11.17. Every square in this grid has been divided into four smaller ones. These internal squares are used to display the potential of each square in each direction. The square
labeled "H" in Figure 11.17 will be used to evaluate the horizontal row potential. "V" will be used for the vertical column potential. "D" will be used for the diagonal potential. "T" will be used for the total of the previous three squares. Note that there is no diagonal value shown for four of the squares on the board. This is because they are not placed on diagonals. Also note that the center square has two diagonal values since it is at the intersection of two diagonals.

Once our algorithm has computed the total threat and win potentials for each square, it must then decide on the best square in which to move. The obvious solution is to move to the square having the highest win or threat potential.

Now we shall test the value of our algorithm on some real examples. We will look at some typical board configurations and evaluate them by using our algorithms to check if the moves it generates make sense.

**A Test of the Initial Algorithm**

Let us look at the situation in Figure 11.18. It is the player's turn ("O") to play. We will evaluate the board from two standpoints: potential for "X" and threat from "O." We will then select the square that has the highest total in each of the two grids generated and make our move there.

```
<table>
<thead>
<tr>
<th></th>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*Fig. 11.18: Test Case 1*

Let us first complete the evaluation grid for the first row. Since there is an "O" in the first row, the horizontal potential for the player is zero (refer to row C, Figure 11.16 and look up the value of this configuration). This is indicated in Figure 11.19. Let us now look at row 2: it contains two blank squares and an "X." Referring to line B of Figure 11.16, the corresponding value is "two." It is entered at the appropriate location in the grid, as shown in Figure 11.20. Finally, the third row is examined, and since there is an "O" in it, the row potential is "zero," as indicated in Figure 11.20. The process is then repeated for the three columns. The result is indicated in Figure 11.21.

The value of each square of column 1 is "zero," since there is an "O" at the bottom. Similarly, for column 2 the value is also "zero," and for column 3 it is "one" for each square, since all three squares are open (blank). (Refer to line A in Figure 11.16.)

The process is repeated for each of the two diagonals and the results are shown in Figure 11.22. Finally, the total is computed for each square. The results are shown in Figure 11.23. Remember that the total appears in the bottom right-hand corner of each square.

It can be seen that at this point, two squares (indicated by an arrow in Figure 11.23) have the highest total, "three." This indicates where
we should play. But wait! We have not yet examined the threat, i.e., the potential from our opponent “O.”

We will now evaluate the threat posed by “O” by again computing the potential of each square on the board, but this time from “O’s” standpoint. The position values for the six meaningful combinations are indicated in Figure 11.24. When we apply this strategy to our evaluation grid, we obtain the results shown in Figure 11.25. The square with the highest score is the one indicated by the arrow. It scores “four,” which is higher than the two previous squares that were determined when we evaluated the potential for “X.”

Using our algorithm, we decide that the move we should make is to play into square 1, as indicated in Figure 11.26.

Let us verify whether this was indeed the appropriate move, assuming that each player makes the best possible move. A continuation of the game is shown in Figure 11.27. It results in a draw.
Let us now examine what would have happened if we had not evaluated the threat and played only according to the highest potential for “X” as shown in Figure 11.23. This alternative ending for the game is shown in Figure 11.28. This game also results in a draw. In this instance, then, the square with the value “four” did not truly have a higher strategic value than the one with the value “three.” However, our algorithm worked.

Let us now test our algorithm under more difficult circumstances.

Improving the Algorithm

In order to test our algorithm, we should consider clear-cut situations in which there is one move that is best. To begin, we will assume that it is the player’s turn. The first test situation, evaluated for “X,” is illustrated in Figure 11.29, and the potential for “O” is shown in Figure 11.30. This time we have a problem. The highest overall potential is “four” for “X” in the lower right corner square. If the computer moved there, however, the player would win! At this point our algorithm should be refined.

We should note that whenever there are already two “X”s in a row the configuration should result in a very high potential for the third square. We should therefore assign it a value of “five” rather than
“three” to ensure that we move there automatically. We have thereby identified and made our first improvement to the algorithm.

The second test situation is shown in Figure 11.31. Our algorithm assigns the value “six” to the lower right corner square (as indicated by an arrow in Figure 11.31). This is clearly the correct move. It works! Now, let us test the improvement we have made.

**The First Move**

When the board is empty, our algorithm must decide which square should be occupied first. Let us examine what this algorithm does. (The results are shown in Figure 11.32.) The algorithm always chooses to move to the center. This is reasonable. It could be shown, however, that it is not indispensable in the game of Tic-Tac-Toe. In fact, having the computer always move to the center makes it appear “boring,” or simply “lacking imagination.” Something will need to be done about this. This will be shown in the final implementation.

**Another Test**

Let us try one more simple situation. This situation is shown in Figure 11.33. Again, the recommended move is a reasonable one. The reverse situation is shown in Figure 11.34 and does, indeed, lead to a certain win. So far, our algorithm seems to work. Let us try a new trap.

**A Trap**

The situation is shown in Figure 11.35. It is now “X’s” turn to play. Using our algorithm, we will move into one of the two squares having
the total of "four." This time, however, such a move would be an error! Assuming such a move, the end of the game is shown in Figure 11.36. It can be seen that "O" wins. The move by "X" was an incorrect choice if there was a way to get at least a draw. The correct move that would lead to a draw is shown in Figure 11.37. This time, our algorithm has failed. Following is a simple analysis of the cause: it moved to a square position of value "four" corresponding to a high level of threat by "O," but left another square with an equal threat value unprotected (see Figure 11.35). Basically, this means that if "O" is left free to move in a square whose threat potential is equal to "four," it will probably win. In other words, whenever the threat posed by "O" reaches a certain threshold, the algorithm should consider alternative strategies. In this instance, the strategy should be to place an "X" in a square that is horizontally or vertically adjacent to the first one in order to create an imminent "lose threat" for "O," and thereby force "O" to play into the desired square. In short, this means that the algorithm should analyze the situation further or better still, analyze the situation one level deeper, i.e., one turn ahead. This is called two-ply analysis.
In conclusion, our algorithm is simple and generally satisfactory. However, in at least one instance, Trap 3 in Figure 11.35, it fails. We must therefore, include either a special consideration for this case, or we must analyze the situation one turn ahead every time and look at what would happen if we were to place an "X" or an "O" in every one of the available squares. The latter is actually the "cleanest" solution. Ideally, we should analyze all of the possible sequences until an end-of-game situation is obtained. The programming complexity, the storage required, and the time that would be needed to analyze the situations would, however, make this approach impractical. In a more complex game, such as chess or checkers, it would be necessary to use such a multi-ply analysis. For example, if a row consists of a move by the player and two empty squares, its "row-sum" is "one." If the player has played twice, its row-sum is "two." If the player has played three times, the row-sum is "three." Since "three" is the highest total that can be achieved in rows where only the player has played, the value of "four" has been assigned to a computer move. For example, if the value of a row is "five," we know that there is one computer move ("X"), one player move ("O"), and one empty square. The six possible patterns are shown in Figure 11.38. It can readily be seen that the row-sum values of "two" or "eight" are winning situations. A row-sum value of "five" is a blocked position, i.e., one that has no value for the player. If a win situation is not possible, then the best potentials are represented by either a value of "one" or a value of "four" depending on whose turn it is to play.

The Ad Hoc Algorithm

This algorithm assigns a value to each square on the board depending on who has played there. Initially a value of "zero" is assigned to each square on the board. Every time the player occupies a square, however, the corresponding value of the square becomes "one." Every time the computer occupies a square, the value of that square becomes "four." This is illustrated in Figure 11.38. The value of "four" has been chosen so that it is possible to know the combination of moves in that row just by looking at the total of every row. For example, if a row consists of a move by the player and two empty squares, its "row-sum" is "one." If the player has played twice, its row-sum is "two." If the player has played three times, the row-sum is "three." Since "three" is the highest total that can be achieved in rows where only the player has played, the value of "four" has been assigned to a computer move. For example, if the value of a row is "five," we know that there is one computer move ("X"), one player move ("O"), and one empty square. The six possible patterns are shown in Figure 11.38. It can readily be seen that the row-sum values of "two" or "eight" are winning situations. A row-sum value of "five" is a blocked position, i.e., one that has no value for the player. If a win situation is not possible, then the best potentials are represented by either a value of "one" or a value of "four" depending on whose turn it is to play.

The algorithm is based on such observations. It will first look for a win by checking to see if there is a row-sum of value "eight." If this is the case, it will play there. If not, the algorithm will check for a so-called "trap" situation in which two intersecting rows each have a computer move in them and nothing else (the algorithm is always used..."
Fig. 11.38: Row-sums

for the computer’s benefit). This is illustrated in Figure 11.39. By examining Figure 11.39, it becomes clear that each unoccupied square that belongs to two rows having a row-sum of “four” is a trap position where the algorithm should play. This is exactly what it does.

The complete flowchart for the board analysis is shown in Figure 11.40. Now, let us examine it in more detail. Remember that it is always the computer’s turn when this algorithm is invoked.

First, it checks for a possible immediate win. In practice, we will examine all row-sums and look for one which has a total of “eight.” This would correspond to a case where there are two computer moves in the same row with the last square being empty. (Refer to Figure 11.38.)

Next, we will check for a possible player win. If the player can win with the next move, the algorithm must block this move. To do so, it should scan the row-sums and look for one that has a total of “two,” which would indicate a winning combination for the player. (Refer to Figure 11.38.)

