Contents

Preface vii
Introduction 1

1 Optional Hardware Support 2
Connecting the System 4
Games Board Interconnect 7
The Keyboard Input Routine 13

2 Generating Square Waves (Music Player) 20
Introduction 20
The Rules 20
A Typical Game 22
The Connections 22
The Algorithm 22
The Program 23

3 Pseudo Random Number Generator (Translate) 41
Introduction 41
The Rules 41
A Typical Game 42
The Algorithm 43
The Program 43

4 Hardware Random Number Generator (Hexguess) 59
Introduction 59
The Rules 59
A Typical Game 59
The Algorithm 60
The Program 60

5 Simultaneous Input/Output (Magic Square) 73
Introduction 73
The Rules 73
A Typical Game 76
The Algorithm 78
The Program 80

6 Simple Real Time Simulation (Spinner) 87
Introduction 87
The Rules 87
The Algorithm 88
The Program 89
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Real Time Simulation (Slot Machine)</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>The Rules</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>A Typical Game</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>The Algorithm</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>The Program</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>Real Time Strategies (Echo)</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>The Rules</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>A Typical Game</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>The Algorithm</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>The Program</td>
<td>144</td>
</tr>
<tr>
<td>9</td>
<td>Using Interrupts (Mindbender)</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>The Rules</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>A Typical Game</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>The Algorithm</td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>The Program</td>
<td>167</td>
</tr>
<tr>
<td>10</td>
<td>Complex Evaluation Technique (Blackjack)</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>The Rules</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>A Typical Game</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>The Program</td>
<td>194</td>
</tr>
<tr>
<td>11</td>
<td>Artificial Intelligence (Tic-Tac-Toe)</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>The Rules</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>A Typical Game</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>The Algorithm</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>The Program</td>
<td>247</td>
</tr>
<tr>
<td></td>
<td>Appendices</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>A. 6502 Instructions—Alphabetic</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>B. 6502 Instruction Set—Hex and Timing</td>
<td>288</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>290</td>
</tr>
</tbody>
</table>
Preface

This book has been designed to teach you advanced programming techniques for the 6502 microprocessor in a systematic and progressive way. Developing a program involves devising a suitable algorithm and appropriate data structures, and then coding the algorithm. In the case of a microprocessor such as the 6502, the design of the algorithm and the data structures is generally constrained by three conditions:

1. The amount of memory available is often limited or must be minimized; i.e., the program must be terse.

2. The highest possible execution speed may be required. Efficient coding of the program into assembly level language instructions then becomes an essential consideration. In particular, the use of registers must be optimized.

3. The specific input/output design requires an understanding of the input and output chips and their programming.

Thus, when evaluating designs for an algorithm and data structures, the programmer must weigh the merits of the various techniques in terms of his skill, the memory limitations, the require speed of execution, and the overall probability of success.

Advanced programming for the 6502, therefore, involves knowledge of all the chips that may be affected by the program, in addition to the usual programming skills concerned with the algorithm, the data structures, and the efficient use of internal instructions and registers. This book provides a comprehensive and complete overview of all the important techniques required to program a 6502 system efficiently. The book has been designed as an educational text. Each chapter introduces new concepts, chips, or techniques in turn. In the final chapters more complex algorithms are presented, which integrate the techniques presented throughout the book.

For clarity and consistency, this book uses a specific 6502-based system on which all the programs will run. The details are presented in Chapter 1. However, the programs and techniques presented here are applicable to all 6502-based systems. Similarly, all the programs studied in this book are presented in the form of realistic games involving successively all the techniques described. They cover most types of applications ranging from simple input/output techniques to sophisticated real-time simulations, including the handling of interrupts and the design of complex data structures.
A case study approach is used, and each chapter contains the following:

1. A description of the concepts and techniques to be studied
2. The specifications of the program's behavior and a typical session with the program, i.e., the problem to be solved
3. The algorithm(s): theory of operation, design, and trade-offs
4. The actual program: data structures, programming techniques, specific subroutines, merits of alternative techniques, and a complete program listing.

Variations and exercises are also proposed in each chapter.

Thus, you will first study the definition of the problem, then observe the expected program behavior, and then learn how to devise a possible solution (algorithm plus data structures). Finally, you will design a complete program for this algorithm in 6502 assembly level language, paying specific attention to the required data structures, the efficient use of registers, the input/output chips, and the techniques used for efficient programming.

You will sharpen your skills at using input/output techniques including timers and interrupts. But most importantly, you will be consistently reminded of the trade-offs between ease in programming, use of memory, efficiency of execution, and algorithmic improvements by use of specialized hardware or software techniques.

In order to learn the advanced programming techniques presented in this book, it is not necessary to build any actual hardware. However, it is necessary to write programs on your own along the ten chapters of this book. By showing you and explaining in detail the design of many actual programs, the author hopes to facilitate your next step: actual programming.
Acknowledgments

The author would like to acknowledge the contributions of Chris Williams and Eric Novikoff, who thoroughly checked all of the games 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.

The author would also like to express his appreciation to Rockwell International and in particular, to Scotty Maxwell, who made available to him one of the very first system 65 development systems. The availability of this powerful development tool, at the time the first version of this book was being written, was a major help for the accurate and efficient check-out of all the programs.
1. Introduction

In order to learn the techniques and study the program examples presented in this book, no specific equipment is required. However, the availability of a 6502-based system is a major advantage to develop and test 6502 programs on your own. Bear in mind that each 6502-based system will have a somewhat different input/output configuration. The techniques presented in this book are applicable to all, and the programs can be easily adapted once you understand input/output operations.

To read this book, you should be familiar with the 6502 instruction set and basic programming techniques on the level of Programming the 6502. A basic knowledge of input/output techniques is also recommended. (This topic is covered in 6502 Applications.)

The programs presented in Chapters 2 through 11 range from simple to complex. In order to implement these programs, algorithms will be devised and data structures will be designed. This is the process any disciplined computer programmer must go through when designing a
program solution for a given problem. The ten case studies presented in this book will also familiarize you with common input/output techniques. Toward the end of the book, you will find that the problems presented pose increasingly complex intellectual challenges to devising efficient solutions. All the strategies presented in this book, including the one used for the Tic-Tac-Toe game in Chapter 1, are believed to be original. These strategies and the design process will be analyzed in detail. As an additional design constraint intended to teach you efficient design, all the algorithms and data structures presented in this book have been designed to result in a program that can reside within less than 1K of available memory.

The programs presented in this book have been tested on actual hardware by many users and have been found to be error-free in the conditions under which they were tested. As in any large set of programs, however, inadequacies or improvements may be found.

OPTIONAL HARDWARE SUPPORT

The programs contained in this book can be developed on any 6502-based system. However, in order to be executed they require a specific input/output environment. For the sake of simplicity, a uniform hardware environment has been used throughout this book. It assumes a 6502-based board, the SYM board (by Synertek Systems), and an additional input/output board, called the Games Board, which can be easily built. For completeness, an overview of the SYM board and a complete description of the Games Board will be provided in this chapter. However, it is not necessary to purchase or build these boards to understand the information presented in this book. The Games Board may also be adapted easily to other 6502-based computers such as Commodore or Apple computers. The programs remain essentially unchanged except for input/output device allocations.

The Games Board can also be simulated on a standard terminal by displaying information on a CRT screen and capturing input from a normal alphanumeric keyboard.

A photograph of the Games Board is shown in Figure 1.1. The keyboard on the right is used to provide inputs to the microcomputer board, while the LEDs on the left are used to display the information sent by the program. The specific use of the keys and the LEDs will be explained in each chapter. A speaker is also provided for sound effects. It can be mounted in an enclosure (box) for improved sound quality (see Figure 1.2). This input/output board can be easily built at home from a small number of low cost components.
Fig. 1.1: The Games Board

Fig. 1.2: Enclosure May Be Used for Improved Sound
CONNECTING THE SYSTEM

If you wish to assemble the actual system and build the input/output board, read on. If you are not interested in building any actual hardware, proceed to the description of an important program subroutine that will be used repeatedly in this book: the keyboard input routine.

Four essential components are required to assemble the Games Board:

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. 1.3: Two Wires Must Be Connected to the Power Supply

Fig. 1.4: The Games Board is Connected to the SYM with 2 Connectors (Note also Power and Cassette Connectors)
Fig. 1.5: Connecting the Cassette Recorder

Fig. 1.6: The System is Ready to be Used
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

---

**Fig. 1.7: Games Board Elements (Prototype)**
Figure 1.8: The LEDs

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.
**Fig. 1.9: Decoder Connection to Keyboard**

**Fig. 1.10: Detecting a Key Closure**
Fig. 1.11: LED Connection
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” x 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.
Fig. 1.13: Detail of LED Connection to the Ports

Assembly

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 corresponds 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.

![Diagram of VIA Connection to Keyboard Decoder](image)

**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 locations 100 to 12E. It is important to remember that this is the low 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 VIA #3. The data direction register for Port A is set for input (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 location 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.

Fig. 1.18: "Production" Games Board

Fig. 1.19: Removing the Cover
2. Generating Square Waves (Music Player)

INTRODUCTION

This program will teach you how to synthesize frequencies by generating square waves. It will use a table-driven algorithm to generate tones and play music. It will make systematic use of indexed addressing techniques.

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 "3" 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>KEY NUMBER</th>
<th>NOTE</th>
<th>KEY NUMBER</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>G</td>
<td>8</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>BACK</td>
</tr>
</tbody>
</table>

Fig. 2.1: Playing Music on the Keyboard
### 9th Symphony:

\[
5 - 5 - 6 - 8 - 8 - 6 - 5 - 4 - 3 - 3 - 4 - 4 - 5 - 5 - 4 - 4 - D - 5 - 5 - 6 - 8 - 6 - 5 - 4 - 3 - 3 - 4 - 4 - 5 - 5 - 4 - 3 - 4 - D
\]

### Clementine:

\[
3 - 3 - D - 2 - D - 5 - 5 - 5 - D - 3 - D - 3 - 5 - 8 - D - D - 8 - 6 - 5 - 4 - D - D - 4 - 5 - 6 - D - 5 - 4 - 5 - D - 3 - D - 3 - 5 - 4 - D - D - 2 - 3 - 4 - 3
\]

### Frere Jacques:

\[
3 - 4 - 5 - 3 - 3 - 4 - 5 - 3 - 5 - 6 - 8 - D - 5 - 6 - 8 - D - 8 - A - 8 - 6 - 5 - D - 3 - D - 3 - D - 3 - 5 - 6 - 8 - 6 - 5 - 5 - 8 - 8 - 6 - 4 - 3
\]

### Jingle Bells:

\[
5 - 5 - 5 - D - 5 - 5 - 5 - D - 5 - 8 - 3 - 4 - 5 - D - D - D - 6 - 6 - 6 - 6 - 5 - 5 - 5 - 8 - 8 - 6 - 4 - 3
\]

### London Bridge:

\[
8 - A - 8 - 6 - 5 - 6 - 8 - D - 4 - 5 - 6 - D - 5 - 6 - 8 - D - 8 - A - 8 - 6 - 5 - 6 - 8 - D - 4 - 5 - D - 8 - D - 5 - 3
\]

### Mary Had a Little Lamb:

\[
5 - 4 - 3 - 4 - 5 - 5 - 5 - D - 4 - 4 - 4 - D - 5 - 8 - 8 - D - 5 - 4 - 3 - 4 - 4 - 5 - 5 - 4 - 4 - 5 - 4 - 3
\]

### Row Row Row Your Boat:

\[
3 - D - 3 - D - 3 - 4 - 5 - D - 5 - 4 - 5 - 6 - 8 - D - D - D - C - C - 8 - 8 - 5 - 5 - 3 - 3 - 8 - 6 - 5 - 4 - 3
\]

### Silent Night:

\[
8 - D - D - A - 8 - D - 5 - D - D - D - 8 - D - D - A - 8 - D - 5 - D - D - 3 - D - D - 3 - D - B - D - D - D - C - C - C - D - 8 - D - D - C - D - 8 - 5 - 8 - D - 6 - D - 4 - D - 3
\]

### Twinkle Twinkle Little Star:

\[
3 - 3 - 8 - 8 - A - 8 - D - 6 - 6 - 5 - 5 - 4 - 4 - 3 - D - 8 - 8 - 6 - 6 - 5 - 5 - 4 - D - 3 - 3 - 8 - 8 - A - A - 8 - D - 6 - 6 - 5 - 5 - 4 - 4 - 3
\]

---

**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 \( \frac{1}{(2 \times 195)} = 0.002564 \) to

\[ \frac{T}{2} \quad \text{SQUARE WAVE} \quad \text{SPEAKER} \]

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

Fig. 2.3: Generating a Tone
1/(2 \times 523) = .000956 \text{ seconds. 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:

\[ 1 : \sqrt[13]{2} \]

The frequencies for the middle C octave are given in Figure 2.5. When computing the corresponding frequencies of the higher or the
lower octave, they are simply obtained by multiplying by two, or dividing by two, respectively.