At this point the algorithm should check to see if the computer can play into any of the trap positions defined above. (See Figure 11.39 for an example.)

One more feature has been built into the algorithm: the computer is equipped with a variable IQ level, i.e., with a variable level of intelligence. The above moves are ones that any “reasonable computer” must make. From this point on, however, the algorithm can let the computer make a few random moves and even possible mistakes if its intelligence level is set to a low level. In order to provide some variety to the game, we will obtain a random number, compare it to the IQ, and vary our play depending upon the results. If the IQ is set to the maximum, the program will always execute the right branch of the flowchart; however, if the IQ is not set to the maximum, it will sometimes execute the left branch. Let us follow the right branch of the flowchart. At this point, we will check for two special situations that correspond to moves #1 and #4 in the game.

For the first situation, i.e., the first move in a game, the algorithm will occupy any position on the board. That way, its behavior will be different every time and, thus, appear “intelligent.”
For the next situation we must look at move #4. It is the computer's turn. In other words, the player started the game (move #1), the computer responded (move #2), then the player made his or her second move (move #3), and it is now the computer's turn. In short, in the game thus far, the player has played twice and the computer has played once. At this point, we want to check to see if the first three moves have all been made along one of the diagonals. If so, since the player has made two moves and the computer has made one, the row-sum of one of the diagonals will be "six." The algorithm must check explicitly for this. If the first 3 moves have all been made along a
diagonal, the computer must move to a side position. This is a special situation which must be built into the algorithm, or it cannot be guaranteed that the computer (assuming the highest IQ level) will win every time. This situation is illustrated in Figure 11.41. Note that if straightforward logic was used, the algorithm would play into one of the free corners since a threat exists from the player that he or she might play there, and thereby set up a trap situation. The results of such an action are shown in Figure 11.42. By looking at this illustration, it can be seen that such a move would result in a loss. However, let us examine what happens if we play on one of the sides. This situation is illustrated in Figure 11.43; it results in a draw. This is clearly the move that should be made. This is a relatively little-known trap in the game of Tic-Tac-Toe, and a provision must be built into the algorithm so that the computer will win.

![Fig. 11.41: The Diagonal Trap](image)

![Fig. 11.42: Falling Into the Diagonal Trap](image)

![Fig. 11.43: Playing to the Side](image)

If it was not the fourth move, or if there was not a diagonal trap set, the next thing the computer should do is to check to see if the player can set a trap. (Refer to the flowchart in Figure 11.40.) If the player can set a trap, the computer plays in the appropriate square to block it. Otherwise, the computer moves to the center square, if available; if that is not possible, it moves randomly to any position. Since this algorithm was built in an *ad hoc* fashion, it is difficult to prove that it wins or achieves a draw in all cases. It is suggested that you try it on a board or that you try out the actual program on the Games Board. You will discover that in all conditions under which it has been tested, the computer always wins or achieves a draw. If the computer keeps winning, however, its IQ level will drop, and eventually it will allow the player to win. As an example, some sequences obtained on the actual board are shown in Figure 11.44.
Suggested Modifications

Exercise 11-1: Designate a special key on the Games Board that, when pressed will display the computer's IQ level.

Exercise 11-2: Modify the program so that the IQ level of the computer can be changed at the beginning of each game.

Credits

The ad hoc algorithm which was described in this section is believed to be original. Eric Novikoff was the main contributor. "Scientific American" (selected issues from 1950 through 1978), as well as Dr. Harvard Holmes must also be credited with providing several original ideas.

Alternative Strategies

Other strategies can also be considered. In particular, a short program can be designed by using tables of moves that correspond to various board patterns. The tables can be short because when symmetries and rotations are taken into account, the number of situations that can be represented is limited. This type of approach results in a shorter program, however, the program is somewhat less interesting to design.

Exercise 11-3: Design a Tic-Tac-Toe program using this type of table.

THE PROGRAM

The overall organization of the program is quite simple. It is shown in Figure 11.42. The most complex part is the algorithm that is used to determine the next move by the computer. This algorithm, called "FINDMOVE," was previously described.

Let us now examine the overall program organization. The corresponding flowchart is shown in Figure 11.45.

1. The computer IQ level is set to 75 percent.
2. The user's keystroke is read.
3. The key is checked for the value "F." If it is an "F," the player starts; otherwise the computer starts. Depending on the value of the key pressed, the flowchart continues into boxes 4 or 5, then to 6.
If the player starts (PLAYER is not equal to "0"), then we move to the left side of the flowchart.

7. The key, pressed by the player specifying his or her move, is read and the move is displayed on the board.

8. The corresponding LED is lit on the board. It then becomes the computer's turn to play and the variable PLAYER is set to "0" in box 9.

When exiting from box 6, if it is the computer's turn, we move to box 10.

11. The next move to be made by the computer must be computed at this time. This is the complex algorithm we have described above.

12. PLAYER is reset to "one" to reflect the fact that it is now the player's turn.

After either party has moved, the board is checked for a winning se-
sequence of lights in box 13. If there is not a winning sequence of lights, we move to the left on the flowchart.

14. We next check to see if all moves have been exhausted: we check for move #9. If the ninth LED is lit and a winning situation has not been detected, it is a draw, and all lights on the board must be flashed.

15. We flash all the LEDs on the board. Then, we return to box 6 and the next player plays.

When exiting from box 13, if there is a win situation, this fact must be displayed:

16. All of the lights are blanked except for the winning three LEDs, Next, it must be determined by the algorithm whether the player or the computer has won.

17. A determination is made as to whether it was the player or the computer who won. If the computer has won, we branch to the right on the flowchart.

18. A low frequency tone is sounded.

19. The computer's IQ is decremented (to a minimum of 0).

The situation for a player win, shown in boxes 20 and 21, is analogous.

The general program flow is straightforward. Now, we shall examine the complete information. The subroutine which analyzes the board situation is called “ANALYZE” and uses “UPDATE” as a subroutine to compute the values of various board positions.

**Data Structures**

The main data structure used by this program is a linear table with three entry points that are used to store the eight possible square alignments on the board. When evaluating the board, the program will have to scan each possible alignment for three squares every time. In order to facilitate this process, all possible alignments have been listed explicitly, and the memory organization is shown in Figure 11.46.

The table is organized in three sections starting at RWPT1, RWPT2, and RWPT3 (RWPT stands for “row pointer”). For example, the first elements RWPT1, RWPT2, and RWPT3, for the first three-square sequence are looked at by the evaluation routine. The sequence is: "0, 3, 6," as indicated by the arrows in Figure 11.43. The next three-square sequence is obtained by looking at the second entry in each RWPT table. It is "1, 4, 7," which is, in fact, the second column on our LED matrix.
The table has been organized in three sections in order to facilitate access. To be able to access all of the elements successfully, it will be necessary to keep a running pointer that can be used as an index for efficient table access. For example, if we number our generalized rows of sequences from 0 to 7, "row" 3 will be accessed by retrieving elements at addresses RWPT1 + 3, RWPT2 + 3, RWPT3 + 3. (It is the sequence "0, 1, 2," as seen in Figure 11.46.)

**Memory Organization**

Page 0 contains the RWPT table which has just been described, as well as several other tables and variables. The rest of the low memory is shown in Figure 11.47.

The GMBRD table occupies nine locations and stores the status of the board at all times. A value of "one" is used to indicate a position occupied by the player, and a value of "four" indicates a position occupied by the computer.

The SQSTAT table also occupies nine words of memory and is used to compute the tactical status of the board.

The ROWSUM table occupies eight words and is used to compute the value of each of the eight generalized rows on the square.

The RNDSCR table occupies six words and is used by the random number generator.

The remaining locations are used by temporary variables, masks, and constants, as indicated in Figure 11.47. The role of each variable or constant will be explained as we describe each routine in the program.

**High Memory**

High memory locations are essentially reserved for input/output devices. Ports 1 and 3 are used, as well as interrupts. The corresponding memory map is shown in Figure 11.48. The interrupt-vector resides at addresses A67E and A67F. It will be modified at the beginning of the program so that interrupts will be generated automatically by the interval timer. These interrupts will be used to blink the LEDs on the board.

**Detailed Program Description**

At the beginning of each game, the intelligence level of the computer is set at 75 percent. Each time that the player wins, the IQ level
START LDA #12
STA INTEL Set IQ at 75%

Initialization occurs next:

RESTRT JSR INIT

Let us examine the INIT subroutine which has just been called. It resides at address 0050 and appears on lines 0345 and following on the program listing. The first action of the initialization subroutine is to clear all low memory locations used by program variables. The locations to be cleared are those between CLRST and CLREND (see lines 41 and 57 of the program listing). Note that a seldom-used facility of the assembler — multiple labels for the same line — has been utilized to facilitate the clearing of the correct number of memory locations. Since it may be necessary to introduce more temporary variables in the course of program development, a specific label was assigned to the first location to be cleared, CLRST (memory location 18), and another to the last location to be cleared (CLREND). For example, memory location 18 corresponds both to CLRST and to GMBRD. The clearing operation should start at address CLRST and proceed forward forty locations (CLREND-CLRST). Thus, we first load the number of locations to be cleared into index register X, then we use a loop to clear all of the required locations:

INIT
LDA #0
LDX #CLREND-CLRST
CLRALL
STA CLRST,X Clear location
DEX
BPL CLRALL

After low memory has been cleared, the two starting locations for the random number generator must be seeded. As usual, the low-counter of timer 1 is used:

LDA T1LL
STA RNDSCR + 1
STA RNDSCR + 4
Ports 1A, 1B, and 3B are then configured as outputs. The appropriate pattern is loaded into the data direction registers:

```
LDA #$FF
STA DDR1A
STA DDR1B
STA DDR3B
```

All LEDs on the board are turned off:

```
LDA #0
STA PORT1A
STA PORT1B
```

Next, the interrupt vector's address must be loaded with a new pointer. The address to be deposited there is the address of the interrupt handler, which has been designed to provide the regular blinking of the LEDs. (This process has already been explained in previous chapters.) The interrupt handler resides at address INTVEC. The high byte and the low byte of this address will be loaded in memory locations IRQVH and IRQVL, respectively. A special assembler symbol is used to denote the low byte of the interrupt vector: #$<INTVEC. Conversely, the high byte is represented in assembly language by #$> INTVEC. The new interrupt vector is loaded at the specified memory locations:

```
JSR ACCESS
LDA #$<INTVEC
STA IRQVL
LDA #$>INTVEC
STA IRQVH
```

As usual, the interrupt-enable register must first be cleared, then the appropriate interrupt must be enabled:

```
LDA #$7F
STA IER
LDA #$C0
STA IER
```

Timer 1 is set to the free-running mode:

```
LDA #$40
STA ACR
```

The latch for timer 1 is loaded with the highest possible count, “FFFF”:

```
LDA #$FF
STA T1LL
STA T1CH
```

Finally, interrupts are enabled, the decimal mode is cleared as a precaution, and we terminate the initialization stage:

```
CLI
CLD
RTS
```

**Back to the Main Program**

We are now at line 69 of the program listing. We read the next key closure on the keyboard:

```
JSR GETKEY
```

It is the first move. We must determine whether it is an “F” or not. If it is an “F,” the player moves first; otherwise the computer moves first. Let us check it:

```
CMP #$F
BNE PLAYLP
```

It is the player's turn and this information is stored in the temporary variable PLAYR, shown in Figure 11.44:

```
LDA #01
STA PLAYR
```

It is time for a new move, and the move counter is incremented by one. Variable MOVNUM is stored in low memory. This is shown in Figure 11.44. It is now incremented:

```
PLAYLP INC MOVNUM
```
At this point, PLAYR indicates whose turn it is to play. If it is set at "zero," it is the computer's turn. If it is set at "one," it is the player's turn. Let us check it:

```
LDA PLAYR
BEQ CMPMU
```

We will assume here that it is the player's turn. PLAYR is reset to "zero" so that the computer will make its move next:

```
DEC PLAYR
```

The player's move is received by the PLRMV subroutine which will be described below. Let us allow the player to play:

```
JSR PLRMV
```

The move made by the player is specified at this point by the contents of the X register. Since it was the player's move, the corresponding code on the board's representation should be "01," which will be deposited in the accumulator:

```
LDA #01
```

We will now display the move on the board by blinking the proper LED. In addition, the corresponding ROWSUM will automatically be updated:

```
JSR UPDATE
```

The UPDATE routine will be described in detail below. Once the move has been made, we should check for a possible win. In the case of a win, the player has three blinking LEDs in a row, and the corresponding row total is automatically equal to "three." We will therefore simply check all eight rows for a ROWSUM of three:

```
LDA #03
BNE WINTST
```

At address WINTST a test is performed for a winning configuration. Index register Y is loaded with "seven" and used as a loop counter. All of the rows, 7 through 0, are checked for the value "three":

```
WINTST
TSTLP
```

```
LDY #7
CMP ROWSUM,4
BEQ WIN
DEY
WIN
BPL TSTLP
```

Let us now continue with the player's move. We will examine the computer's move later. (The computer's move corresponds to lines 83-88 of the program listing, which have not been described yet.) A maximum of nine moves is possible in this game. Let us verify whether or not we have reached the end of the game by checking the value of MOVNUM, which contains the number of the current move:

```
LDA MOVNUM
CMP #9
BNE PLAYLP
```

This is the end of our main loop. At this point, a branch occurs back to location PLAYLP, and execution of the main program resumes. If we had reached the end of the game at this point, the game would be a tie, since there has not been a winner yet. At this point all of the lights on the board would be set blinking and then the game would restart. Let us set the lights blinking:

```
LDA #$FF
STA LTMSKL
STA LTMSKH
BNE DLY
```

The delay is introduced to guarantee that the lights will be blinked for a short interval. Let us now examine the end-of-game sequence.

When a win situation is found, it is either the player's win or the computer's win. When the player wins, the row total is equal to "three." When the computer wins, the row total is equal to "twelve." (Recall that each computer move results in a value of "four" for the square. Three squares in a row will result in $3 \times 4 = 12$.) If the computer won, its IQ will be decremented:

```
```

LDA MOVNUM
CMP #9
BNE PLAYLP
```

This is the end of our main loop. At this point, a branch occurs back to location PLAYLP, and execution of the main program resumes.
At this point a jump would occur to INTDN, where the intelligence level will be decreased (intelligence lowered).

A losing tone will be generated to indicate to the player that he or she has lost. The corresponding frequency constant is “FF,” and it is stored at address FREQ:

```
INTDN   LDA #$FF
STA FREQ
```

The intelligence level will now be decreased unless it has already reached “zero” in which case it will remain at that value:

```
LDA INTEL
BEQ GTMSK
DEC INTEL
```

For a brief time the winning row will be illuminated on the board, and the end-of-game tone will be played. First, we clear all LEDs on the board:

```
GTMSK   LDA #0
STA PORT1A
STA PORT1B
```

At this point, the number of the winning row is contained in index register Y. The three squares corresponding to that row will simply be retrieved from the RWPT table. (See Figure 11.43.) Let us display the first square:

```
LDX RWPT1,Y
JSR LEDLTR
```

The LEDLTR routine will be described below. It lights up the square whose number is contained in register X. Let us now display the next square:

```
LDX RWPT2,Y
JSR LEDLTR
```

Then, the third one:

```
LDX RWPT3,Y
JSR LEDLTR
```

At this point, we should turn off all unnecessary blinking LEDs on the board. The new pattern to be blinked is the one with the winning row and we must, therefore, change the LTMSKL mask:

```
LDA PORT1A
AND LTMSKL
STA LTMSKL
```

We now do the same for Port 1B:

```
LDA PORT1B
AND LTMSKH
STA LTMSKH
```

Exercise 11-4: Subroutine LEDLTR on line 125 of the program listing has just lit the third LED on the board for the winning row. Immediately after that, we start reading the contents of Port 1A, and then Port 1B.

There is, however, the theoretical possibility that an interrupt might occur immediately after LEDLTR, that might change the contents of Port 1A. Would this be a problem? If it would not be a problem, why not? If it would, modify the program to make it always work correctly.

At this point, Ports A and B contain the appropriate pattern to light the winning row. If the player has won, the blink masks LTMSKL and LTMSKH contain the same pattern, and will blink the row. We are now ready to sound the win or lose tone. The duration is set at “FF”:

```
LDA #$FF
STA DUR
```

The frequency, FREQ, was set above. We simply have to play it:

```
LDA FREQ
JSR TONE
```
A delay must be provided:

\[ \text{DLY} \quad \text{JSR DELAY} \]

We are now ready to start a new game with the new intelligence level of the computer:

\[ \text{JMP RESTART} \]

**Back to WIN**

Let us now go back to line 103 of the program listing and examine the case in which the computer did not win (i.e., the player won). A different frequency constant is loaded at location FREQ:

\[ \text{LDA} \quad \#30 \quad \text{STA} \quad \text{FREQ} \]

Since the player won, the intelligence level of the computer will be raised this time. Before it is raised, however, it must be checked against the value "fifteen," which is our legal maximum:

\[ \text{LDA INTEL} \quad \text{CMP} \quad \#OF \quad \text{BEQ} \quad \text{GTMSK} \quad \text{INC} \quad \text{INTEL} \]

The sequence was exactly analogous to the one in which the computer wins, except for a different tone frequency, and for the fact that the intelligence level of the computer is increased rather than decreased.