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 µs cycle time. In the program, the "loop index," or iteration counter is used to count the number of 10 µs cycles executed. The loop will result in a total delay of:

\[(\text{loop index}) \times 10 - 1 \text{ microseconds}\]
GENERATING SQUARE WAVES

<table>
<thead>
<tr>
<th>NOTE</th>
<th>FREQUENCY (HERTZ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>220.00</td>
</tr>
<tr>
<td>A#</td>
<td>223.08</td>
</tr>
<tr>
<td>B</td>
<td>246.94</td>
</tr>
<tr>
<td>C</td>
<td>261.62</td>
</tr>
<tr>
<td>C#</td>
<td>277.18</td>
</tr>
<tr>
<td>D</td>
<td>293.66</td>
</tr>
<tr>
<td>D#</td>
<td>311.13</td>
</tr>
<tr>
<td>E</td>
<td>329.63</td>
</tr>
<tr>
<td>F</td>
<td>349.23</td>
</tr>
<tr>
<td>F#</td>
<td>369.99</td>
</tr>
<tr>
<td>G</td>
<td>391.99</td>
</tr>
<tr>
<td>G#</td>
<td>415.30</td>
</tr>
</tbody>
</table>

Fig. 2.5: Frequencies for the Middle C Octave

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       STA   FREQ
            LDA #$FF
            STA DDRB
            LDA #$00
            LDX DUR

 FL2        LDY   FREQ
            DEY
            CLC
            BCC .+2
            BNE FL1
            EOR #$FF
            STA OPB
            DEX
            BNE FL2
            RTS

 FL1        INNER LOOP
            OUTER LOOP
```

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:

\[T = 10^4/N\]

Finally, the above equation becomes:

\[\text{note constant} = (10^4/N - 2 \times \text{OD})/2 \times \text{ID}\]

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:

\[\text{note constant} = (10^4/N - 2 \times 13)/2 \times 10\]
\[= \frac{1000000/261.62 - 26}{20}\]
\[= 190 \text{ (or BE in hexadecimal)}\]

It can be verified that this corresponds to the fourth entry in the table.
at address NOTAB (see Figure 2.9 at the end of the listing, at address 02C4). The note constants are shown in Figure 2.6.

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

Fig. 2.7: Music Flowchart
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
MUSIC PLAYER PROGRAM
USES 16 - KEY KEYBOARD AND BUFFERED SPEAKER
PROGRAM PLAYS STORED MUSICAL NOTES.
THERE ARE TWO MODES OF OPERATION: INPUT AND PLAY.
INPUT MODE IS THE DEFAULT, AND ALL NON-COMMAND KEYS
PRESSED (0-D) ARE STORED FOR REPLAY. IF AN OVERFLOW
FCCURRS, THE USER IS WARNED WITH A THREE-TONE WARNING.
THE SAME WARBLING TONE IS ALSO USED TO SIGNAL A
RESTART OF THE PROGRAM.

GETKEY = $100
PILEN = $100  # LENGTH OF NOTE LIST
TEMP = $01   # TEMPORARY STORAGE
PTR = $02    # CURRENT LOCATION IN LIST
FREQ = $03   # TEMPORARY STORAGE FOR FREQUENCY
DUR = $04    # TEMP STORAGE FOR DURATION
TABEG = $300 # TABLE TO STORE MUSIC
DPB = $AC00  # VIA OUTPUT PORT B
DDRB = $AC02 # VIA PORT B DIRECTION REGISTER
= $200 # ORIGIN

COMMAND LINE INTERPRETER
IF AS INPUT MEANS RESET POINTERS, START OVER.
SE MEANS PLAY CURRENTLY STORED NOTES
ANYTHING ELSE IS STORED FOR REPLAY.

02001 A9 00  # START LDA $0  # CLEAR NOTE LIST LENGTH
02021 85 00  # STA PILEN
02041 18     # CLC  # CLEAR NIBBLE MARKER
02051 20 00 01  # NXKEY JSR GETKEY
02081 C9 0F   # CMP #15  # IS KEY #15?
020A1 00 05   # BNE NXSTT  # NO, DO NEXT TEST
020C1 20 87 02  # JSR BEEP3  # TELL USER OF CLEARING
020F1 90 EF   # BCC START  # CLEAR POINTERS AND START OVER
02111 C9 0E   # NXSTT CMP #14  # IS KEY #14?
02131 D0 06   # BNE NNUMKEY  # NO, KEY IS NOTE NUMBER
02151 20 48 02  # JSR PLAYEN  # PLAY NOTES
02181 10     # CLC
02191 90 EA   # BCC NXKEY  # GET NEXT COMMAND

ROUTINE TO LOAD NOTE LIST WITH NOTES

021B1 85 01  # NUMKEY STA TEMP  # SAVE KEY, FREE A
021D1 20 70 02  # JSR PLAYIT  # PLAY NOTE
02201 A5 00  # LDA PILEN  # GET LIST LENGTH
02221 C9 FF   # CMP $FF  # OVERFLOW?
02241 D0 05   # BNE OK  # NO, ADD NOTE TO LIST
02261 20 87 02  # JSR BEEP3  # YES, WARN USER
02291 90 DA   # BCC NXKEY  # RETURN TO INPUT MODE
022B1 4A      # OK LSR A  # SHIFT LOW BIT INTO NIBBLE POINTER
022C1 AB      # TAY  # USE SHIFTED NIBBLE POINTER AS
# BYTE INDEX
022D1 A5 01  # LDA TEMP  # RESTORE KEY
022F1 B0 09   # BCS FINBYT  # IF BYTE ALREADY HAS 1 NIBBLE:
# FINISH IT AND STORE
02311 29 0F   # AND $20000111  # 1ST NIBBLE, MASK HIGH NIBBLE
02331 99 00 03  # STA TABEG-Y  # SAVE UNFINISHED 1/2 BYTE
02361 E6 00   # INC PILEN  # POINT TO NEXT NIBBLE
02381 90 CB   # BCC NXKEY  # GET NEXT KEYSTROKE
023A1 0A      # FINBYT ASL A  # SHIFT NIBBLE 2 TO HIGH ORDER
023B1 0A      # ASL A
023C1 0A      # ASL A
023D1 0A      # ASL A
023E1 19 00 03  # ORA TABEG-Y  # JOIN 2 NIBBLES AS BYTE
02411 99 00 03  # STA TABEG-Y  # ... AND STORE.
02441 E6 00   # INC PILEN  # POINT TO NEXT NIBBLE IN NEXT BYTE
02461 90 BD   # BCC NXKEY  # RETURN

Fig. 2.9: Music Program
ADVANCED 6502 PROGRAMMING

; ROUTINE TO PLAY NOTES
; 0248: A2 00 PLAYM LDX #0 ; CLEAR POINTER
024A: B6 02 STX PTR
024C: A5 02 LDA PTR ; LOAD ACUM W/CURRENT PTR VAL
024E: 4A LOOP LSR A ; SHIFT NIBBLE INDICATOR INTO CARRY
0250: F4 AA TAX ; FUSE SHIFTED NIBBLE POINTER
                ; AS BYTE POINTER
0253: BD 00 03 LDA TABEL,#X ; LOAD NOTE TO PLAY
0255: B0 04 RCS ENDYET ; LOW NIBBLE USED; GET HIGH
0255: 29 0F AND #20000011 ; MASK OUT HIGH Bits
0258: 90 06 BCC FINISH ; PLAY NOTE
0259: 29 F0 ENDBYET AND #11110000 ; THROW AWAY LOW NIBBLE
025B: 4A LSR A ; SHIFT INTO LOW
025C: 4A LSR A
025D: 4A LSR A
025E: 4A LSR A
025F: 20 70 02 FINISH JSR PLAYIT ; CALCULATE CONSTANTS & PLAY
0262: A2 20 LBX #$20 ; BETWEEN-NOTE DELAY
0264: 20 9C 02 JSR DELAY
0267: E6 02 INC PTR ; ONE NIBBLE USED
0269: A5 02 LDA PTR
026B: C5 00 CMP PILEN ; END OF LIST?
026D: 90 DF BCC LOOP ; NO, GET NEXT NOTE
026F: 60 RTS ; DONE

; ROUTINE TO DO TABLE LOOK UP, SEPARATE REST
; 0270: C9 0D PLAYIT CMP #13 ; REST?
0272: D0 06 BNE SOUND ; NO.
0274: A2 54 LBX #$54 ; DELAY=NOTE LENGTH=.21SEC
0277: 20 9C 02 JSR DELAY
0279: 60 RTS
027A: AA SOUND TAX ; USE KEY# AS INDEX.
027B: BD D1 02 LDA DURAT,#X ... TO FIND DURATION.
027E: B5 04 STA DUR ; STORE DURATION FOR USE
0280: BD C4 02 LDA NOTAB,#X ; LOAD NOTE VALUE
0283: 20 AB 02 JSR TONE
0286: 60 RTS

; ROUTINE TO MAKE 3 TONE SIGNAL
; 0287: A9 FF BEEP3 LDA #$FF ; DURATION FOR BEEPS
0289: B5 04 STA DUR
028B: A9 4B LDA #$4B ; CODE FOR E2
028D: 20 AB 02 JSR TONE ; 1ST NOTE
0290: A9 3B LDA #$3B ; CODE FOR D2
0292: 20 AB 02 JSR TONE
0295: A9 4B LDA #$4B
0297: 20 AB 02 JSR TONE
0299: 18 CLC
029B: 60 RTS

; VARIABLE-LENGTH DELAY
; 029C: A0 FF DELAY LDY #$FF
029E: EA DLY NOP
029F: D0 00 BNE +2
02A1: BB DEY
02A2: D0 FA BNE DLY ; 10 US LOOP
02A4: CA DEX
02A5: D0 F5 BNE DELAY ; LOOP TIME = 2556*CXJ
02A7: 60 RTS

; ROUTINE TO MAKE TONE: $ OF 1/2 CYCLES IS IN 'DUR',
; AND 1/2 CYCLE TIME IS IN A. LOOP TIME=20*IA1+26 US

---

Fig. 2.9: Music Program (Continued)

32
GENERATING SQUARE WAVES

\[ \text{Since two runs through the outer loop makes one cycle of the tone.} \]

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>02A6</td>
<td>05 03</td>
<td>TONE STA FREQ (FREQ is temp for # of cycles)</td>
</tr>
<tr>
<td>02AA</td>
<td>A9 FF</td>
<td>LDA $FF (set up data direction reg)</td>
</tr>
<tr>
<td>02AC</td>
<td>8D 02 AC</td>
<td>STA DDRB</td>
</tr>
<tr>
<td>02AF</td>
<td>A9 00</td>
<td>LDA $00 (A is sent to port, start hi</td>
</tr>
<tr>
<td>02B1</td>
<td>A6 04</td>
<td>LDX DUR</td>
</tr>
<tr>
<td>02B5</td>
<td>A4 03</td>
<td>FL2 LDY FREQ</td>
</tr>
<tr>
<td>02B6</td>
<td>8B</td>
<td>BFL</td>
</tr>
<tr>
<td>02B7</td>
<td>1B</td>
<td>CLC</td>
</tr>
<tr>
<td>02BB</td>
<td>90 00</td>
<td>BNE FL1 (inner, 10 us loop)</td>
</tr>
<tr>
<td>02BD</td>
<td>49 FF</td>
<td>EOR $FF (complement I/O port)</td>
</tr>
<tr>
<td>02BE</td>
<td>BD 00 AC</td>
<td>STA DFB (AND set it)</td>
</tr>
<tr>
<td>02CF</td>
<td>0A</td>
<td>DEX</td>
</tr>
<tr>
<td>02D1</td>
<td>D0 F0</td>
<td>BNE FL2 (outer loop)</td>
</tr>
<tr>
<td>02D3</td>
<td>60</td>
<td>RTS</td>
</tr>
</tbody>
</table>

TABLE OF NOTE CONSTANTS

- Contains:
  - Octave below middle C1: G,A,B
  - Octave of middle C1: C,d,e,f,f,g,g,a,b
  - Octave above middle C1: C

02C4: FE NOTAB .BYTE $FE,$F2,$C9,$BE,$A9,$96,$8E

02C5: 02
02C6: C9
02C7: BE
02C8: A9
02C9: 96
02CA: 8E
02CB: 86
02CC: 7E
02CD: 77
02CE: 70
02CF: 64
02D0: 5E

TABLE OF NOTE DURATIONS IN # OF 1/2 CYCLES

- Set for a note length of about 0.21 sec.

02D1: 55 DURTAB .BYTE $55,$60,$6B,$72,$80,$8F,$94
02D2: 60
02D3: 68
02D4: 72
02D5: 80
02D6: 8F
02D7: 94
02DB: A1 .BYTE $A1,$A2,$B5,$BF,$D7,$E4
02DP: AA
02DA: A5
02DB: BF
02DC: D7
02DD: E4

SYMBOL TABLE:

- GETKEY 0100 PILLEN 0000 TEMP 0001
- PTR 0002 FREQ 0003 DUR 0004
- TABEG 0300 OPB AC00 DDRB AC02
- START 0200 MXKEY 0205 MXST 0211
- NUMKEY 021B OK 022B FINTAB 023A
- PLAYEM 0248 LOOP 024E ENDTAB 0259
- FINISH 025F PLAYIT 0270 SOUD 027A
- BEEP3 0267 DELAY 029C DLY 029E
- TONE 02A8 FL2 02B3 FL1 02B5
- NOTAB 02C4 DURTAB 02D1

---

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.

```
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 PILEN to zero. The corresponding routine appears below:
GENERATING SQUARE WAVES

BEEP3

LDA #$FF  Beep duration constant
STA DUR
LDA #$4B  Code for E2
JSR TONE 1st note
LDA #$38  Code for D2
JSR TONE 2nd note
LDA #$4B  Code for E2
JSR TONE 3rd note
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 PILEN  Get length of list
CMP #$FF  Overflow?
BNE OK  No: add note to list
JSR BEEP3  Yes: warn player
BCC NKEY  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 LDA TEMP  Use as byte index  
      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 TABEG,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 #%00001111     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:

![Diagram of the list structure](image)

**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 = 6X (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:
PLAYEM
LDX #0
STX PTR
LDA PTR

LOOP
LSR A
TAX
LDA TABEG,X
BCS ENDBYT
AND #\%00001111
BCC FINISH

ENDBYT
AND #\%11110000
LSR A
LSR A
LSR A
LSR A

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 ex-
tracted 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 PLAY IT
Play note

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

LDX #$20
JSR DELAY
INC PTR
LDA PTR
CMP PILEN
BCC LOOP
RTS

Delay constant
Delay between notes
One nibble used
Check for end of list
No: get next note
Done

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 ap-
propriate half-period value from the table at address NOTAB into the
A register, using indexed addressing, and calls subroutine TONE to play it:

```
PLAYIT
  CMP #13
  BNE SOUND
  LDX #$54
  JSR DELAY
  RTS

SOUND
  TAX
  LDA DURTAB,X
  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:

\[(20 \times A + 26) \text{ } \mu s\]

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
```
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  10 μs inner loop
```

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?
3. Pseudo Random Number Generator
(Translate)

INTRODUCTION

This program will use a pseudo random number generator, display patterns from tables, measure elapsed time, and generate delays. It will check your knowledge of basic input/output techniques before we proceed to more complex concepts.

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
the winner by lighting up either the left-most or the right-most three LEDs of the bottom row. The winner is the player with the shortest guessing time. The game is continued until one player wins ten times. He or she then wins the match. The computer signals the match winner by blinking the player’s three LEDs ten times. At the end of the match, control is returned to the SYM-1 monitor.

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

42
"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
“00000010.” Outputting a “1” will be accomplished with the pattern “10000010.” Outputting “2” will be accomplished with the pattern “00100010.” Outputting “3” 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.”

```
 7 6 5 4 3 2 1 0
0 0 1 1 1 0 1 0
  1 1 1 0
```

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
Fig. 3.5: LED Connections
variable locations with the statement:

\[ \langle \text{VARIABLE NAME} \rangle = \langle \text{MEMORY ADDRESS} \rangle \]

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:

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

DDR3A is conditioned as input:

```
LDA #0
STA DDR3A
```

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

```
STA PLYR1
STA PLYR2
```

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:

```
MOVE
LDA #%01111001
STA PORT1A  \quad \text{Display right arrow}
```
ADVANCED 6502 PROGRAMMING

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 CNT1L, CNT1H:

LDA CNTLO
STA CNT1L
LDA CNTHI
STA CNT1H

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
LINE # LOC  CODE        LINE
0002 0000  ; 'TRANSFORM'
0003 0000  ; PROGRAM TO TEST 2 PLAYER'S SPEED
0004 0000  ; FIN TRANSFORMING A BINARY NUMBER TO A SINGLE
0005 0000  ; HEXADECIMAL DIGIT, EACH PLAYER IS GIVEN A
0006 0000  ; TURN, AS SHOWN BY A LIGHTED LEFT OR RIGHT
0007 0000  ; POINTER, THE NUMBER WILL SUDDENLY Flash ON
0008 0000  ; LEDS 12-15, ACCOMPANIED BY THE LIGHTING
0009 0000  ; OF LED #10, THE PLAYER MUST THEN
0010 0000  ; PUSH THE CORRESPONDING BUTTON, AFTER
0011 0000  ; BOTH PLAYERS TAKE TURNS, RESULTS ARE
0012 0000  ; SHOWN ON BOTTOM ROW, AFTER 10 WINS,
0013 0000  ; A PLAYER'S RESULTS WILL FLASH,
0014 0000  ; SHOWING THE BETTER PLAYER, THEN
0015 0000  ; THE GAME RESTARTS.
0016 0000  ;
0017 0000  ; /O:
0018 0000  ;
0019 0000  ; PORT1A = #A001  LEDS 1-8
0020 0000  ; PORT1B = #A001  LEDS 9-15
0021 0000  ; DDR1A = #A003
0022 0000  ; DDR1B = #A002
0023 0000  ; PORT3A = #A001  KEY STROBE INPUT.
0024 0000  ; PORT3B = #A001  KEY # OUTPUT.
0025 0000  ; DDR3A = #A003
0026 0000  ; DDR3B = #A002
0027 0000  ;
0028 0000  ; /VARIABLE STORAGE:
0029 0000  ;
0030 0000  ; x = #00
0031 0000  ;
0032 0000  ; TEMP =###1  CTNH###1  TEMPORARY STORAGE FOR AMT. OF
0033 0000  ; CTN###1  TIME PLYR USES TO GUESS.
0034 0000  ; CTNL#1  CTNH###1  I AMT. OF TIME PLYR USES TO GUESS.
0035 0000  ; CNTL#1  CNTL###1  I AMT. OF TIME PLYR USES TO GUESS.
0036 0000  ; PLYR###1  PLYR###1  I SCORE OF # WON FOR PLYR.
0037 0000  ; PLYR###1  PLYR###1  I PLAYER 2 SCORE.
0038 0000  ; NUMBER###1  NUMBER###1  I STORES NUMBER TO BE GUESSED.
0039 0000  ; SCR###1  SCR###1  I SCRATCHPAD FOR RND. # GEN.
0040 0000  ;
0041 0000  ; TABLE OF 'REVERSED' NUMBERS FOR DISPLAY
0042 0000  ; IN BITS 3-8 OF PORT1B, OR LEDS 12-15.
0043 0000  ;
0044 0000  ;
0045 0000  ;
0046 0000  ; 02  NUMTAB .BYTE X00000010
0047 0000  ; 02  .BYTE X00100010
0048 0000  ; 02  .BYTE X00110010
0049 0000  ; 02  .BYTE X00120010
0050 0000  ; 02  .BYTE X00130010
0051 0000  ; 02  .BYTE X00140010
0052 0000  ; 02  .BYTE X00150010
0053 0000  ; 02  .BYTE X00160010
0054 0000  ; 02  .BYTE X00170010
0055 0000  ; 02  .BYTE X00180010
0056 0000  ; 02  .BYTE X00190010
0057 0000  ; 02  .BYTE X001A0010
0058 0000  ; 02  .BYTE X001B0010
0059 0000  ; 02  .BYTE X001C0010
0060 0000  ; 02  .BYTE X001D0010
0061 0000  ; 02  .BYTE X001E0010
0062 0000  ;
0063 0000  ;
0064 0000  ;
0065 0000  ;
0066 0000  ;
0067 0000  ; 02  START LDA #8F
0068 0000  ; 02  STA DDR1A
0069 0000  ; 02  STA DDR1B
0070 0000  ; 02  STA DDR3B
0071 0000  ; 02  LDA #0
0072 0000  ; 02  STA PLYR1
0073 0000  ; 02  STA PLYR2
0074 0000  ; 02  MOVE LDA #20111100
0075 0000  ; 02  STA PORTIA
0076 0000  ; 02  STA PORT1B
0077 0000  ; 02  STA CNTL
0078 0000  ; 02  STA CNTMI
0079 0000  ; 02  STA CNTLD
0080 0000  ; 02  STA CNTM2
0081 0000  ; 02  JMP PLYR 1'S TIME.
0082 0000  ; 02  STA CNTL
0083 0000  ; 02  STA CNTL
0084 0000  ; 02  LDA CNTMI

Fig. 3.6: Translate Program
```assembly
; Fig. 3.6: Translate Program (Continued)

0085 0228 BS 03 STA CNT1H
0086 022D AP 3C LDA #$00111100 ; SHOW LEFT ARROW.
0087 022F BD 01 AO STA PORTIA
0088 0332 A9 01 LDA #$1
0089 0234 BD 00 AO STA PORTIB
0090 0237 A9 00 LDA #$0 ; CLEAR COUNTERS.
0091 0239 BS 02 STA CNT0H
0092 023A BS 01 STA CNT0L
0093 023D 20 BD 02 JSR PLAY ; GET PLAYER 2'S TIME.
0094 023E 20 01 BD 02 LDA CNT1H ; GET PLAYER 2'S COUNT AND...
0095 0240 A5 01 CMP CNT1H ; COMPARE TO PLAYER 1'S.
0096 0242 C5 03 BNE EQUAL ; CHECK LOW ORDER BYTES TO RESOLVE WINNER.
0097 0244 F0 04 BNE MOV ; PLAYER 2 HAS SMALLER COUNT; SHOW IT.
0098 0246 90 27 BCS PLR2 ; PLAYER 1 HAS SMALLER COUNT; SHOW IT.
0099 0248 BD 00 AO BCS PLR1 ; IHT BYTES WERE EQUAL; SD
0100 024A A5 02 EQUAL ; CHECK LOW BYTES.
0101 024C 01 BD 00 AO CMP CNT1H ; COMPARE SCORES.
0102 024E 90 1F BCC PLR2 ;PLAYER 2 WINS; SHOW IT.
0103 0250 B0 00 BCS PLR1 ;PLAYER 1 WINS; SHOW IT.
0104 0253 A9 F0 PLR1 LDA #$11110000 ; LIGHT RIGHT SIDE OF BOTTOM ROW
0105 0254 BD 00 AO STA PORTIB ; TO SHOW WIN.
0106 0257 A9 00 STA PORTIA
0107 0259 BD 01 AO LDA #$0 ; CLEAR LOW LEDS.
0108 025C A9 40 STA PORTIB
0109 025E 20 E3 02 JSR DELAY ; WAIT A WHILE TO SHOW WIN.
0110 0261 EA 05 INC PLYR1 ; IPLAYER 1 WINS ONE MORE...
0111 0263 A9 0A LDA #$10 ; ...HAS HE WON 10?
0112 0265 C5 05 CMP PLYR1 ; IF NOT: PLAY ANOTHER ROUND.
0113 0267 50 0A BNE MOV ; IF NOT: PLAY ANOTHER ROUND.
0114 0269 A9 F0 CMP PLYR1 ; IF NOT: PLAY ANOTHER ROUND.
0115 026B 20 CB 02 JSR BLINK ; PLAY PLUS BLINKING SIDE.
0116 026D A9 00 RTS ; ENDGAME! RETURN TO MONITOR.
0117 026F A9 0E LDA #$11110000 ; LIGHT LEFT SIDE OF BOTTOM.
0118 0271 BD 00 AO STA PORTIB
0119 0274 A9 00 STA PORTIA
0120 0276 BD 01 AO LDA #$0 ; CLEAR LOW LEDS.
0121 0279 A9 40 STA PORTIB
0122 027B 20 E3 02 JSR DELAY ; WAIT A WHILE TO SHOW WIN.
0123 027E EA 06 INC PLYR2 ; IPLAYER 2 HAS WON ANOTHER ROUND....
0124 0280 A9 0A LDA #$10 ; ...HAS HE WON 10?
0125 0282 C5 05 CMP PLYR2 ; IF NOT: PLAY ANOTHER ROUND.
0126 0284 50 BE BNE MOV ; IF NOT: PLAY ANOTHER ROUND.
0127 0286 A9 0E LDA #$111100 ; IF NOT: PLAY ANOTHER ROUND.
0128 0288 20 CB 02 JSR BLINK ; BLINKING PATTERN TO BLINK LEDS.
0129 028A 56 02 RTS ; END.
0130 028C 05
0131 028C 05 SUBROUTINE 'PLAY'
0132 028C 05 ; GETS TIME COUNT OF EACH PLAYER, AND IF
0133 028C 05 ; BAD GUESSES ARE MADE; THE PLAYER IS
0134 028C 05 ; GIVEN ANOTHER CHANCE; THE NEW TIME ADDED TO
0135 028C 05
0136 028C 05 THE OLD.
0137 028C 05
0138 028C 05 PLAY JSR RANDOM ; GET RANDOM NUMBER.
0139 028C 05 JSR DELAY ; RANDOM - DELAY DELAY.
013A 028C 05 JSR RANDOM ; GET ANOTHER.
013B 028C 05 AND #$1F ; KEEP UNDER 16 FOR USE AS
013C 028C 05 STA NUMBER ; NUMBER OF GUESSES.
013D 028C 05 ; USE AS INDEX TO TABLE.
013E 028C 05 LDA NUNTAB + X ; GET REVERSED PATTERN FROM TABLE.
013F 028C 05 PHA ; TO DISPLAY IN LEDS 12-15.
0140 028C 05 ORA PORTIB ; TO DISPLAY IN LEDS 12-15.
0141 028C 05 ; GET KEYS: SCRAMLE DURATION COUNT.
0142 028C 05 LDA CYRUS ; IS KEYS TROKKE SYSTEM CORRECT? instanceof.
0143 028C 05 ; GETS KEYS W/TIME OF KEYPRESS.
0144 028C 05 JSR CYRUS ; IF SO: DONE.
0145 028C 05 BE 00 AO BNE MOV ; DISPLAYForeignKey.
0146 028C 05 JSR CNTSUB ; GET OLD NUMBER.
0147 028C 05 CS 02 LDA CNTSUB ; DISPLAYForeignKey.
0148 028C 05 C4 07 BNE MOV ; NOT A DIFFERENT OLD NUMBER.
0149 028C 05 JSR KEYLP ; IF SO: DONE.
0150 028C 05 AB 00 AO BNE MOV ; NOT A DIFFERENT OLD NUMBER.
0151 028C 05 JSR PORTB ; IF SO: DONE.
0152 028C 05 1C 02 JMP PLAY ; TRY AGAIN W/ANOTHER NUMBER.
0153 028C 05 60 DEX DONE
0154 028C 05
0155 028C 05 SUBROUTINE 'COUNTER'
0156 028C 05 JSR KEYPRESS WHILE KEEPING TRACK OF AMT OF
0157 028C 05
0158 028C 05 ; TOTAL KEYS FROM EIGHT
0159 028C 05 JSR CNTSUB ; SET UP KEYS COUNTER.
0160 028C 05 BS 00 AC JSR KEYPRESS ; GET OUTPUT KEYS TO KEYBOARD MPX.
0161 028C 05 C4 07 JSR KEYLP ; GET NEXT KEY.
0162 028C 05 10 0B JSR PORTA ; KEY DOWN.
0163 028C 05 10 0B JSR PORTB ; KEY DOWN.
0164 028C 05 JSR KEYLP ; KEY DOWN.
0165 028C 05 10 F5 JSR PORTA ; KEY DOWN.
0166 028C 05 06 02 JSR PORTB ; KEY DOWN.
```

---

50
PSEUDO RANDOM NUMBER GENERATOR

0166 02C4 D0 EF  BNE CNTSUB  TRY KEYS AGAIN IF NO OVERFLOW.
0167 02C6 E6 01  INC CNTHI  OVERFLOW, INCREMENT HIGH BYTE.
0168 02CB D0 EB  BNE CNTSUB  TRY KEYS AGAIN.
0169 02CA 60  FINISH RTS  DONE TIME RUN OUT OR KEY PRESSED.
0170 02CB
0171 02CB  ISUBROUTINE 'BLINK'
0172 02CB  BLINKS LEDS WHOSE BITS ARE SET IN ACCUMULATOR
0173 02CB  ION ENTRY.
0174 02CB
0175 02CB A2 14  BLINK  LDX #20  120 BLINKS.
0176 02CD B6 01  STX CNTHI  SET BLINK COUNTER.
0177 02CF B5 02  STA CNTLO  BLINK REGISTER.
0178 02D1 A5 02  BLOOP  LDA CNTLO  BLINK PATTERN.
0179 02D3 40 00 AA  EOR PORTIB  BLINK LEDS.
0180 02D6 8D 00 A0  STA PORTIB
0181 02DF A9 0A  LDA #10  SHORT DELAY.
0182 02DB 20 E3 02  JSR DELAY
0183 02BE C6 01  DEC CNTHI
0184 02E0 D0 EF  BNE BLOOP  ILOOP IF NOT DONE.
0185 02E2 60  RTS
0186 02E3
0187 02E3  ISUBROUTINE 'DELAY'
0188 02E3  ICONTENTS OF REG. A DETERMINES DELAY LENGTH.
0189 02E3
0190 02E3 B5 00  DELAY  STA TEMP
0191 02E5 A0 10  DL1  LDY #10
0192 02E7 A2 FF  DL2  LDX #FF
0193 02E9 CA  DL3  DEX
0194 02EA D0 FD  BNE DL3
0195 02EC BB  BNE
0196 02ED D0 F8  BNE DL2
0197 02EF C6 00  DEC TEMP
0198 02F1 D0 F2  BNE DL1
0199 02F3 60  RTS
0200 02F4
0201 02F4  ISUBROUTINE 'RANDOM'
0202 02F4  IRANDOM NUMBER GENERATOR.
0203 02F4  IRETURN RANDOM NUMBER IN ACCUM.
0204 02F4
0205 02F4 38  RANDOM SEC
0206 02F5 A5 09  LDA SCR+1
0207 02F7 65 0C  ADC SCR+4
0208 02F9 65 0B  ADC SCR+5
0209 02FB B5 0B  STA SCR
0210 02FD A2 04  LDX #4
0211 02FF B5 0B  RNDLP  LDA SCR+X
0212 0301 95 09  STA SCR+1,X
0213 0303 CA  DEX
0214 0304 10 F9  BPL RNDLP
0215 0306 60  RTS
0216 0307  .END

SYMBOL TABLE

SYMBOL VALUE

BLINK 02CB  BLOOP  02D1  CNT1H  0003  CNT1L  0004
CNT1H  0001  CNT1L  0002  CNTSUB  02B5  DDR1A  0002
DDR1B  A002  DDK3A  AC03  DDK3B  AC02  DELAY  02E3
DL1  02E5  D2  02E7  DL3  02E9  DDNE  02B4
EQUAL  024A  FINISH  02CA  KEYP1  02B7  MOVE  0214
NUMBER  0007  NUMTAB  000E  PLAY  028C  PLR1  0252
PLR2  026F  PLREV  0005  PLTR2  0006  PORTIB  AC00
PORTIB  A000  PORT3A  AC01  PORT3B  AC00  RANDOM  02F4
RNDLP  02FF  SCR  0008  START  0200  TEMP  0000

END OF ASSEMBLY

Fig. 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:

```
LDA CNTHI
CMP CNT1H
BEQ EQUAL
BCC PLR2
BCS PLR1

EQUAL
LDA CNTLO
CMP CNT1L
BCC PLR2
CMP CNT1L
BCC PLR2
BCS PLR1
```

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
```
PSEUDO RANDOM NUMBER GENERATOR

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 #%1110
       STA PORT1B
       LDA #0
       STA PORT1A
       LDA #$40
       JSR DELAY
       INC PLYR2
       LDA #10
       CMP PLYR2
       BNE MOVE
       LDA #%1110
       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       Mask off high nibble
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:

```
TAX           Use X as index
LDA NUMTAB,X  Retrieve pattern
```

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:
PSEUDO RANDOM NUMBER GENERATOR

CPY NUMBER Correct guess?
BEQ DONE No: clear old guess
LDA #01
AND PORT1B
STA PORT1B
JMP PLAY Try again
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:

CNTSUB
LDY #$F
KEYLP
STY PORT3B
BIT PORT3A
BPL FINISH
DEY Count down key #
BPL KEYLP Next key
FINISH BNE CNTSUB

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

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
  STX CNTHI
  STA CNTLO
BLOOP
  LDA CNTLO
  EOR PORT1B
  STA PORT1B
  LDA #10
  JSR DELAY
  DEC CNTHI
  BNE BLOOP
  RTS
```

20 blinks
Blink counter
Blink register
Get blink pattern
Blink LEDs
Short delay
Loop if not done

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
DL1
  LDY #$10
DL2
  LDX #$FF
DL3
  DEX
  BNE DL3
  DEY
```
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. Hardware Random Number Generator (Hexguess)

INTRODUCTION

In this chapter random numbers will be generated using the timer’s latch on an input/output chip. More complex algorithms will be devised and simultaneous light and sound effects will be created.

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  
Next guess: “20”  
Computer’s response: 3  
We are going further away  
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
Fig. 4.1: Hexguess Flowchart
— 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
DUR and FREQ are the usual parameters required to generate a tone (TONE subroutine)
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
HARDWARE RANDOM NUMBER GENERATOR

; 'HEXGUESS'
; HEXADECIMAL NUMBER GUESSING GAME.
; THE OBJECT OF THE GAME IS TO GUESS A HEXADECIMAL
; NUMBER THAT THE COMPUTER HAS THOUGHT UP.
; WHEN THE COMPUTER 'BEEPS', A GUESS SHOULD
; BE ENTERED. GUESSES ARE TWO DIGIT HEXADECIMAL.
; NUMBERS. WHEN TWO DIGITS HAVE BEEN RECEIVED,
; THE COMPUTER WILL DISPLAY THE NEARNESS
; OF THE GUESS BY LIGHTING A NUMBER OF
; LEDs PROPORTIONAL TO THE CLOSENESS OF
; THE GUESS. TEN GUESSES ARE ALLOWED.
; IF A GUESS IS CORRECT, THEN THE COMPUTER
; WILL FLASH THE LEDS AND MAKE A WARBLING
; TONE.
; THE ENTRY LOCATION IS $200.
;
; GETKEY = $100
; #5222 VIA #1 ADDRESSES:
; TIMER = $A004  LOW LATCH OF TIMER 1
; DDR1A = $A003  PORTA DATA DIRECTION REG.
; DDR1B = $A002  PORTB DATA DIRECTION REG.
; PORTIA = $A001  PORT A
; PORTIB = $A000  PORT B
; #5222 VIA #3 ADDRESSES:
; DDR3B = $AC02  PORTB DATA DIRECTION REG.
; PORT3B = $AC00  PORT B
; @STORAGE:
; GUESS = $00
; GUESS$ = $01
; NUM = $02
; FREQ = $03
; NUMBER = $04
;
; . = $200
0200: A9 FF  LDA $FF  ;SET UP DATA DIRECTION REGISTERS
0202: BD 03 A0  STA DDR1A
0205: BD 02 A0  STA DDR1B
0208: BD 02 AC  STA DDR3B
020B: B5 02  STA DUR  ;SET UP TONE DURATIONS.
020D: A9 0A  START LDA $0A  ;10 GUESSES ALLOWED
020F: B5 01  STA GUESS$  ;BLANK LEDs
0211: A9 00  LDA $00  ;START NUMBER
0213: BD 01 A0  STA PORTIA
0216: BD 00 A0  STA PORTIB
0219: AB 04 A0  LDA TIMER  ;GET RANDOM NUMBER TO GUESS
021B: 85 04  STA NUMBER  ;...AND SAVE.
021E: A9 20  GETGUES LDA $120  ;GET RANDOM NUMBER TO GUESS
0220: 20 96 02  JSR TONE  ;MAKE BEEP.
0223: 20 00 01  JSR GETKEY  ;GET HIGH ORDER USER GUESS
0226: 0A  ASL A  ;SHIFT INTO HIGH ORDER POSITION
0227: 0A  ASL A
0229: 0A  ASL A
022A: 05 00  STA GUESS  ;SAVE
022C: 20 00 01  JSR GETKEY  ;GET LOW ORDER USER GUESS
022F: 29 0F  AND $20000111  ;MASK HIGH ORDER BITS.
0231: 05 00  ORA GUESS  ;ADD HIGH ORDER NIBBLE.
0233: 85 00  STA GUESS  ;FINAL PRODUCT SAVED.
0235: A5 04  LDA NUMBER  ;GET NUMBER FOR COMPARE
0237: 38  SEC
0238: E5 00  SBC GUESS  ;SUBTRACT GUESS FROM NUMBER
023A: B0 05  BCS ALRIGH  ;DETERMINE NEARNESS OF GUESS.
023C: 49 FF  EOR $21111111  ;MAKE DISTANCE ABSOLUTE
023E: 38  SEC  ;MAKE IT A TWO'S COMPLEMENT
023F: 69 00  ADC $00  ;...NOT JUST A ONES COMPLEMENT.

---Fig. 4.2: Hexguess Program---
ADVANCED 6502 PROGRAMMING

0241: A2 00 ALRIGHT LDX #00
0243: DD AD 02 LOOP CMP LIMITS,x
0246: B0 27 BCS SIGNAL
0248: E8 INX
0249: E0 09 CPX #9
024B: DD F6 BNE LOOP
024D: A9 08 WIN LDA #11
024F: B5 00 STA GUESS
0251: A9 FF LDA $FF
0253: B0 01 A0 STA PORT1A
0256: BD 00 A0 STA PORT1B
0259: A6 32 WOW LDA #$50
025B: 20 96 02 JSR TONE
025E: A9 FF LDA #$FF
0260: 4D 01 A0 EOR PORT1A
0263: BD 01 A0 STA PORT1A
0266: DD 00 A0 STA PORT1B
0269: C6 00 DEC GUESS
026B: DD EC BNE WOW
026D: 0D 9E BRK
026F: EB SIGNAL INC
0270: A9 00 LDA #0
0273: BD 00 A0 STA PORT1B
0275: 20 06 02 JSR LITE
0278: BD 01 A0 STA PORT1A
027B: 90 05 BCC CC
027D: A9 01 LDA #01
027F: BD 00 A0 STA PORT1B
0282: C7 00 CC DEC GUESS
0285: B0 9B BNE GUESSES
0288: A9 BE LDA #$BE
028B: 20 96 02 JSR TONE
028D: 4C 00 02 JMP START

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

028F: A9 00 LITE LDA #0
0291: 38 SHIFT SEC
0292: 2D ROL A
0293: 21 DEX
0294: B0 FB BNE SHIFT
0295: 60 RTS

; TONE GENERATION ROUTINE.

0296: B5 03 TONE STA FRED
0298: A9 00 LDA #$00
029A: A6 02 LDX DUR
029C: A4 03 FL2 LDY FREQ
029E: 00 FL1 DEF
029F: 1B CLC
02A0: 90 00 BCC .+2
02A2: D0 FA BNE FL1
02A4: A9 FF EOR #$FF
02A7: BD 00 AC STA PORT3B
02A9: CA DEX
02AB: BD FO BNE FL2
02AC: 60 RTS

; TABLE OF LIMITS FOR CLOSENESS LEVELS:

Fig. 4.2: Hexguess Program (Continued)

64
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETKEY</td>
<td>0100</td>
</tr>
<tr>
<td>DBR1B</td>
<td>A002</td>
</tr>
<tr>
<td>DBR3B</td>
<td>A000</td>
</tr>
<tr>
<td>GUESS#</td>
<td>0001</td>
</tr>
<tr>
<td>NUMBER</td>
<td>0004</td>
</tr>
<tr>
<td>ALRIGHT</td>
<td>0241</td>
</tr>
<tr>
<td>WOW</td>
<td>0259</td>
</tr>
<tr>
<td>LITE</td>
<td>028E</td>
</tr>
<tr>
<td>FL2</td>
<td>029C</td>
</tr>
<tr>
<td>LIMITS</td>
<td>BYTE 200,128,64,32,16,8,4,2,1</td>
</tr>
</tbody>
</table>

---

**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:

```plaintext
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 T1CL 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 refreshed 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
memory location DUR while the note frequency is set by the contents of the accumulator:

GETGES LDA #$20
          JSR TONE

High pitch

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.)
HARDWARE RANDOM NUMBER GENERATOR

Fig. 4.4: Collecting the Player's Guess

JSR GETKEY
ASL A
ASL A
ASL A
ASL A
STA GUESS
JSR GETKEY

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:

STA GUESS
ADVANCED 6502 PROGRAMMING

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  Positive?
EOR #\%11111111  It is negative: complement
SEC         Make it two's complement
ADC #00     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 #00
CMP LIMITS,X   Look up limit value
BCS SIGNAL
INX            Closeness is less
CPX #9         Keep trying 10 times
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     Scratch storage
LDA #FF
```

68
HARDWARE RANDOM NUMBER GENERATOR

STA PORT1A
STA PORT1B

WOW
LDA #50       Tone pitch
JSR TONE      Generate tone

The blinking is generated by complementing the LEDs repeatedly:

LDA #$FF
EOR PORT1A   Complement ports
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       Increment closeness level
         LDA #0     Clear high LED port
         STA PORT1B
         JSR LITE   Get LED pattern
         STA PORT1A
         BCC CC
         LDA #01    If carry set, PB0 = 1
         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-
plicit 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

![Diagram of LED patterns with labels: JUST BEFORE 1st ROTATION, BEFORE 2nd ROTATION, BEFORE 3rd ROTATION, BEFORE 8th ROTATION (CARRY WILL BE 0), AFTER 9th ROTATION (CARRY IS 1)].

**Fig. 4.5:** Obtaining the LED pattern for 8 LED's
new game is started; otherwise, the next guess is obtained:

```
CC    DEC GUESS#
BNE GETGES   Any guesses left?
LDA #$BE   Low tone
JSR TONE   New game
JMP START
```

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:

```
LITE SHIFT
LDA #0
SEC
ROL A
DEX
BNE SHIFT
RTS
```

**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:

```
 TONE   STA FREQ
       LDA #$00
       LDX DUR
 FL2   LDY FREQ
 FL1   DEY
```
CLC
BCC . +2
BNE FL1
EOR #$FF
STA PORT3B
DEX
BNE
RTS

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. Simultaneous Input/Output
(Magic Square)

INTRODUCTION

Special visual patterns will be created by this program. Random numbers will be generated by the hardware source, the timer. Delays, blinkers, and counters will be used.