**The Computer Moves**

Let us now go back to line 83 of the program listing and describe what happens when the computer makes a move. Variable PLAYR is incremented, then a delay is provided to simulate "thinking time" for the computer:

\[ \text{COMPMV} \quad \text{INC} \quad \text{PLAYR} \quad \text{JSR} \quad \text{DELAY} \]

The computer move is determined by the ANALYZ routine described below:

\[ \text{JSR ANALYZ} \]

The computer's move is entered as a "four" at the appropriate location on the board:

\[ \text{LDA} \quad \#04 \quad \text{JSR UPDATE} \]

Next, we check all of the rows for the possibility of a computer win, i.e., for a total of "twelve":

\[ \text{LDA} \quad \#12 \quad \text{WINTST} \quad \text{LDY} \quad \#7 \]

and so on. We are now back in the main program described previously.

When the program segment outlined above is compared to the one that is used for the player's move, we find that the primary difference between the two is that the move was specified by the ANALYZ routine rather than being picked up from the keyboard. This routine is the key to the level of intelligence of the algorithm. Let us now examine it.

**Subroutines**

**The ANALYZ Subroutine**

The ANALYZ subroutine begins at line 143 of the program listing. The corresponding conceptual flowchart is shown in Figure 11.40. In the ANALYZ subroutine the ODDMSK is first set to "zero."

\[ \text{ANALYZ} \quad \text{LDA} \quad \#0 \quad \text{STA} \quad \text{ODDMSK} \]

We now check for the possibility of a computer win during its next turn. If that possibility exists, we clearly must play into the winning square. This will end the game. A winning situation is characterized by a total of "eight" in the corresponding row; therefore let us deposit the total "eight" into the accumulator:
A winning situation will occur when the squares in rows 1, 2, or 3 all total "three" at the same time. Let us set our filter variable, X, for the number of rows that qualify, to "three":

LDX #03

We are now ready to use the FINDMV routine:

JSR FINDMV

The FINDMV routine will be described below. It must be called with the specified ROWSUM in A and with the number of times a match is found in X. It will systematically check all of the rows and squares. If a square is found, it exits with a specified square number in X and the Z flag is set to "0." Let us test it:

BNE DONE

If a winning move has been found, the ANALYZ routine exits. Unfortunately, this is not usually the case, and more analysis must be done.

The next special situation to be checked is to see if the player has a winning move. If so, it must be blocked. A winning situation for the player is indicated by a row total of "2." Let us load "2" into the accumulator and repeat the previous process:

LDA #02
LDA #03
JSR FINDMV
BNE DONE

If the player could make a winning move, this is the square where the computer should play and we exit to DONE; otherwise, the situation should be analyzed further.

We will now check to see if the computer can implement a trap. A trap corresponds to a situation in which a computer move has already been made in the same row. We would like to play at the intersection of two rows containing computer moves. This was explained above when the algorithm was described. This situation is characterized by $A = 4$ and $X = 2$. Let us load the registers with the appropriate values and call the FINDMV routine:

LDA #04
LDX #02
JSR FINDMV
BNE DONE

If we succeed, we exit to DONE; otherwise, we proceed down the flowchart diagrammed in Figure 11.40.

It is at this point that the computer can demonstrate either intelligent or ill-advised play. The behavior of the computer will be determined by its intelligence level. We will now obtain a random number and compare it to the computer's IQ. If the random number exceeds the computer's IQ, we will proceed to the left side of the flowchart in Figure 11.40 and make an ill-advised move (i.e., a random one). If the random number does not exceed the computer's IQ, we will make an intelligent move on the right side of the flowchart. Let us generate the random number:

JSR RANDOM

We truncate the random number to its right byte so that it does not exceed fifteen:

AND #$0F

and we compare it to the current IQ of the computer:

CMP INTEL
BEQ OK
BCS RNDMV

If the random number is higher than the IQ level stored in INTEL, we branch to RNDMV and play a random move. At this point, we will assume that the random number was not greater than the IQ level, and that the computer will play an intelligent move. We now proceed from line 162 (location "OK").

We will first check to see if this is move #1; then we check to see if this is move #4. Let us check for move #1:

OK

and call the FINDMV routine:

LDA #04
LDX #02
JSR FINDMV
BNE DONE

If we succeed, we exit to DONE; otherwise, we proceed down the flowchart diagrammed in Figure 11.40.

It is at this point that the computer can demonstrate either intelligent or ill-advised play. The behavior of the computer will be determined by its intelligence level. We will now obtain a random number and compare it to the computer's IQ. If the random number exceeds the computer's IQ, we will proceed to the left side of the flowchart in Figure 11.40 and make an ill-advised move (i.e., a random one). If the random number does not exceed the computer's IQ, we will make an intelligent move on the right side of the flowchart. Let us generate the random number:

JSR RANDOM

We truncate the random number to its right byte so that it does not exceed fifteen:

AND #$0F

and we compare it to the current IQ of the computer:

CMP INTEL
BEQ OK
BCS RNDMV

If the random number is higher than the IQ level stored in INTEL, we branch to RNDMV and play a random move. At this point, we will assume that the random number was not greater than the IQ level, and that the computer will play an intelligent move. We now proceed from line 162 (location "OK").

We will first check to see if this is move #1; then we check to see if this is move #4. Let us check for move #1:

OK

and call the FINDMV routine:

LDA #04
LDX #02
JSR FINDMV
BNE DONE

If we succeed, we exit to DONE; otherwise, we proceed down the flowchart diagrammed in Figure 11.40.

It is at this point that the computer can demonstrate either intelligent or ill-advised play. The behavior of the computer will be determined by its intelligence level. We will now obtain a random number and compare it to the computer's IQ. If the random number exceeds the computer's IQ, we will proceed to the left side of the flowchart in Figure 11.40 and make an ill-advised move (i.e., a random one). If the random number does not exceed the computer's IQ, we will make an intelligent move on the right side of the flowchart. Let us generate the random number:

JSR RANDOM

We truncate the random number to its right byte so that it does not exceed fifteen:

AND #$0F

and we compare it to the current IQ of the computer:

CMP INTEL
BEQ OK
BCS RNDMV

If the random number is higher than the IQ level stored in INTEL, we branch to RNDMV and play a random move. At this point, we will assume that the random number was not greater than the IQ level, and that the computer will play an intelligent move. We now proceed from line 162 (location "OK").

We will first check to see if this is move #1; then we check to see if this is move #4. Let us check for move #1:

OK

and call the FINDMV routine:

LDA #04
LDX #02
JSR FINDMV
BNE DONE

If we succeed, we exit to DONE; otherwise, we proceed down the flowchart diagrammed in Figure 11.40.

It is at this point that the computer can demonstrate either intelligent or ill-advised play. The behavior of the computer will be determined by its intelligence level. We will now obtain a random number and compare it to the computer's IQ. If the random number exceeds the computer's IQ, we will proceed to the left side of the flowchart in Figure 11.40 and make an ill-advised move (i.e., a random one). If the random number does not exceed the computer's IQ, we will make an intelligent move on the right side of the flowchart. Let us generate the random number:

JSR RANDOM

We truncate the random number to its right byte so that it does not exceed fifteen:

AND #$0F

and we compare it to the current IQ of the computer:

CMP INTEL
BEQ OK
BCS RNDMV

If the random number is higher than the IQ level stored in INTEL, we branch to RNDMV and play a random move. At this point, we will assume that the random number was not greater than the IQ level, and that the computer will play an intelligent move. We now proceed from line 162 (location "OK").

We will first check to see if this is move #1; then we check to see if this is move #4. Let us check for move #1:

OK
If it is move #1, we occupy any square:

BEQ RNDMV

Let us now check for move #4:

CPX #4

If it is not move #4, we will check to see if the player can set a trap. This will be performed at location TRAPCK. Let us assume here that it is move #4.

BNE TRAPCK

This section will check both diagonals for the possibility of the sequence player-computer-player. If this sequence is found, we will play to the side. Otherwise, we will go back to the mainstream of this routine and check to see if the player can set a trap. The combination player-computer-player in a row is detected when the row totals "six." Therefore, we load the value "six" into the accumulator and check the corresponding diagonal. By coincidence, diagonals correspond to the sixth and seventh entries in our RWPT table. (See Figure 11.46.) Let us do it:

LDX #6
TXA
CMP ROWSUM,X
REQ ODDRND

If a match is found, we branch to address ODDRND, where we will play to the side. This will be described below. If a match is not found we check the next diagonal:

INX
CMP ROWSUM,X
BEQ ODDRND

If, at that point, the test also fails for the second diagonal, we will check to see if the player can set a trap.

---

Checking To See If the Player Can Set a Trap (TRAPCK)

The possibility of a trap for the player is identified (as in the case of the computer), when two intersecting rows each contain only a player's move. This has been explained in the description of the algorithm above. The value of a row which is a candidate for a trap is thereby equal to "one" (one player's move). The parameters must, therefore, be set to $A = 1$, and $X = 2$ before we can call the FINDMV routine:

TRAPCK    LDA #1
          LDX #2
          JSR FINDMV
          BNE DONE

If the proper location for a trap can be found, the next move is to play there. Otherwise, if possible, the computer moves to the center or, if the center is occupied, it makes a random move on the side.