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.

![Diagram of LED pattern]

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

73
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.

Likewise, key #8 will complement the pattern formed by LEDs 7, 8, and 9, and key #6 will complement the pattern formed by LEDs 3, 6, and 9.
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:

```
  O  O  O
  O  O  *
  O  *  O
```

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

```
  O  *  O
  *  *  O
  O  O  O
```

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

```
  *  *  *
  *  O  *
  *  *  *
```
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 #2.
The resulting pattern is: 2-4-6-7-8.
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 #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.
SIMULTANEOUS INPUT/OUTPUT

Fig. 5.1: Magic Square Flowchart
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, 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, 0, 0, 0, 0, 1, 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>01001001</td>
</tr>
<tr>
<td>5</td>
<td>10111010</td>
</tr>
<tr>
<td>6</td>
<td>00100100</td>
</tr>
<tr>
<td>7</td>
<td>11011000</td>
</tr>
<tr>
<td>8</td>
<td>11000000</td>
</tr>
<tr>
<td>9</td>
<td>10110000</td>
</tr>
</tbody>
</table>

**Fig. 5.2: Complementation 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.
SIMULTANEOUS INPUT/OUTPUT

; 'MAGIC SQUARE' PROGRAM
; KEYS 1-9 ON THE HEX KEYBOARD ARE EACH ASSOCIATED
; WITH ONE LED IN THE 3X3 ARRAY. WHEN A KEY IS PRESSED,
; IT CHANGES THE PATTERN OF THE LIT LEDS IN THE ARRAY.
; THE OBJECT OF THE GAME IS TO CONVERT THE RANDOM
; PATTERN THE GAME STARTS WITH TO A SQUARE OF LIT
; LEDS BY PESSING THE KEYS. THE Leds WILL FLASH WHEN
; THE WINNING PATTERN IS ACHIEVED.
; KEY $00 CAN BE USED AT ANY TIME TO RESTART
; THE GAME WITH A NEW PATTERN.
;
; GETKEY =#100
; TICL =#A004   ; LOW REGISTER OF TIMER IN 6522 VIA
; PORT1 =#A001   ; 6522 VIA PORT A
; PORT2 =#A000   ; 6522 VIA PORT B
; TEMP =#0000   ; TEMPORARY STORAGE
; DDRA =#A003   ; DATA DIRECTION REGISTER OF PORT A
; DDRB =#A002   ; SAME FOR PORT B

; ; COMMENTS: THIS PROGRAM USES A TIMER REGISTER FOR A
; ; RANDOM NUMBER SOURCE. IF NONE IS AVAILABLE, A
; ; RANDOM NUMBER GENERATOR COULD BE USED, BUT
; ; DUE TO ITS REPEATABILITY, IT WOULD NOT WORK AS
; ; WELL. THIS PROGRAM USES PORT A'S REGISTERS FOR
; ; STORAGE OF THE LED PATTERN. SINCE WHAT IS READ
; ; BY THE PROCESSOR IS THE POLARITY OF THE
; ; OUTPUT LINES, AN EXCESSIVE LOAD ON THE LINES WOULD
; ; PREVENT THE PROGRAM FROM WORKING CORRECTLY.
;
; 0200:   A9 FF   ; LDA #$FF   ; SET UP PORTS FOR OUTPUT
; 0201:   BD 03 A0   ; STA DDRA
; 0202:   BD 02 A0   ; STA DDRB
; 0203:   AD 04 A0   ; START LDA TICL   ; GET 1ST RANDOM NUMBER
; 0204:   BD 01 A0   ; STA PORT1
; 0205:   AD 04 A0   ; LDA TICL   ; ...AND SECOND
; 0206:   29 01   ; AND #$01   ; MASK OUT BOTTOM ROW LEDs
; 0207:   BD 00 A0   ; STA PORT2
; 0208:   20 00 01   ; KEY JSR GETKEY
; 0209:   C9 00   ; CMP #$0   ; KEY MUST BE 1-9; IS IT 0?
; 020A:   F0 05   ; BNE START   ; YES, RESTART GAME WITH NEW BOARD.
; 020B:   C9 0A   ; CMP #$10   ; IS IT LESS THAN 10?
; 020C:   1F 05   ; BFE KEY   ; IF KEY >=10, DO GET ANOTHER

; ; FOLLOWING SECTION USES KEY NUMBER AS INDEX TO FIND IN
; ; TABLE A BIT PATTERN USED TO COMPLEMENT LED'S
;
; 0211:   38   ; SEC   ; DECREMENT A FOR TABLE ACCESS
; 0212:   E9 01   ; SBC #$1
; 0213:   0A   ; ASL A   ; MULTIPLY A*2, SINCE EACH ENTRY IN
; 0214:   25   ; TAX   ; TABLE IS TWO BYTES.
; 0215:   AA   ; Fuse A AS INDEX
; 0216:   AD 01 A0   ; LDA PORT1   ; GET PORT CONTENTS FOR COMPLEMENT
; 0217:   5D 0A 02   ; EOR TABLE,X   ; EOR PORT CONTENTS W/PATTERN
; 0218:   BD 01 A0   ; STA PORT1   ; RESTORE PORT1
; 0219:   AD 00 A0   ; LDA PORT2   ; IDO SAME WITH PORT2,
; 021A:   5D 0C 02   ; EOR TABLE+1,X   ; USING NEXT TABLE ENTRY.
; 021B:   29 01   ; AND #$01   ; MASK OUT BOTTOM ROW LEDS
; 021C:   BD 00 A0   ; STA PORT2   ; ...AND RESTORE.

; ; THIS SECTION CHECKS FOR WINNING PATTERN IN LEDs
;
; 0231:   4A   ; LSR A   ; SHIFT BIT 0 OF PORT 1 INTO CARRY.
; 0232:   90 D9   ; BEC KEY   ; IF NOT WIN PATTERN, GET NEXT MOVE
; 0233:   AD 01 A0   ; LDA PORT1   ; LOAD PORT FOR WIN TEST
; 0234:   C9 EF   ; CMP #$11101111   ; CHECK FOR WIN PATTERN
; 0235:   D0 B2   ; BNE KEY   ; NO WIN, GET NEXT MOVE

---

Fig. 5.3: Magic Square Program
; WIN BLINK LED’S EVERY 1/2 SEC, 4 TIMES

0244: A9 0E            LDA $14
0246: B5 00            STA TEMP
024B: A2 20            BLINK
024A: A0 FF            DELAY
024C: EA DLY NOP
024D: B0 00            BNE .42
024F: B8 DEX
0250: B0 FA            BNE DLY
0252: CA DEX
0253: D0 F5            BNE DELAY
0255: AD 01 A0         LDA PORT1
0258: 49 FF            EOR $$FF
025A: BD 01 A0         STA PORT1
025D: AD 00 A0         LDA PORT2
0260: 49 01            EOR $1
0262: BD 00 A0         STA PORT2
0265: C6 00            DEC TEMP
0267: D0 DF            BNE BLINK
0269: F0 AB            BEQ KEY

; TABLE OF CODES USED TO COMPLEMENT LEDS

026B: 1B TABLE .BYT $00011011,$00000000
026D: 00
026F: 07 .BYT $00000111,$00000000
0271: 3F .BYT $00110110,$00000000
0275: 49 .BYT $00010100,$00000000
0277: 00 .BYT $10111101,$00000000
0279: 00 .BYT $10100010,$00000000
027B: 00 .BYT $11000000,$00000000
027C: 01 .BYT $10110000,$00000000

SYMBOL TABLE:
GETKEY 0100 TICL A004 PORT1 A001
PORT2 A000 TEMP 0000 DDRA A003
DDRB A002 START 0208 KEY 0216
BLINK 0248 DELAY 024A DLY 024C
TABLE 0268

Fig. 5.3: Magic Square Program (Continued)
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 DDRB

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.)

<table>
<thead>
<tr>
<th>START</th>
<th>LDA T1CL</th>
<th>Get 1st number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>STA PORT1</td>
<td>Use it</td>
</tr>
<tr>
<td></td>
<td>LDA T1CL</td>
<td>Get 2nd number</td>
</tr>
<tr>
<td></td>
<td>AND #01</td>
<td>Keep only position 0</td>
</tr>
<tr>
<td></td>
<td>STA PORT2</td>
<td>Use it</td>
</tr>
</tbody>
</table>

An explanation of the use of T1CL 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:

<table>
<thead>
<tr>
<th>KEY</th>
<th>JSR GETKEY</th>
<th>Is key 0?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CMP #0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEQ START</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CMP #10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BPL KEY</td>
<td>If key = 10 get another</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th></th>
<th>SBC #1</th>
<th>Subtract 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC</td>
<td>ASL A</td>
<td>Multiply by 2</td>
</tr>
</tbody>
</table>
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:

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:

```
LDA PORT1
EOR TABLE,X     Complement Port1
STA PORT1
LDA PORT2       Same for Port2
EOR TABLE + 1,X
AND #01         Mask out unused bits
STA PORT2
```

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

```
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:

```
LSR A          Shift bit 0 of Port 2
BCC KEY
```

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

```
LDA PORT1
CMP #%11101111
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.

```
LDA #14
STA TEMP
BLINK
LDX #$20
DELAY
LDY #$FF

DLY
NOP
BNE .+2
DEY
BNE DLY
DEX
BNE DELAY
LDA PORT1

EOR #$FF
STA PORT1
LDA PORT2
EOR #1
STA PORT2
DEC TEMP
BNE BLINK
BEQ KEY

Get ports and complement them

Count down number of blinks

Do again if not done

Get next 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. Simple Real Time Simulation (Spinner)

INTRODUCTION

This program will react in real time to an operator input. The game will operate at multiple levels of difficulty using more complex loop counters.

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 “000000001, or 01 hexadecmal, must be sent. For LED #2, the bit pattern “00000010” must be sent, or 02 hexadecmal. 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:

```
DURAT      Is the delay between two successive LED illuminations
DIFCLT     Is the “difficulty level” (reversed)
DNTST      Is a flag used to detect an illegal key closure when scanning the keys
```

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

LDA #8
STA DFCLT

The keystore 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 Y contains counter
AND #$07 Extract lower 3 bits
TAX Use as index

The pattern is obtained from LTABLE, using an indexed addressing
SIMPLE REAL TIME SIMULATION

LINE # LOC  CODE  LINE
0002  0000  ; 'SPINNER'
0003  0000  ; PROGRAM TO TEST REACTION TIME OF PLAYER.
0004  0000  ; FLIP OF LIGHT SPINS AROUND EDGE
0005  0000  ; FOR 3X3 LED MATRIX, AND USER MUST PRESS
0006  0000  ; CORRESPONDING KEY, IF, AFTER A NUMBER OF
0007  0000  ; SPINS, CORRECT KEY HAS NOT BEEN PRESSED,
0008  0000  ; BLIP SPINS SLOWER, IF CORRECT KEY HAS BEEN
0009  0000  ; I/PRESSED, BLIP SPINS FASTER. ALL
0010  0000  ; LEDS LIGHT WHEN SUCCESSFUL KEYPRESS
0011  0000  ; OCCURS ON MAXIMUM SPEED.
0012  0000  ;
0013  0000  ; I/O :
0014  0000  ;
0015  0000  ; PORTA = $A001  ; LEDS 1-8
0016  0000  ; PORTB = $A000  ; LEDS 8-15
0017  0000  ; DDRA = $A000
0018  0000  ; DDRB = $A002
0019  0000  ; PORTA = $A001  ; KEY STORE INPUT.
0020  0000  ; PORTB = $A000  ; KEY # OUTPUT.
0021  0000  ; DDRA = $A000
0022  0000  ; DDRB = $A002
0023  0000  ;
0024  0000  ; VARIABLE STORAGE:
0025  0000  ;
0026  0000  ; * = $0
0027  0000  ;
0028  0000  ; DURAT =##1  ; DURATION OF INTER-MOVEMENT DELAY.
0029  0001  ; DTC =##1  ; DIFFICULTY LEVEL.
0030  0002  ; DMTST =##1  ; FSET TO $01 IF KEY DOWN AT START
0031  0003  ; OF INTER-MOVEMENT DELAY.
0032  0003  ;
0033  0003  ; TABLE OF PATTERNS TO BE SENT TO LED
0034  0003  ; MATRIX AT EACH LOOP COUNT.
0035  0003  ; FSET FOR CLOCKWISE ROTATION STARTING AT LED #1.
0036  0003  ;
0037  0003  ; LTABLE .BYTE $01,$02,$04,$20,$00,$80,$40,$08
0038  0004  ;
0039  0004  ; TABLE OF PATTERNS TO BE SENT TO KEYBOARD
0040  0004  ; FTO TEST IF LEDS ARE ON AT EACH LOOP COUNT.
0041  0004  ;
0042  0008  ; KYTEBL .BYTE 1,2,3,6,9,8,7,4
0043  0008  ;
0044  0013  ; MAIN PROGRAM
0045  0013  ;
0046  0013  ; * = $200
0047  0200  ; START LDA #$FF  ; SET I/O REGISTERS.
0048  0200  ; AP FF
0049  0202  ; BD 03 00  ; STA DDRA
0050  0205  ; BD 02 00  ; STA DDRB
0051  0208  ; BD 02 0C  ; STA DDR3
0052  020B  ; AP 08  ; LDA #8
0053  020D  ; BS 01  ; STA DTC
0054  020F  ; BD 03 AC  ; STA DDR3
0055  0212  ; D0 00  ; NUMGE LY #0
0056  0214  ; BS 00  ; LOOP LDA #0
0057  0216  ; BS 02  ; STA DMTST
0058  0218  ; BD 00 0A  ; STA PORTB
0059  021B  ; 9B  ; TYA
0060  021C  ; 29 07  ; AND #$07  ; AS INDEX TO FIND LED PATTER
0061  021E  ; AA  ; TAX  ; FIN TABLE OF PATTERNS.
0062  021F  ; BS 03  ; LDA LTABLE+X  ; FSET PATTERN FOR LED TO

Fig. 6.3: Spinner Program

93
0043 0221 JBE TURNED ON.
0064 0221 BD 01 A0 STA PORTA1B.
0065 0222 BD 01 D0 D0 CHECK BUSC DRAG < 0, SKIP.
0066 0226 A9 01 LDA #1.
0067 0228 BD 00 A0 STA PORT1B.
0068 0228 BD 05 08 CHECK LDA KYTLK.
0069 0230 BD 00 AC STA PORT1B.
0070 0230 2C 01 A0 BIT PORTA3.
0071 0233 30 04 BMI DELAY.
0072 0235 A9 01 INVALID.
0073 0237 BD 05 02 STA DNTST.
0074 0239 A9 B0 DELAY LDA #XXX.
0075 023B BD 00 00 STA DURAT.
0076 023B A9 01 DL1 LDA DIFCLT.
0077 023F 0A ASL A.
0078 0240 0A ASL A.
0079 0241 AA TAX.
0080 0242 2D 02 DL2 ROL DNTST.
0081 0244 66 02 ROR DNTST.
0082 0246 CA DEX.
0083 0247 BD 09 BNE DL2.
0084 0249 A5 02 LDA DNTST.
0085 024B D0 00 BNE NOTST.
0086 024D 0C A0 DELAY.
0087 024D BD 01 AC BIT PORTA3.
0088 0250 1D 19 BFL HIT.
0089 0252 BD 00 00 NOTST.
0090 0254 EB 07 BNE DL1.
0091 0256 CB 01 INY.
0092 0257 D0 BB BNE LOOP.
0093 0259 A6 01 DLX DIFCLT.
0094 025B EB INX.
0095 025C 0A TXX.
0096 025D 09 10 CMP #16.
0097 025F 0D 02 BNE OK.
0098 0261 A9 0F LDA #15.
0099 0263 BD 05 01 STA DIFCLT.
0100 0265 20 BD 02 JSR WAIT.
0101 0266 4C 12 02 JMP NMGME.
0102 0268 2D 00 00 JMP WAIT.
0103 0269 20 BD 00 HIT.
0104 026A 2D 00 DIFCLT.
0105 026B 00 A0 BNE NMGME.
0106 026C 1D 19 PLA YER MADE IT TO TOP.
0107 026D BD 01 A0 STA PORTA1B.
0108 026E BD 00 A0 STA PORT1B.
0109 026F 2D 00 JSR WAIT.
0110 0270 4C 00 02 JMP START.
0111 0271 4C 02 JSR START.
0112 0272 00 "SUBROUTINE 'WAIT'."
0113 0273 00 ISHORT DELAY.
0114 0274 BD 00 BD 00 "SUBROUTINE 'WAIT'."
0115 0275 BD 00 BD 00 "SUBROUTINE 'WAIT'."
0116 0280 A0 FF WAIT LDX #FF.
0117 0282 A2 FF LX DIFCLT.
0118 0284 66 00 LP2 ROR DURAT.
0119 0286 26 00 ROL DURAT.
0120 0288 66 00 ROR DURAT.
0121 028A 26 00 ROL DURAT.
0122 028C 0A CA DEX.
0123 028D D0 05 BNE LP2.
0124 028F BB DEY.
0125 0290 D0 00 BNE LP2.
0126 0292 60 RTS.
0127 0293 .END.

SYMBOL TABLE

CHECK 0228.

DRL9A A003.

DDR18 A002.

DDR3A A001.

DRL2 A024.

DNTST 0002.

DURAT 0000.

HIT 026B.

INVALID 0235.

KYTLK 0008.

LOP 0214.

LP1 0282.

DRL2 0242.

DNTST 0002.

DURAT 0000.

HIT 026B.

END OF ASSEMBLY.

Fig. 6.3: Spinner Program (Continued)
mechanism with register X, and this pattern is output on Port 1A to light up the appropriate LED:

```
LDA LTABLE, X  Get pattern
STA PORT1A      Use it to light up LED
```

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       Was pattern = 0?
LDA #1          If not, set LED #9
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 KYTBL,X   X contains correct pointer
                STA PORT 3B  Select correct key
                BIT PORT 3A  Strobe hi?
                BMI DELAY    If not, skip
```

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

```
INVALID        LDA #01
                STA DNTST
```

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:
DL1
LDA DIFCLT
ASL A
ASL A
TAX

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:

DL2
ROL DNTST
ROR DNTST
DEX
BNE DL2 Loop til count = 0

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 Check key strobe
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 Loop if not 0
INY Increment counter
BNE LOOP 32 loops?
Once the Y register has overflowed, i.e., 32 loops have been executed, the difficulty value is increased, resulting in a slower spin:

```assembly
LDX DIFCLT
INX
```

No hits. Make it easier

The maximum difficulty level is 15, and this is tested explicitly:

```assembly
TXA
CMP #16
BNE OK
LDA #15
OK
STA DIFCLT
```

Only A may be compared

Stay at 15 maximum

Finally, a brief pause is implemented:

```assembly
JSR WAIT
```

and a new spin is started:

```assembly
JMP NWGME
```

In the case of a hit, a pause is also implemented:

```assembly
HIT
JSR WAIT
```

then the game is made harder by decrementing the difficulty count (DIFCLT)

```assembly
DEC DIFCLT
```

The difficulty value is tested for "0" (fastest possible spin). If the "0" level has been reached, the player has won the game and all LEDs are illuminated:

```assembly
BNE NWGME
LDA #$FF
STA PORT1A
STA PORT1B
```

If not 0, play next game

It is a win

Light up

The usual pause is implemented, and a new game is started:
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. Real Time Simulation
(Slot Machine)

INTRODUCTION

This program simulates an actual electro-mechanical machine and operates in real time. It performs a complex score evaluation using indexed addressing techniques as well as special data structures to facilitate and expedite the process.

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 ma-
chines). 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, indi-
cating 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 spin-
ing. 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 (LOLIM, HILIM). 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
Fig. 7.3: Slots Flowchart
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
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
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:

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

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 \( \text{SCORTMP} \).
13. At this point, the score computed above (\( \text{SCORTMP} \)) is examined by the program, and two possibilities exist: if the \( \text{SCORTMP} \) is 0, a branch occurs to 20, where the game score is decremented. If \( \text{SCORTMP} \) is not 0, the game score will be increased by the score for this spin — \( \text{SCORTMP} \). 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’ed 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
REAL TIME SIMULATION

LINE  LOC  CODE  LINE
0002  0000  1  SLOT MACHINE SIMULATOR PROGRAM.
0003  0000  1  PRESS ANY KEY TO START 'SPIN'.
0004  0000  1  SCORE DETERMINED BY ARRAY 'SCOREB'.
0005  0000  1  #8 POINTS INITIAL SCORE; ONE PENALTY
0006  0000  1  #8 FOR EACH BAD SPIN.
0007  0000  1  
0008  0000  1  X = $0
0009  0000  1  TEMP  ***#1  TEMPORARY STORAGE.
0010  0001  1  SCORPT  ***#1  TEMPORARY SCORE STORAGE.
0011  0002  1  SCORE  ***#1  ISCORE.
0012  0003  1  DUR  ***#1  DSUM (SUM OF DURATION OF TONES).
0013  0004  1  FREQ  ***#1  DFREQUENCY OF TONES.
0014  0005  1  SPEEDS  ***#3  FSPEEDS OF REVOLUTION FOR LEDS
0015  0006  1  # IN COLUMNS
0016  0006  1  INDX  ***#3  IDelay COUNTERS FOR LED REVOLUTIONS.
0017  0006  1  INCR  ***#3  IPOINTERS FOR LED POSITIONS.
0018  0006  1  LTHB  ***#3  IUSED TO FETCH PATTERNS OUT OF TABLES.
0019  0006  1  VALUES  ***#3  IPOINTERS FOR LIT LEDS.
0020  0009  1  RND  ***#6  ISCRATCHPAD FOR RND @ GEN.
0021  0009  1  LTBK  ***#3  IPATTERNS FOR LIT LEDS AND  
0022  0009  1  VALUES  ***#3  IND. OF LIT LEDS IN EACH ROW.
0023  0009  1  IT0  
0024  0009  1  1/I
0025  000A  1  PORTI A = $A001  I VIAA1 PORT A I/O REG (LEDS)
0026  000A  1  DDRI A = $A003  I VIAA1 PORT A DATA DIRECTION REG.
0027  000A  1  PORTIB = $A000  I VIAA1 PORT B I/O REG. (LEDS)
0028  000A  1  DDRIB = $A002  I VIAA1 PORT B DATA DIRECTION REG.
0029  000A  1  PORT3B = $AC00  I VIA#3 PORT B I/O REG. (SPKR)
0030  000A  1  DDR3B = $AC02  I VIA#3 PORT B DATA DIRECTION REG.
0031  000A  1  TICL = $A004  I TICL
0032  000A  1  IARRAYS
0033  000A  1  IARRAYS
0034  000A  1  IARRAY OF PATTERNS TO LIGHT LEDS.
0035  000A  1  IARRAY ROWS CORRESPOND TO COLUMNS OF LED
0036  000A  1  IARRAY, AND COLUMNS TO ROWS. FOR EXAMPLE, THIRD
0037  000A  1  BYTE 8 ROW ONE WILL LIGHT LED 7.
0038  000A  1  LTABLE .BYTE 1*8,64
0039  000A  1  #8
0040  000A  1  01
0041  000A  1  08
0042  000A  1  04
0043  000A  1  #1
0044  000A  1  02
0045  000A  1  02
0046  000A  1  02
0047  000A  1  02
0048  000A  1  02
0049  000A  1  02
0050  000A  1  02
0051  000A  1  02
0052  000A  1  02
0053  000A  1  02
0054  000A  1  00
0055  000A  1  00
0056  000A  1  00
0057  000A  1  A9 FF  LDA #FF  SET UP PORTS.

Fig. 7.9: Slot Machine Program
0057 0202 8D 03 A0 STA DDR1A
0058 0205 8D 02 A0 STA DDR1B
0059 0208 8D 02 AC STA DDR3B
0060 020B AD 04 A0 LDA TICL
0061 020E 85 15 STA RAND+1
0062 0210 A9 08 START LDA #8 ; INITIAL SCORE IS EIGHT.
0063 0212 B5 02 STA SCORE
0064 0214 46 TAY ; SHOW INITIAL SCORE
0065 0215 20 30 03 JSR LIGHT
0066 0218 20 00 01 KEY JSR GETKEY ; ANY KEY PRESSED STARTS PROGRAM.
0067 021B 20 27 02 JSR DISPLAY ; SPIN WHEELS
0068 021E 20 0A 02 JSR EVAL ; CHECK SCORE AND SHOW IT
0069 0221 A5 02 LDA SCORE
0070 0223 D0 F3 BNE KEY ; IF SCORE <> 0, GET NEXT PLAY.
0071 0225 F0 E9 BEQ START ; IF SCORE = 0, RESTART.
0072 0227
0073 0227 ISUBROUTINE TO DISPLAY "SPINNING" LEDS;
0074 0227 IF FIND COMBINATION TO USED TO DETERMINE SCORE.
0075 0227 ;
0076 0227 LOLIM = 90
0077 0227 HILIM = 135
0078 0227 SFDPM = 80
0079 0227 SPIN
0080 0227 A9 00 DISPLAY LDA #0 ; IRESET POINTERS.
0081 0229 B5 09 STA INCR
0082 0228 B5 0C STA INCR+1
0083 0222 B5 0B STA INCR+2
0084 0222 A0 02 LDNY #2 ; SET INDEX FOR 3 ITERATIONS.
0085 0231 20 B0 03 GETRD JSR RANDOM ; GET RANDOM #.
0086 0234 C9 87 CMP #HILIM ; ITOO LARGE?
0087 0236 B0 F9 BSX GETRD ; IF SO, GET ANOTHER.
0088 0238 C9 5A CMP #LOLIM ; ITOO SMALL?
0089 023A 90 F5 BCC GETRD ; IF SO, GET ANOTHER.
0090 023C 99 08 00 STA INDX,Y ; SAVE IN LOOP INDEXES AND
0091 023F 99 05 00 STA SPEEDS,X ; LOOP SPEED COUNTERS.
0092 0242 B8 DEY
0093 0243 10 EC BPL GETRD ; GET NEXT RND #.
0094 0245 A2 02 UPDATE LDX #2 ; SET INDEX FOR THREE ITERATIONS.
0095 0237 B4 05 UDTP RLD SPEEDS,X ; II SPEED(X)=0?
0096 0249 F0 44 BEQ NXUPD ; IF SO, DO NEXT UPDATE.
0097 024B D6 08 DEC INDX,Y ; DEC DECREMENT LOOP INDEX(X)
0098 024D D0 40 BNE NXUPD ; IF LOOPINDEX(X) <> 0.
0099 024F B4 0B LBY INCR,X ; INCREMENT POINTER(X).
0100 0251 CB INY
0101 0232 0D 03 CPY #3 ; IFPOINTER = 3?
0102 0254 D0 02 BNE NORST ; IF NOT SKIP...
0103 0256 A0 00 LBY #0 ; ...RESET OF POINTER TO 0.
0104 0258 94 0B NORST STY INCR,X ; IRESTORE POINTER(X).
0105 025A 86 00 STX TEMP ; IMULTIPLY X BY 3 FOR ARRAY ACCESS.
0106 025C 8A TXA
0107 025D 8A ASL A
0108 025E 1B CLC
0109 025F 55 00 ADC TEMP
0110 0261 75 0B ADC INCR,X ; ADD COLUMN# TO PTR(X) FOR ROW.
0111 0264 26 0B TAY ; FXFER TO Y FOR INDEXING.
0112 0264 B9 1A 00 LDA TABLE,Y ; IGET PATTERN FOR LED.
0113 0267 95 0E STA LTMSK,X ; STORE IN LIGHT MASK(X).
0114 0269 B4 05 SPDUP RLD SPEEDS,X ; INCREMENT SPEED(X).
0115 026B CB INY
0116 026C 94 05 STY SPEEDS,X ; IRESTORE.
0117 026D 94 0B STY INDX,Y ; IRESET LOOP INDEX(X).
0118 0270 A9 00 LEDUP LDA #0 ; UPDATE LIGHTS.
0119 0272 BD 00 A0 STA PORTIB ; IRESET LED #9
0120 0275 A5 10 LDA LTMSK+2 ; ICOMBINE PATTERNS FOR OUTPUT.
0121 0277 D0 07 BNE OFFLD9 ; IIF MASK# <> 0, LED 9 OFF.
0122 0279 A9 01 LDA PORT ; JTURN ON LED 9.
0123 027B BD 00 A0 STA PORTIB
0124 027E A9 00 LDA #0 ; IRESET A 40 PATTERN WON'T BE BAD.
0125 0280 05 0E OFFLD9 ORA LTMSK ; ICOMBINE REST OF PATTERNS.
0126 0282 05 0F ORA LTMSK+1
0127 0284 BD 01 A0 STA PORTIA ; ISET LIGHTS.
0128 0287 AD 00 AC LDA PORT3B ; IToggle SPEAKER.