LDX GMBRD + 4
BNE RNDMV
LDX #5
BNE DONE

Playing a Random Move on the Side

The four sides on the board are numbered externally 2,4,6 and 8, or internally 1,3,5, and 7. Any odd internal number specified for a move will result in our occupying a side position. If we want to occupy a side position, we simply load the value "one" in ODDMSK, and we guarantee that the random number generated will be one of the four corners. This is performed by entering at address ODDRND:

ODDRND    LDA #1
          STA ODDMSK

Generally, however, we may want to make a random move. This will be accomplished by generating and using any random number that is reasonable, i.e., by setting ODDMSK to "0" prior to entering at address RNDMV. Let us obtain a random number:
RNDMV JSR RANDOM

Let us strip off the left byte:

AND #$0F

Then let us OR this random number with the pattern stored in ODDMSK. If the mask had been set to "0," it would have no effect on the random number. If the mask had been set to "1," however, it would result in our playing into one of the corners (the center is occupied here):

ORA ODDMSK

Since the random number which was generated was between "0" and "15," we must check to be sure that it does not exceed "9"; otherwise, it cannot be used:

CMP #9
BCS RNDMV

We must now check to make sure that the space into which we want to move is not occupied. We load the square's number into index register X and verify the square's status by reading the appropriate entry of the GMBRD table (see the memory map in Figure 11.47):

TAX
LDA GMBRD,X

If there is any entry other than "0" in this square, it means that it is occupied and we must generate another random number:

BNE RNDMV

We have selected a valid square and will now play into it. When we exit from this routine, the external LED number should be contained in X. It is obtained by adding "1" to the current contents of X, which happens to be the internal LED number:

INX
DONE RTS

FINDMV Subroutine

This subroutine will evaluate the board until it finds a square which meets the specifications in the A and the X registers. The accumulator A contains a specified row-sum that a row must meet in order to qualify. Index register X specifies the number of times that a particular square must belong to a row whose row-sum is equal to the one specified by A.

The FINDMV subroutine starts with a square status of "0" for every square on the board. Every time it finds a square that meets the row-sum specification, it will increase its status by "1." Thus, at the end of the evaluation process, a square with a status of "1" is a square which meets the row-sum specifications once. A square with a status of "2" is one that meets the specification twice, etc.

The final selection is performed by FINDMV, which checks the value of each square in turn. As soon as it finds a square whose status matches the number contained in register X, it selects that square as one that meets the initial specification.

The complete flowchart for FINDMV is shown in Figure 11.49. Essentially, the subroutine operates in three steps. These steps are indicated in Figure 11.49. Step 1 is the initialization phase. Step 2 corresponds to the selection of all squares that meet the row-sum specifications contained in register A. The status of every empty square in a row that meets this specification is increased by one as all the rows are scanned. Step 3 is the final selection phase. In this phase, each square is looked at in turn until one is found whose status matches the value contained in X. As soon as one is found, the process stops. That square is the one that will be played by the computer. If a square is not found, the routine will exit, with the index X having decremented to "0," and this will be used as a failure flag for the calling routine.

Let us now examine the corresponding program. It starts at line 204 in the program listing.

Step 1: Initialization

Index registers X and A will be used in the body of this subroutine. Their initial contents must first be preserved in temporary memory locations. Addresses TEMP1 and TEMP2 are used for that purpose. (See Figure 11.47 for the memory map.) Let us preserve X and A:
The status of the board is then cleared. Each square's status must be set to "0." This is accomplished by loading the value "0" into the accumulator, then going through a nine cycle loop that will clear the status of each square in turn:

```
LDA #0
LDY #8
CLRLP STA SQSTAT,4
DEY
BPL CLRLP
```

**Step 2: Computing the Status of Each Square**

Each of the eight possible row-sums will now be examined in turn. If the row-sum matches the value specified in the accumulator on entry, each empty square within the specified row will have its status incremented by "1." If the row-sum value does not meet the minimum, the next one will be examined. Index register Y is used as a row pointer. The RWPT table described at the beginning of this program and shown in Figure 11.46 will be used to successively retrieve the three squares that form every row. Let us first initialize our counter:

```
LDY #7
```

Now, we will check the value of the corresponding row-sum:

```
CHEKLP LDA TEMP1
CMP ROWSUM,Y
BNE NOCHEK
```

Let us assume at this point that the row-sum is indeed the correct one. We must now examine each of the three squares in the row. If the square is empty, we increment its status. The first step is to obtain the square's value by looking it up in the table, using index register Y as a displacement, and using addresses RWPT1, RWPT2, and RWPT3 successively as entry points into the row table. Let us try it for the first square:
LDX RWPT1,Y

Index register X now contains the square number. If the square is empty, a new subroutine, CNTSUB, is used to increment its status:

JSR CNTSUB

It will be described below.

Let us now do the same for the second and third squares:

LDX RWPT2,Y
JSR CNTSUB
LDX RWPT3,Y
JSR CNTSUB

We have now completely scanned one row. Let us look to see if any more rows need to be checked:

NOCHEK DEY BPL CHECKLP

The process is repeated until all the rows have been checked. At this point, we enter into step 3 of FINDMV. (Refer to the flowchart in Figure 11.49.)

Step 3: Final Selection

Index register X will be used as a square pointer. It will start with square #9 and continue to examine squares until one is found that meets the additional X register specifications, i.e., the number of times that the given square belongs to a row with the appropriate row-sum value. Let us initialize it:

LDX#9

Now, we compare the value of the square status with the value of the specified X parameter:

FNMTCH LDA TEMP2
AND SQSTAT-1,X

If the square status matches the value of the parameter, we select this square:

BNE FOUND

Otherwise, we try the next one:

DEX BNE FNMTCH
FOUND RTS

Exercise 11-5: Why are "AND" and "BNE" rather than "CMP" and "BEQ" used to find a matching square above? (Hint: decide what the difference in the program's strategy would be.)

COUNTSUB Subroutine

This subroutine is used exclusively by the FINDMV subroutine and increments the status of the square whose number is in register X, if the square is empty. First, it examines the status of the square by looking for its code in the GMBRD table:

CNTSUB LDA GMBRD,X
BNE NOCNT

If the square is occupied, an exit occurs. If it is not, the status value of the square is incremented:

INC SQSTAT,X
NOCNT RTS

UPDATE Subroutine

Every time a move is made, it must be displayed on the board. Then, the appropriate code must be stored in the board representation, i.e., in the table GMBRD. Finally, the new ROWSUMs must be computed and stored at the appropriate locations. These functions are accomplished by the UPDATE subroutine.

The player's code is contained in the accumulator. The position into which the move is made is contained in register X. Since the number in index register X is the value of an external LED, it is first decremented in order to match the actual internal LED number:
The value must now be stored in the appropriate location of the GMBRD

Table which contains the internal representation of the board:

\[
\text{STA GMBRD, X}
\]

Note that the value of X is simply used as a displacement into the

table. However, the accumulator happens to contain the appropriate

code that is merely written at the specified location. At this point, UPDATE

would like to display the move on the LEDs. It must first
decide, however, whether to light a steady LED or make it blink. To
do this, it must determine whether it is the player's move or the com-
puter's move. It does this by examining the code contained in the ac-
cumulator. If the code is "four," it is the computer's move. If the
code is "1," it is the player's move. Let us examine it:

\[
\begin{align*}
\text{CMP } & \#04 \\
\text{BEQ } & \text{NOBLNK}
\end{align*}
\]

If it is the computer's move, a branch will occur to address NOBLNK;
otherwise, we proceed. Let us assume for the time being that it was the
player's move:

\[
\text{JSR LIGHT}
\]

The LIGHT subroutine is used to set the bit blinking and will be
described below. Upon exit from LIGHT, the accumulator contains
the bit in the position that is required to set the LED blinking. At this
point, the blink masks should be updated:

\[
\begin{align*}
\text{ORA LTMSKL} \\
\text{STA LTMSKL}
\end{align*}
\]

If the carry was "zero" upon completion of LIGHT, one of the bits
zero through seven had been set and we are done:

\[
\text{BCC NOBLNK}
\]

Otherwise, if the carry had been set to 1, it would mean that LED #9
had to be set, i.e., that the high order part of the mask had to be

modified. Let us do it:

\[
\begin{align*}
\text{LDA } & \#01 \\
\text{STA LTMSKH}
\end{align*}
\]

At this point, the LED masks are properly configured and we can give
the order to light the LEDs:

\[
\begin{align*}
\text{NOBLNK} & \quad \text{JSR LEDLTR}
\end{align*}
\]

The LEDLTR routine lights up the LED specified by register X. Note
that if it was a computer move, this LED will remain steadily on. If it
was a player's move, this LED will be turned off and on automatically
as interrupts occur.

Next, we must update all row-sums. Index register X is used as a
row pointer. We will look at all eight rows in turn. In anticipation of
the addition, the carry bit is cleared:

\[
\begin{align*}
\text{LDA } & \#01 \\
\text{STA LTMSKH}
\end{align*}
\]

The first square of row eight is examined first:

\[
\begin{align*}
\text{LDY} & \text{ RWPT1,X} \\
\end{align*}
\]

Note that index register Y will contain the internal square number
following this instruction. This will immediately be used for another
indexed operation. The contents of the square will be read so that the
new row-sum may be computed. (The row-sum for that row may or
may not be the same as before. No special provision has been made
for restricting the search to the two or three rows affected.) All rows
are examined in turn, and all row-sums are re-computed to keep the
program simple.

Let us obtain the current square's value:

\[
\begin{align*}
\text{LDA GMBRD,Y}
\end{align*}
\]

The GMBRD table is accessed using index register Y as a displace-
ment. Note that the two instructions shown above implement a two-
level indexing operation. This is a most efficient data retrieval tech-
nique. At this point, the accumulator contains the value of the first
square. It will be added to the value of the two following squares. The process will now be repeated:

```
LDY RWPT2,X
ADC GMBRD,Y
```

The number of the second square has been looked up by the LDY instruction and its value stored in Y. The addition instruction looks up the actual value of that square from GMBRD, and adds that value to the accumulator. This process is performed one more time for the third square:

```
LDY RWPT3,X
ADC GMBRD,Y
```

The final value contained in the accumulator is then stored in the ROWSUM table at the position specified by the value of index register X (the row index):

```
STA ROWSUM,X
```

The next row will now be scanned:

```
DEX
BPL ADDROW
```

If X becomes negative, we are done:

```
RTS
```

**LED LIGHTER Subroutine**

This subroutine assumes upon entry that register X contains the internal LED number of the LED on the board which must be turned on. The subroutine will therefore turn that LED on using the LIGHT subroutine, which converts a number in register X into a bit pattern in the accumulator for the purpose of turning on the specified LED:

```
LEDLTR JSR LIGHT
```

At this point, either Port 1A or Port 1B must be updated. Let us assume initially that it is Port 1A (if it is not Port 1A, which we can find out by examining the carry bit below, then the pattern contained in the accumulator is all zeroes and will not change the value of Port 1A):

```
ORA PORT1A
STA PORT1A
BCC LTRDN
```

The carry bit is tested. If it has been set to 1 by the LIGHT subroutine, then LED #9 must be turned on. This is accomplished by sending a "1" to Port 1B:

```
LDA #1
STA PORTB
RTS
```

**PLRMV Subroutine (Player’s Move)**

This subroutine obtains one correct move from the player. It chirps to get his or her attention and waits for a keyboard input. If a key other than 1 through 9 is pressed, it will be ignored. Whenever the subroutine gets a move, it verifies that the square on the board is indeed empty. If the square is not empty, the subroutine will ignore the player’s move. Let us first generate a chirp in order to get the player’s attention:

```
PLRMV LDA #$80
STA DUR
LDA #$10
JSR TONE
```

Now, let us capture the key closure:

```
KEYIN JSR GETKEY
```

We must now check to see that the key that is pressed is between 1 and 9. Let us first check to see that it is not greater than or equal to 10:

```
CMP#10 BCS KEYIN
```

Let us now verify that it is not equal to "zero":

```
CMP #10 BCS KEYIN
```
TAX
BEQ KEYIN

Finally, let us verify that it does not correspond to a square that is already occupied:

LDA GMBRD-1,X
BNE KEYIN
RTS

**Exercise 11-6:** Modify the PLRMV subroutine above so that a new chirp is generated every time a player makes an incorrect move. To tell the player that he or she has made an incorrect move, you should generate a sequence of two chirps, using a different tone than the one used previously.

**LIGHT Subroutine**

This subroutine accepts an LED number in register X. It returns with the pattern to be output to the LEDs in the accumulator. If LED 9 is to be lit (X = 8), the carry bit is set. This subroutine is straightforward and has been described previously:

```
LIGHT  STX TEMP1
      SEC
      ROL A
      DEX
      BPL SHIFT
      LDX TEMP1
      RTS
```

**DELAY Subroutine**

This is a classic delay subroutine that uses two nested loops that have a few extra instructions within the loop that are designed to waste time:

```
DELAY  LDY #$FF
      LDX #$FF
      ROL DUR
      ROR DUR
      DEX
      BNE DL2
      DEY
      BNE DL1
      RTS
```

**Interrupt Handling Routine**

Every time that an interrupt is received, the appropriate LEDs will be complemented (turned off if on, or on if off). The positions of the LEDs to be blinked are specified by the contents of the LTMSK masks. Two bytes are used in memory for the low and high halves, respectively. (See Figure 11.47 for the memory map.)

Turning the bits on or off is accomplished by an exclusive-OR instruction that is the equivalent of a logical complementation. Since this routine uses the accumulator, the contents of A must be preserved at the beginning of the routine. It is pushed onto the stack and restored upon exit. The subroutine is shown below:

```
INTVEC PHA
      LDA PORT1A
      EOR LTMSKL
      STA PORT1A
      LDA PORT1B
      EOR LTMSKH
      STA PORT1B
      LDA TILL
      PLA
      RTI
```

**Exercise 11-7:** Notice the LDA TILL instruction above. The next instruction in this subroutine is PLA. It will overwrite the contents of the accumulator with the words pulled from the stack. The contents of the accumulator, as read from TILL, will therefore be immediately destroyed. Is this a programming error that was accidentally left in this program? If not, what purpose does it serve? (Hint: this situation has been encountered before. Refer to one of the earlier chapters.)

**INITIALIZE Subroutine**

This subroutine was described in the body of the main program above.
RANDOM and TONE Subroutines

These two subroutines were described in previous programs.

SUMMARY

This program was the most complex we have developed. Several algorithms have been presented, and one complete implementation of an ad hoc algorithm has been studied in great detail. Readers interested in games of strategy and programming are encouraged to implement an alternative algorithm.

```
LINE  LOC  CODE  LINE
0002  0000  0010  0011  0000  0005  0000  0004  0000
0009  0002  0010  0000  0015  0000  0009  0000
0012  0000  0010  0000  0017  0000  0018  0000
0026  0000  0027  0000  0032  0000  0035  0003  00
0035  0001  01
0035  0000  00
0035  0007  02
0035  0006  00
0035  0004  03
0035  0002  02
0036  0000  04
0036  0000  07
0036  0004  04
0037  0010  06

---

Fig. 11.50: Tic-Tac-Toe Program

---

Fig. 11.50: Tic-Tac-Toe Program (Continued)
```
0103 0251 A9 1E LDA #50 ILDA FREQ. CONST FOR WIN TIME.
0104 0253 B5 3F STA FREQ.
0105 0255 A5 41 LDA INTEL.
0106 0257 C9 0F CMP #00.If J.G. AS HIGH AS POSSIBLE?
0107 0259 A5 0E CMP #01. INTEG. DON'T CHANGE IT.
0108 025B E6 41 INC INTEL. 
0109 025D A5 0E CMP #00. 100 FLASH BOX.
0110 025F AF FF INTD IN LDA #5F ILDA FREQ. CONST. FOR LOSS TIME.
0111 0261 B5 3F STA FREQ.
0112 0263 A5 41 LDA INTEL.
0113 0265 F0 02 BEO STMSK. IF YES DON'T DECREMENT!
0114 0267 A4 41 DEC INTEL.
0115 0269 A9 00 OTHSTK LDA #0 CLEAR ALL LPRS.
0116 026B B0 01 A0 STA FORIN.
0117 026D B0 00 STA FORIN.
0118 0271 B6 00 LEX RXPTY IF HIT BIT IN ACCUM. TO LIGHT
0119 0273 F0 03 LER CONJ Programm TO 1ST SQUARE
0120 0275 20 06 JSX WINNING RUN.
0121 0277 20 4F 03 JSR LEITTH.
0122 0279 B4 00 LEX RXPTY IF HEADING NEXT BIT.
0123 027B 20 4F 03 JSR LEITTH.
0124 027D 2A 10 LEX RXPTY IF HEADING 3RD BIT.
0125 027F 20 A0 0A JSR LEITTH.
0126 0280 A1 01 LDA PORTIA IMAG. OUT UNNECESSARY BITS IN
0127 0281 25 3D AND LMSKH. IMAG. BIT IN MASK.
0128 0285 B5 3D STA LMSKH.
0129 0287 A0 0A LDA FORTIN.
0130 028A 25 3C AND LMSKH.
0131 028C B5 3C STA LMSKH.
0132 028E A9 FF LDA #00 IF HIT WIN/LOSS TIME DURATION.
0133 0290 B9 3F STA DUR.
0134 0292 A5 3F LDA FREQ.
0135 0294 20 AE 00 JSR TONE IF FREQUENCY.
0136 0296 20 A4 03 DLY JSR DELAY IF DELAY TO SHOW WIN OR TIE.
0137 0298 20 AC 04 JSR RESTK IF INSTANT GAME, DON'T CHNG. I.D.
0138 029A 02 00