--- Fig. 7.9: Slot Machine Program (Continued) ---
REAL TIME SIMULATION

```
0129 028A 49 FF EOR #FF
0130 028C 8D 00 AC STA PORT3h
0131 028F CA NXTPUD DEX #DECINEMNT X FOR NEXT UPDATE.
0132 0290 10 B5 BPL UPDTLP #IF X=0, DO NEXT UPDATE.
0133 0292 A0 50 LDY #SPDPRM #DELAY A BIT TO SLOW
0134 0294 BB WAIT DEY #FLASHING OF LEADS.
0135 0295 DD FD BNE WAIT #CHECK IF ALL COLUMNS OF
0136 0297 A5 05 LDA SPEEDS #LEDS STOPPED.
0137 0299 05 06 ORA SPEEDS+1
0138 029B 05 07 ORA SPEEDS+2
0140 029D 00 A6 BNE UPDATE #IF NOT, DO NEXT SEQUENCE
0141 029F C0 00 LDA #FF #OF UPDATES.
0142 029F A9 FF LDA #FF
0143 02A1 B5 03 STA DUR #DELAY TO SHOW USER PATTERN.
0144 02A3 20 30 03 JSR DELAY #ALL LEADS STOPPED; DONE.
0145 02A6 60 RTS
0146 02A7 60
0147 02A7 60
0148 02A7 60
0149 02A7 60
0150 02A7 60 #SUBROUTINE TO EVALUATE PRODUCT OF SIGN
0151 02A7 60 AND LOSEHENDGAME.
0152 02A7 60
0153 02A7 60 #DISPLAY SCORE W/TONES FOR WIN, LOSE, WINDGAME.
0154 02A7 60 #AND LOSEHENDGAME.
0155 02A7 60
0156 02A7 60
0157 02A7 60
0158 02A7 60
0159 02A7 60
0160 02A7 60
0161 02A7 60
0162 02A7 60
0163 02A7 60
0164 02A7 60
0165 02A7 60
0166 02A7 60
0167 02A7 60
0168 02A7 60
0169 02A7 60
0170 02A7 60
0171 02A7 60
0172 02A7 60
0173 02A7 60
0174 02A7 60
0175 02A7 60
0176 02A7 60
0177 02A7 60
0178 02A7 60
0179 02A7 60
0180 02A7 60
0181 02A7 60
0182 02A7 60
0183 02A7 60
0184 02A7 60
0185 02A7 60
0186 02A7 60
0187 02A7 60
0188 02A7 60
0189 02A7 60
0190 02A7 60
0191 02A7 60
0192 02A7 60
0193 02A7 60
0194 02A7 60
0195 02A7 60
0196 02A7 60
0197 02A7 60
0198 02A7 60
0199 02A7 60
0200 02A7 60
0201 02A7 60
```

---

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

115
0202 02FC AP 04 LDA #4
0203 02FE BS 03 STA DUR
0204 0300 ; SHORT DURATION FOR INDIVIDUAL
0205 0300 A5 05 RISE LDA TEMP
0206 0300 26 4A 03 JSR TONE
0207 0300 CA 00 DEC TEMP
0208 0300 60 60 NEXT DEEP WILL BE HIGHER.
0209 0300 D0 F7 BNE RISE
0210 0309 60 RTS
0211 030A CA 02 LOSE DEC SCORE
0212 030E 26 3D 03 JSR LIGHT
0213 0311 AF 00 LDA #!TONE
0214 0312 26 64 03 JSR TONE
0215 0316 A4 02 LDY SCORE
0216 031B FC 01 BEQ LOSEND
0217 031A 60 RTS
0218 031B AF 00 IF NOT, RETURN FOR NEXT SPIN.
0219 031D 85 00 LOSEND LDA #0
0220 031F BD 01 A0 STA PORTIA
0221 0322 AF 04 LDA #4
0222 0324 BS 03 STA DUR
0223 0326 A5 05 FALL LDA TEMP
0224 0328 26 64 03 JSR TONE
0225 032B E6 00 INC TEMP
0226 032D DF F7 BNE FALL
0227 032F 60 RTS
0228 0330 $VARIABLE LENGTH DELAY SUBROUTINE.
0229 0330 $DELAY LENGTH = (2048+!CONTENT OF DURH16) US.
0230 0330 $;
0231 0330 ;
0232 0330 A4 03 DELAY LDY DUR
0233 0332 A2 FF DL1 LDX //FF
0234 0334 D0 00 DL2 BNE //2
0235 0336 CA DEX
0236 0337 DF F6 BNE DL2
0237 0339 BB DEY
0238 033A D0 F6 BNE DL1
0239 033C 60 RTS
0240 033D 60 RETURN.
0241 033D 60 SUBROUTINE TO LIGHT LED CORRESPONDING
0242 033D 60 TO THE CONTENTS OF REGISTER Y ON ENTERING.
0243 033D 60 ;
0244 033D A9 00 LIGHT LDA #0
0245 033E BS 00 STA TEMP
0246 0341 BD 01 A0 STA PORTIA
0247 0344 BD 00 A0 STA PORTIB
0248 034A CA 0F CPY $15
0249 034B F0 01 BEQ #3
0250 0348 88 DEY
0251 034C 3B SEC
0252 034D 2A LSHFT ROL A
0253 034E 90 05 BCC LTCC
0254 0350 82 04 OCCURRED INTO HIGH BYTE.
0255 0352 A2 FF LDX //FF
0256 0352 B6 00 STX TEMP
0257 0354 2A ROL A
0258 0355 BB LTCC
0259 0356 10 F5 BPL LSHFT
0260 0358 A6 00 LDX TEMP
0261 035A 8E 00 BNE HBYTE
0262 035C 1000 HBYTE.
0263 035C BD 01 A0 LBYTE STA PORTIA
0264 035F 60 RTS
0265 0360 8D 00 A0 HBYTE STA PORTIB
0266 0364 BD 00 A0 STORE A IN HIGH ORDER LEDS.
0267 0368 8E 00 RTS
0268 0369 8E 00 ;
0269 036A 64 44 TONE GENERATION SUBROUTINE.
0270 036A 64 44
0271 036B A9 00 TONE STA FRED
0272 036B BD 00 AC STA PORTB
0273 036B A9 00 LDA $00

---

Fig. 7.9: Slot Machine Program (Continued)

116
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:

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

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 \( \text{RND} + 1 \), where it will be used later by the random number generation subroutine:

\[
\begin{align*}
\text{LDA TI\text{CL}} \\
\text{STA RND} + 1
\end{align*}
\]

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

\[
\begin{align*}
\text{START} \\
\text{LDA } \#8 \\
\text{STA SCORE}
\end{align*}
\]

and displayed:

\[
\begin{align*}
\text{TAY} \\
\text{Y must contain it} \\
\text{JSR LIGHT}
\end{align*}
\]

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:

\[
\begin{align*}
\text{KEY} \\
\text{JSR GETKEY}
\end{align*}
\]
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 INCRR 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 INCRR, INCRR + 1, and INCRR + 2. They are initialized to 0:
ADVANCED 6502 PROGRAMMING

DISPLY
   LDA #0
   STA INCR
   STA INCR + 1
   STA INCR + 2

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
   CMP #LOLIM    Too small?
   BCC GETRND

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:
**Fig. 7.10: Spinning the Wheels**

- **Update**
  - LDX #2: Set counter for 3 iterations

The speed is tested for the value 0:

- **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>Instruction</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>STY INCR,X</td>
</tr>
<tr>
<td></td>
<td>Restore pointer(X)</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>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STX TEMP</td>
<td>Multiply X by 3</td>
</tr>
<tr>
<td>TXA</td>
<td></td>
</tr>
<tr>
<td>ASL A</td>
<td>Left shift</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
SPDUPD
LDY SPEEDS,X
INY
STY SPEEDS,X

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 PORTIB    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 PORT1B

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:

```
LDA #0
OFFLD9 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 PORT1A

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 UPDLP  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:

\[
\begin{align*}
\text{HITONE} &= \$20 \\
\text{LOTONE} &= \$F0
\end{align*}
\]

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:

\[
\text{EVAL} \quad \text{LDA #0} \\
\text{STA VALUES} \\
\text{STA VALUES + 1} \\
\text{STA VALUES + 2}
\]

The temporary score is also set to 0:

\[
\text{STA SCORTP}
\]

Index register Y will be used as a column pointer, and the number of LEDs lit in each 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:

\[
\begin{align*}
\text{CNTLP} & \quad \text{LDY #2} \quad 3 \text{ iterations} \\
\text{LDX INCR,Y} \\
\text{INC VALUES,X}
\end{align*}
\]
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
ASL A
ASL A
```

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  Turn on all LEDs
STA PORT1B
```

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  Freq. parameter
LDA #0
STA SCORE  Clear for restart
LDA #4
STA DUR  Beep duration
```

The TONE subroutine is used to generate a beep:

```
RISE  LDA TEMP  Get frequency
      JSR TONE  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:
LOSE       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

Next, the frequency constant is increased by 1, and the process is
restarted until the TMP register overflows.
INC TEMP    Next tone will be lower
      BNE FALL
      RTS

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 \((2046 \times \text{DUR} + 10)\) 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:

\[
\text{DELAY} \quad \text{LDY DUR}
\]

The inner loop delay is then implemented:

\[
\begin{align*}
\text{DL1} & : \quad \text{LDX \#FF} \\
\text{DL2} & : \quad \text{BNE * + 2} \quad \text{Waste time} \\
 & : \quad \text{DEX} \quad \text{Inner loop counter} \\
 & : \quad \text{BNE DL2} \quad \text{Inner loop}
\end{align*}
\]

And, finally, the outer loop is implemented:

\[
\begin{align*}
\text{DEY} \\
\text{BNE DL1} \\
\text{RTS}
\end{align*}
\]

**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 * + 3
```

Code for uncorrected bit?
If so, no change

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

```
DEY
SEC
LTSHFT
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
```
Fig. 7.12: Creating the LED Pattern

However, if the ‘‘1’’ bit does fall off the left end of the accumulator, the value ‘‘FF’’ is loaded at memory location TEMP to signal this occurrence. Remember that the value was cleared in the second instruction of the LIGHT subroutine.

LDX #$FF
STX TEMP

The ‘‘1’’ bit is then moved from the carry into the right-most position of the accumulator. Later, the value contained in memory location TEMP will be checked, and this will determine whether the pattern contained in the accumulator is to be sent to Port 1A or to Port 1B.
The shifting process continues. The counter is decremented, and, if it reaches the value "0," we are done; otherwise, the process is repeated:

\[
\begin{align*}
\text{LTCC} & \quad \text{ROL A} \\
& \quad \text{DEY} \\
& \quad \text{BPL LTSHFT}
\end{align*}
\]

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:

\[
\begin{align*}
\text{LDX TEMP} & \quad \text{Get overflow flag} \\
\text{BNE HIBYTE} & \\
\text{LOBYTE} & \quad \text{STA PORT1A} \\
& \quad \text{A sent to low LEDs} \\
\text{RTS} & \quad \text{Return} \\
\text{HIBYTE} & \quad \text{STA PORT1B} \\
& \quad \text{A sent to high LEDs} \\
\text{RTS} &
\end{align*}
\]

**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. Real Time Strategies (Echo)

INTRODUCTION

A stack technique is used to accumulate information. It is compared to the use of scratch locations.