I "**** SUBROUTINE 'ANALYZE' ******
0139 029C 20 00 HIDES A STATIC ANALYSIS OF GAME BOARD, AND
0140 029E 20 00 RETURNS WITH A MOVE INDEX IN X.
0142 029F 20 00

0143 029B A9 00 ANALYZ LDA #70 SET MASK THAT MAKES RANDOM MOVE.
0144 029D BF 40 STA PORTIA A BIT SIDES TO 0.
0145 029F A5 09 STA LDA #00 CHEK RXPTY MOVING MOVE.
0146 02A1 A5 03 LEX #03 COMPUTER.
0147 02A3 20 04 03 JSR FINDM.
0148 02A5 B0 59 BNE DONE IF FOUND. RETURN.
0149 02A7 A9 03 LDA #03 CHECK FOR WINNING MOVE FOR
0150 02A9 A2 03 LEX #03 IPLAYER.
0151 02AC A5 03 JSR FINDM.
0152 02B1 B0 50 BNE DONE IF FOUND. RETURN.
0153 02B3 A8 03 LDA #03 COMPUTER.
0154 02B5 A0 03 JSR RESTK IF CAN COMPUTE A TRAP?
0155 02B7 20 A2 03 LEX #03 CX.
0156 02B9 A5 03 JSR FINDM.
0157 02BA 20 0A 00 BNE DONE IF YES PLAY IT.
0158 02BB 29 0F AND #00 IF NOT K.O. AND MAKE 0-15.
0159 02BD A5 0E CMP #00 IF USE AS RPT/RND DETRITTER.
0160 02BF F0 02 BEO ORM IF BOTH ARE FORM #UWRK TST.
0161 02C1 D8 09 BEG ORM.
0162 02C3 A5 03 BCS KMSKH IF HIND X INTEG. PLAY A RND MOVE.
0163 02C5 A6 3A OX LDK MOUNT.
0164 02C7 E0 01 CPX #1 LIST MOVE.
0165 02C9 FC 02 BEO ORM.
0166 02CB C4 04 CPX #4 AND MOVE.
0167 02CD F0 25 BEO ORM.
0168 02CE A5 41 BRA TRAP.
0169 02CF E6 41 CPU GRAY.
0170 02D1 A2 06 LDX #00 LOAD IND TO 1ST DIG. RXPTY.
0171 02D3 8A 02 BRA RXPTY.
0172 02D5 2A 00 LDA RXPTY #10 FLASH BOX.
0173 02D7 F0 1A BEO ORMONG IF YES, PLAY SIDE.
0174 02D9 F0 11 BEO ORMONG.

--- Fig. 11.50: Tic-Tac-Toe Program (Continued) ---
**Fig. 11.50: Tic-Tac-Toe Program (Continued)**

- **0247 0343 CP 04** CMP #04 COMPUTER'S MOVE
- **0248 0345 FF 05** BNE BLK2
- **0249 0347 00 08** JSR LIGHT IF YES, DON'T SET LED BLINKING.
- **0250 0346 00 09** JSR LIGHT SET LED BLINKING.
- **0251 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0252 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0253 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0254 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0255 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0256 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0257 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0258 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0259 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0260 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.

**Additional Code:**

- **0247 0343 CP 04** CMP #04 COMPUTER'S MOVE
- **0248 0345 FF 05** BNE BLK2
- **0249 0347 00 08** JSR LIGHT IF YES, DON'T SET LED BLINKING.
- **0250 0346 00 09** JSR LIGHT SET LED BLINKING.
- **0251 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0252 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0253 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0254 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0255 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0256 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0257 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.
- **0258 0345 00 09** JSR LIGHT PLACE TOP BLINK MASKS.
- **0259 0345 00 09** JSR LIGHT PLACE BTM BLINK MASKS.

**Note:** The above code appears to be a continuation of the Tic-Tac-Toe program, showing various bitwise operations and conditional jumps to control the game's logic, particularly focusing on the computer's move and led blinking mechanisms.
APPENDIX A

6502 INSTRUCTIONS—ALPHABETIC

ADC  Add with carry
AND  Logical AND
ASL  Arithmetic shift left
BCC  Branch if carry clear
BCS  Branch if carry set
BEQ  Branch if result = 0
BIT  Test bit
BMI  Branch if minus
BNE  Branch if not equal to 0
BPL  Branch if plus
BRK  Break
BVC  Branch if overflow clear
BVS  Branch if overflow set
CLC  Clear carry
CLD  Clear decimal flag
CLI  Clear interrupt disable
CLV  Clear overflow
CMP  Compare to accumulator
CPX  Compare to X
CPY  Compare to Y
DEC  Decrement memory
DEX  Decrement X
DEY  Decrement Y
EOR  Exclusive OR
INC  Increment memory
INX  Increment X
INY  Increment Y
JMP  Jump
JSR  Jump to subroutine
LDA  Load accumulator
LDX  Load X
LDY  Load Y
LSR  Logical shift right
NOP  No operation
ORA  Logical OR
PHA  Push A
PHP  Push P status
PLA  Pull A
PLP  Pull P status
ROL  Rotate left
ROR  Rotate right
RTI  Return from interrupt
RTS  Return from subroutine
SBC  Subtract with carry
SEC  Set carry
SED  Set decimal
SEI  Set interrupt disable
STA  Store accumulator
STX  Store X
STY  Store Y
TXA  Transfer A to X
TAY  Transfer A to Y
TXS  Transfer SP to X
TXA  Transfer X to A
TXS  Transfer X to SP
### APPENDIX B

**6502—INSTRUCTION SET: HEX AND TIMING**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Instruction Set: Hex and Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>0A</td>
<td>add with carry, immediate</td>
</tr>
<tr>
<td>ADD</td>
<td>00</td>
<td>add, immediate</td>
</tr>
<tr>
<td>AND</td>
<td>28</td>
<td>and, absolute</td>
</tr>
<tr>
<td>BCC</td>
<td>08</td>
<td>branch on carry set, immediate</td>
</tr>
<tr>
<td>BCS</td>
<td>18</td>
<td>branch on carry set, immediate</td>
</tr>
<tr>
<td>BNE</td>
<td>1C</td>
<td>branch on negative, absolute</td>
</tr>
<tr>
<td>BMI</td>
<td>0C</td>
<td>branch on negative, immediate</td>
</tr>
<tr>
<td>BPL</td>
<td>2C</td>
<td>branch on positive, absolute</td>
</tr>
<tr>
<td>BPI</td>
<td>3C</td>
<td>branch on positive, immediate</td>
</tr>
<tr>
<td>CLD</td>
<td>0D</td>
<td>clear decimal register, absolute</td>
</tr>
<tr>
<td>CLI</td>
<td>1D</td>
<td>clear decimal register, immediate</td>
</tr>
<tr>
<td>CLV</td>
<td>2D</td>
<td>clear link register, absolute</td>
</tr>
<tr>
<td>CMP</td>
<td>06</td>
<td>compare, absolute</td>
</tr>
<tr>
<td>CPX</td>
<td>16</td>
<td>compare, absolute</td>
</tr>
<tr>
<td>CPH</td>
<td>26</td>
<td>compare, absolute</td>
</tr>
<tr>
<td>CPL</td>
<td>36</td>
<td>complement, absolute</td>
</tr>
<tr>
<td>DEC</td>
<td>4A</td>
<td>decrement, absolute</td>
</tr>
<tr>
<td>DEY</td>
<td>5A</td>
<td>decrement, absolute</td>
</tr>
<tr>
<td>EOR</td>
<td>6A</td>
<td>exclusive or, absolute</td>
</tr>
<tr>
<td>INC</td>
<td>7A</td>
<td>increment, absolute</td>
</tr>
<tr>
<td>INX</td>
<td>8A</td>
<td>index indirect</td>
</tr>
<tr>
<td>JMP</td>
<td>A2</td>
<td>jump, direct</td>
</tr>
<tr>
<td>JSR</td>
<td>AA</td>
<td>jump to subroutine, absolute</td>
</tr>
<tr>
<td>LDA</td>
<td>0A</td>
<td>load accumulator, absolute</td>
</tr>
<tr>
<td>LDX</td>
<td>2A</td>
<td>load indirect, absolute</td>
</tr>
<tr>
<td>LDY</td>
<td>4A</td>
<td>load indirect, absolute</td>
</tr>
<tr>
<td>NOP</td>
<td>00</td>
<td>no operation, absolute</td>
</tr>
<tr>
<td>PHA</td>
<td>8A</td>
<td>push accumulator, absolute</td>
</tr>
<tr>
<td>PHX</td>
<td>06</td>
<td>push indirect, absolute</td>
</tr>
<tr>
<td>PHP</td>
<td>26</td>
<td>push program counter, absolute</td>
</tr>
<tr>
<td>PLP</td>
<td>46</td>
<td>pull indirect, absolute</td>
</tr>
<tr>
<td>PLA</td>
<td>0A</td>
<td>pull accumulator, absolute</td>
</tr>
<tr>
<td>PLX</td>
<td>2A</td>
<td>pull indirect, absolute</td>
</tr>
<tr>
<td>POP</td>
<td>00</td>
<td>pop indirect, absolute</td>
</tr>
<tr>
<td>PHB</td>
<td>8A</td>
<td>push indirect, absolute</td>
</tr>
<tr>
<td>PLA</td>
<td>0A</td>
<td>pull accumulator, absolute</td>
</tr>
<tr>
<td>PLX</td>
<td>2A</td>
<td>pull indirect, absolute</td>
</tr>
<tr>
<td>STA</td>
<td>0A</td>
<td>store accumulator, absolute</td>
</tr>
<tr>
<td>STX</td>
<td>2A</td>
<td>store indirect, absolute</td>
</tr>
<tr>
<td>STY</td>
<td>4A</td>
<td>store indirect, absolute</td>
</tr>
<tr>
<td>TAX</td>
<td>AA</td>
<td>transfer accumulator, absolute</td>
</tr>
<tr>
<td>TSX</td>
<td>AA</td>
<td>transfer indirect, absolute</td>
</tr>
<tr>
<td>TXA</td>
<td>AA</td>
<td>transfer indirect, absolute</td>
</tr>
<tr>
<td>TSX</td>
<td>AA</td>
<td>transfer indirect, absolute</td>
</tr>
<tr>
<td>TXS</td>
<td>AA</td>
<td>transfer indirect, absolute</td>
</tr>
<tr>
<td>TXA</td>
<td>AA</td>
<td>transfer indirect, absolute</td>
</tr>
</tbody>
</table>

(1) Add 1 to n if crossing page boundary

(2) Add 2 to n if branch within page
Add 3 to n if branch to another page
INDEX

ACCESS, 170
Ad hoc algorithm, 239
Ad hoc programming, 238
Analytical algorithm, 225
ANALYZE, 263
Array, 122
Artificial intelligence, 224
Assembler, 47
Assembly, 12
Audio feedback, 163
Auxiliary Control Register, 174
BEO, 154
Binary number, 41
Blackjack, 189
Blackjack Program, 212
BLIN
Blink masks, 175
BLINKER, 208
Blinking, 274
Blinking LEDs, 261
Flip counter, 92
Board analysis flowchart, 242
Bounce, 13
Bracket-filtering, 150
Carry, 206
Cassette recorder, 4
CLI, 174
CNTSUB, 55
Complement, 73
Complementation Table, 80
Computing the Status, 271
Constant symbols, 47
Counter, 65, 101
COUTNSUB, 273
Current limiters, 11
Decimal mode, 151
Decision tables, 225
DELAY, 56, 132, 211, 278
Delay constant, 103
Diagonal trap, 244
Diagonals, 266
DISPLAY, 118
DISPLY, 119
Do-nothing, 55
Draw, 222
Dual Counter, 92
Duration, 148
DURTAB, 144
ECHO, 137
Echo, 35
Echo Program, 145
ESP Tester, 139
EVAL, 118, 126, 153
Evaluating the board, 225
Extra Sensory Perception, 139
FINDMV, 264, 269
FINDMV flowchart, 270
First move, 235
Free run, 198
Free-running, 198
Free-running mode, 171, 256
Frequencies, 25
Frequency, 22, 261
Frequency and duration constants, 161
Games Board, 2, 7
GETKEY, 13, 149
GETKEY Program, 17
GMBRD, 252
Heuristic strategy, 225
Hexagonal, 41
Hexguess Program, 63
IER, 171
IFR, 171
Illegal key closure, 95
Index, 159
Indexed addressing, 37, 39, 122, 126
Initialization, 198
INITIALIZE, 279
Intelligence level, 252, 260
Interrupt, 198, 252, 261
Interrupt Handler, 183, 211
Interrupt handling, 198, 279
Interrupt Registers, 174
Interrupt-enable register, 256
Interrupt-enable, 171, 179, 256
IQ level, 245, 265
Jackpot, 100
JMF, 154
Key closure, 277
Keyboard, 7
Keyboard input routine, 13
Labels, 47
Latch, 65
LED #9, 123
LED Connection, 10
LEDs, 8
Levels of difficulty, 8
LIGHT, 118, 132, 157, 274, 278
LIGHTER, 276
LIGHTTR, 207
LITE, 70, 182
Loop counter, 92
LOSE, 130
Magic Square, 73
MasterMind, 162
Middle C, 23
Mindbender, 162
Mindbender Program, 184
MOVE, 47
Multiplication, 122
Music Player, 20
Music Program, 31
Music theory, 23
Nested loop delay, 39
Nested loop design, 25
NOTAB, 144
Note duration, 159
Note frequency, 159
Note sequence, 139
Parameters, 149
Parts, 11
Perfect square, 73
PLAY, 48, 53
PLAYEM, 37
Playing to the side, 24
PLAYIT, 30, 38
PLAYNOTE, 30
PLRMV, 277
Potential, 225
Power supply, 4
Programmable bracket, 101
Prompt, 42
Protected, 170
Protected area, 170
Pulse, duration, 171
RANDB, 210
RANDOM, 57, 135, 150, 159, 209
Random moves, 241
Random number, 54, 65, 78, 118, 267
Random number generator, 57, 118, 149
Random pattern, 73
Random move, 267
Recursion, 211
Repeat, 13
Resistors, 11
RNDSCR, 252
Row sequences, 251
Row-sum, 239, 271
SBC, 206
Scratch area, 57
Score, 107, 128
Score table, 107, 111, 112
SCORTB, 127
Seed, 118, 149
74154, 8
7416, 8
Shifting loop, 158
SHOW, 152
Side, 267
Simple tones, 21
Siren, 100
Slot Machine, 99
Slot Machine Program, 113
Software filter, 175
Special decimal mode, 150
Spiner, 87
Spiner Program, 93
SQSTAT, 252
Square status, 269
Square wave, 22
Strategy, 225
SYM, 4
T1CL, 6, 83
T1L-L, 65
Threat potential, 226
Tic-Tac-Toe, 218
Tic-Tac-Toe Flowchart, 248
Tic-Tac-Toe Program, 280
TIMER, 65
Timer, 65, 83, 198, 256
Timer 1, 175
TONE, 39, 70, 130, 135
Translate, 41
Translate Program, 49
Trap, 235, 239, 264, 267
Trap pattern, 241
Two-level loop, 211
Two-ply analysis, 237
Unprotect system, 198
UPDATE, 273
Value computation, 226
VIA, 8
V1A memory map, 66
Visual feedback, 163
WAIT, 98
Wheel pointer, 103, 120
WIN, 128
Win, 259
Win potential, 225
WINEND, 129

SYBEX BIBLIOGRAPHY

VIDEO COURSES
Microprocessors – 12 hours (Ref. VI)
Military Microprocessor Systems – 6 hours (Ref. V3)
Bit-Slice – 6 hours (Ref. V5)
Microprocessor Interfacing Techniques – 6 hours (Ref. V7)

AUDIO COURSES
Introduction to Microprocessors – 2½ hours (Ref. S1)
Programming Microprocessors – 2½ hours (Ref. S2)
Designing a Microprocessor – 2½ hours (Ref. S3)
Microprocessors – 12 hours (Ref. S8)
Programming Microprocessors – 10 hours (Ref. S82)
Military Microprocessor Systems – 6 hours (Ref. S83)
Bit-Slice – 6 hours (Ref. S85)
Industrial Microprocessor Systems – 4½ hours (Ref. S86)
Microprocessor Interfacing Techniques – 6 hours (Ref. S87)
Introduction to Personal Computing – 2½ hours (Ref. S810)

REFERENCE TEXTS
Practical Pascal (Ref. C102)
Introduction to Personal and Business Computing (Ref. C200)
Microprocessors (Ref. C201)
The 6502 Series
Volume-1: Programming the 6502 (Ref. C202)
Volume-2: Programming Exercises for the 6502 (Ref. C203)
Volume-3: 6502 Applications Book (Ref. D302)
Volume-4: 6502 Games (Ref. G402)
Microprocessor Interfacing Techniques (Ref. C207)
Programming the Z80 (Ref. C280)
Programming the Z8000 (Ref. C281)
CP/M Handbook – with MP/M (Ref. D300)
International Microprocessor Dictionary – 10 languages (Ref. IMD)
Microprocessor Lexicon (Ref. X1)
Microprogrammed APL Implementation (Ref. Z10)

SOFTWARE
BAS 65™ 6502 Assembler in Microsoft BASIC (Ref. BAS 65)
8080 Simulator for KIM – Cassette Tape or 5" Diskette (Ref. S6580-KIM)
8080 Simulator for APPLE – Cassette Tape or 5" Diskette (Ref. S6580-APL)

SELF-STUDY SYSTEM
Computeacher™ (Ref. CPT)
Games Board™ (Ref. CPTG)
FOR A COMPLETE CATALOGUE OF OUR PUBLICATIONS

U.S.A.
2344 Sixth Street
Berkeley, California 94710
Tel: (415) 848-8233
Telex: 336311

EUROPE
18 rue Planchat
75020 Paris, France
Tel: (1) 3703275
Telex: 211801