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 suc-
cessfully. 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 8.1, 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
REAL TIME STRATEGIES

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 box 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
LINE   LOC    CODE     LINE
0002  0000  : ECHO
0003  0000  I PATTERN/TONE RECALL AND ESP TEST PROGRAM.
0004  0000  I THE USER GUESSES A PATTERN OF LIT LEDS AND
0005  0000  I THEIR ASSOCIATED TONES. THE TONE/LIGHT
0006  0000  I COMBINATION CAN BE PLAYED; SO THAT THE USER
0007  0000  I MUST REMEMBER IT AND REENTER IT CORRECTLY.
0008  0000  I OPERATING THE PROGRAM:
0009  0000  I THE STARTING ADDRESS IS $200
0010  0000  I THE BOTTOM ROW OF LEDS IS AN INDICATOR
0011  0000  I FOR PROGRAM STATUS; THE LEFTMOST
0012  0000  I ONE ($10) INDICATES THAT THE PROGRAM
0013  0000  I IS EXPECTING THE USER TO INPUT THE LENGTH
0014  0000  I OF THE SEQUENCE TO BE GUESSED.
0015  0000  I THE LED SECOND FROM THE LEFT ($11) INDICATES
0016  0000  I THAT THE PROGRAM EXPECTS EITHER A GUESS (1-9),
0017  0000  I THE COMMAND TO RESTART THE GAME (0), OR
0018  0000  I THE COMMAND TO PLAY THE SEQUENCE (A-F).
0019  0000  I THE KEYS 1-9 ARE ASSOCIATED WITH THE
0020  0000  I LEDS 1-9.
0021  0000  I LOOKING AT THE SEQUENCE WHILE IN THE MIDDLE
0022  0000  I OF GUESSING IT WILL ERASE ALL PREVIOUS
0023  0000  I GUESSES (RESET GEDNO AND ERRS TO 0).
0024  0000  I AFTER A WIN, THE PROGRAM RESTARTS.
0025  0000  I
0026  0000  I LINKAGES:
0027  0000  GETKEY = #100
0028  0000  I
0029  0000  I VARIABLE STORAGES:
0030  0000  DIGITS = $00  NUMBER OF DIGITS IN SEQUENCE
0031  0000  GEDNO = $10  NUMBER OF CURRENT GUESS.
0032  0000  (WHERE THE USER IS IN THE SERIES)
0033  0000  ERRS = $02  NUMBER OF ERRORS MADE IN
0034  0000  I GUESSING CURRENT SEQUENCE.
0035  0000  DUR = $03  ITEM STORAGE FOR DURATION.
0036  0000  FREQ = $04  ITEM STORAGE FOR FREQUENCY.
0037  0000  TEMP = $05  TEMPORARY STORAGE FOR $4 REG.
0038  0000  TABLE = $06  ITEM STORAGE FOR SEQUENCE;
0039  0000  RND = $0F  ISCRATCHP FOR RANDOM $4 GEN.
0040  0000  I 6522 VIA $1 ADDRESSES:
0041  0000  PORT1A = $A001
0042  0000  DDR1A = $A003
0043  0000  PORT1B = $A000
0044  0000  DDR1B = $A002
0045  0000  TICL = $A004
0046  0000  I 6522 VIA $3 ADDRESSES
0047  0000  PORT3B = $A000
0048  0000  DDR3B = $A002
0049  0000  I
0050  0000  I $ = #200
0051  0000  I
0052  0200  A9 FF  START LDA #FF  ;SET UP DATA DIRECTION REGISTERS.
0053  0222  B0 03 A0  STA DDR1A
0054  0205  BB 02 A0  STA DDR1B
0055  0208  BB 02 AC  STA DDR3B
0056  020B  AB 00  LDA #0  ;CLEAR VARIABLE STORAGES
0057  020C  BB 01 A0  STA PORT1A  ;AND LEDS
0058  0210  BS 02  STA ERRS
0059  0212  BS 01  STA GEDNO
0060  0214  AB 04 A0  LDA TICL  ;SET SEQ FOR RND & GEN.
0061  0217  BS 10  STA RNDH1  ;AND STORE IN RND SCRATCH.
0062  0219  BS 13  STA RNDL1
0063  021B  AB 02  LDA $C010  ;TURN LED $10 ON TO INDICATE
0064  021D  BB 00 A0  STA PORT1B  ;LENS INPUT.
0065  0220  20 00 01 DISK JSR GETKEY  ;GET LENGTH OF SERIES.
0066  0223  C9 00  CMP #0  ;IS IT 0 ?
0067  0225  F0 F9  BEQ DISK  ;EYES; GET ANOTHER.
0068  0222  C9 0A  CMP #10  ;LENGTH GREATER THAN 10 ?
0069  0229  10 F5  BPL DISK  ;EYES; GET ANOTHER.
0070  0228  BS 00  STA DIGITS HAVE VALID LENGTH

Fig. 8.5: Echo Program

145
ADVANCED 6502 PROGRAMMING

0071 022D AA  TAX     H/WISE LENGTH-1 AS INDEX FOR FILLING...
0072 022E CA  DEX     1..SERIES w/RANDOM VALUES.
0073 022F B6 05  FILL   STX TEMP    SAVE X FROM "RANDOM".
0074 0231 20 E7 02  JSR      RANDOM
0075 0234 A6 05  LDX TEMP   ISTORE X
0076 0236 FB  SED     I/D A DEIMAL ADJUST
0077 0237 18  CLC
0078 0238 49 00  ADC #0
0079 023A 08  CLD
0080 023B 29 0F  AND #0FF IREMOVE UPPER NYBBLE SO
0081 023C 6F 84  BLD     #0 CAN'T BE ZERO.
0082 023D F0 F0  STA #0 IF #0 IN TABLE
0083 023E 95 06  STA TABLE:X FSTORE #0 IN TABLE
0084 0241 CA  DEX     ICREMENT FOR NEXT
0085 0242 10 EB  BPL     FILL #0 IF NOT DONE
0086 0244 09 00  KEY     LDA #0 IREAD LEDS
0087 0246 BD 01 A0  STA PORTA
0088 0249 A9 04  LDA #00100 ITURN INPUT INDICATOR ON.
0089 024A BD 00 A0  STA PORTB
0090 024B 20 00 01  JSR GETKEY IGET GUESS OR PLAY CMD.
0091 0251 C9 00  CMP #0 IIS IT 0 ?
0092 0253 F0 AB  BRA     BED START IIF YES; RESTART.
0093 0255 C9 00  CMP #10 INumber < 10 ?
0094 0257 30 22  BMI     IIF YES; EVALUATE GUESS.
0095 0259 29 00  ; IROUTINE TO DISPLAY SERIES TO BE GUESSED BY
0096 0259 29 00  ; ILIGHTING LEDs AND PLAYING TONES IN SEQUENCE.
0097 0259 29 00  ; IROUTINE TO DISPLAY SERIES TO BE GUESSED BY
0098 0259 29 00  ; IROUTINE TO DISPLAY SERIES TO BE GUESSED BY
0099 0259 29 00  ; ISHOW LDX #0
0100 0259 B6 01  STX GESNO ICLEAR ALL CURRENT GUESSES.
0101 0259 B6 02  STX ERRS ICLEAR CURRENT ERRORS.
0102 025F 85 06  SHOWLP LDA TABLE:X IGET XTH ENTRY IN SERIES TABLE.
0103 0261 B6 05  STX TEMP ISAVE X
0104 0263 0B 0F  JSR LIGHT FLIGHT LED#TABLE(X))
0105 0265 20 FA 02  JSR PLAY IPLAY TONE#(TABLE(X))
0106 0266 0A FF  LDY #FF ISET LOOP CNTR. FOR DELAY
0107 0268 6A 03  DELAY IROL DUR IWASTE TIME
0108 0269 26 03  ROL DUR
0109 026F BB  DEY     ICOUNT DOWN...
0110 0270 D0 F9  BNE DELAY IIF NOT DONE; LOOP AGAIN.
0111 0272 A6 05  LDX TEMP IRESTORE X
0112 0274 8E 00  INX     IINCREMENT INDEX TO SHOW NEXT
0113 0275 4E 00  CPX DIGITS IALL DIGITS SHOWN?
0114 0277 D0 E6  BNE SHOWLP IIF NOT; SHOW NEXT.
0115 0279 F9 C9  BNE KEY IDONE; GET NEXT INPUT.
0116 027B 2B  ; IROUTINE TO EVALUATE GUESSES OF PLAYER.
0117 027B 2B  ; IROUTINE TO EVALUATE GUESSES OF PLAYER.
0118 027B 2B  ; IROUTINE TO EVALUATE GUESSES OF PLAYER.
0119 027B 2B  ; IROUTINE TO EVALUATE GUESSES OF PLAYER.
0120 027B A6 01  EVAL LDX GESNO IGET NUMBER OF GUESS.
0121 027F F0 00  LDA GESNO IFIT CORRESPONDING DIGIT? Icorrect #0 ALWAYS.
0122 0281 E6 02  INC ERRS IERROR COUNTER.
0123 0283 80 00  LDA #00 ILOAD ERROR CODE FOR LOW TONE TO INDICATE
0124 0285 B5 03  LDA #02 IFurrect #0?
0125 0287 A9 FF  JSR PLAY IPLAY IT
0126 0289 20 04 03  JSR PICKUP ICHECK FOR ENDGAME
0127 028C F0 06  JSR ENDCHP ICHECK FOR ENDGAME
0128 028E 20 CF 02  CORRECT JSR LIGHT IIVALIDATE CORRECT GUESS...
0129 0291 20 FA 02  JSR PLAY ISHOW NUMBER OF ERRORS.
0130 0294 E6 01  ENDCHP INC GESNO IONE MORE GUESS TAKEN.
0131 0296 A5 00  LDA DIGITS
0132 0298 C5 01  CMP GESNO TALL DIGITS GUESSED? IERRORS?
0133 029A D0 A8  BNE KEY IF NOT; GET NEXT.
0134 029C A5 02  STA ERRS IGET NUMBER OF ERRORS.
0135 029E C9 00  CMP #0 IHAVE ERRORS?
0136 02A0 F0 15  BNE WIN IIF NOT; PLAYER WINS.
0137 02A2 20 CF 02  LOSE JSR LIGHT ISHOW NUMBER OF ERRORS.
0138 02A5 A9 09  LDA #9 IPAY & DESCENDING TONES
0139 02A7 48  LODELP PHI
0140 02A8 20 FA 02  JSR PLAY IPLAY
0141 02A8 48  PLA

Fig. 8.3: Echo Program (Continued)
REAL TIME STRATEGIES

0142 02AC 3B  SEC
0143 02AD E9 01  SBC #1
0144 02AF DD FF  BNE LDEL0
0145 02B1 B5 01  STA DESMS ICLEAR VARIABLES
0146 02B3 B5 02  STA ENSS
0147 02B5 F0 B0  BNE KEY IGET NEXT GUESS SEQUENCE
0148 02B7 A9 FF  WIN LDA #4F  ITURN ALL LEDS ON FOR WIN
0149 02B9 B0 01 00  STA PORTA
0150 02BC B0 00 00  STA PORTB
0151 02BF A9 01  LDA #1 IPLAY B ASCENDING TONE
0152 02C0 00 06  WINLP PBA
0153 02C2 20 FA 02  JSR PLAY
0154 02C5 68  PLA
0155 02C6 18  CLC
0156 02C7 A9 01  ADC #01
0157 02C9 B9 0A  CMP #10
0158 02CB DD FF  BNE WINLP
0159 02CD F0 B4  BNE Jennings USE DOUBLE--JUMP FOR RESTART
0160 02CF
0161 02CF JROUTINE TO LIGHT MTH LED, WHERE N IS
0162 02CF JTHE NUMBER PASSED AS A PARAMETER IN
0163 02CF JTHE ACCUMULATOR.
0164 02CF
0165 02C0 4B  LIGHT PHA  "SAVE A"
0166 02D0 0A  TAY  "USE A AS COUNTER IN Y"
0167 02D1 A9 00  LDA #0 ICLEAR A FOR BIT SHIFT
0168 02D2 BB 00 00  STA PORTB ICLEAR HI LEDS.
0169 02D6 3B  SEC  IGENERATE HI BIT TO SHIFT LEFT.
0170 02D7 2A  LSHFT ROL A IMOVE HI BIT LEFT.
0171 02DB BB  DEY IDECREMENT COUNTER
0172 02D9 B0 FF  BNE LSHFT ISHIFTS DONE?
0173 02DB B0 01 00  STA PORTA ISTORE CORRECT PATTERN
0174 02DE 90 05  BCC LTCC IBIT 9 NOT HI, DONE.
0175 02E0 A9 01  LDA #1
0176 02E2 B0 00 00  STA PORTB ITURN LED #0 ON.
0177 02E5 4B  LTCC PLA IRESTORE A
0178 02E6 60  RTS IDONE.
0179 02E7
0180 02E7 IRANDOM NUMBER GENERATOR, RETURNS W/ NEW
0181 02E7 IRANDOM NUMBER IN A.
0182 02E7
0183 02E7 RANDOM SEC
0184 02E8 A5 10  LDA RND+1
0185 02EA 65 13  ADC RND+4
0186 02EC 65 14  ADC RND+5
0187 02EE B5 0F  STA RND
0188 02F0 A2 04  LDX #4
0189 02F2 B5 0F  RNDLP LDA RND+x
0190 02F4 95 10  STA RND+x
0191 02FA CA  DEX
0192 02F7 10 F9  BPL RNDLP
0193 02F9 60  RTS
0194 02FA
0195 02FA JROUTINE TO PLAY TONE WHOSE NUMBER IS PASSED
0196 02FA JIN BY ACCUM. IF ENTERED AT FLYTON, IT WILL
0197 02FA JPLAY TONE WHOSE LENGTH IS IN DUR, FREQUENCY
0198 02FA JIN ACCUMULATOR.
0199 02FA
0200 02FA A8  PLAY TAY IUSE TONE# AS INDEX...
0201 02FB BB  DEY IDECREMENT TO MATCH TABLES
0202 02FC B9 27 03  LDA DURAT+y IGET DURATION FOR TONE# N.
0203 02FF B5 03  STA DUR ISAVE IT.
0204 0301 B9 1E 03  LDA NOTAB+y IGET FREQUENCY, CONST FOR TONE# N
0205 0304 B5 04  PLYTON STA FREQ IVSAVE IT.
0206 0306 A9 00  LDA #0 ISET SPKR PORT LO.
0207 0308 BD 00 AC  STA PORT3B
0208 030B A6 03  LDAX DUR IGET DURATION IN # OF 1/2 CYCLES.
0209 030D B4 04  FL2 LDY FREQ IGET FREQUENCY
0210 030F BB  DEY ICOUNT DOWN DELAY...
0211 0310 18  CLC IWASTE TIME
0212 0311 90 00  BCC ++2

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
STA PORT1A
```

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    Read timer counter.
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 #%010   Pattern for LED 10
STA PORT1B  Specify length
```

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":

149
ADVANCED 6502 PROGRAMMING

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  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
REAL TIME STRATEGIES

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 $0F

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:

151
DEX

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  LDA #0
STA PORT1A
LDA #0100
STA PORT1B

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”:

CMP #0  Is it 0?
STRTJP  BEQ START  If yes, restart

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:

CMP #10  Number <10?
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.

*The 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

152
RESTARTED AND THE TEMPORARY VARIABLES ARE RESET TO 0:

SHOW

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX #0</td>
<td></td>
</tr>
<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>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA TABLE,X</td>
<td>Get Xth entry in table</td>
</tr>
<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>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDY #$FF</td>
<td>Dummy instruction</td>
</tr>
<tr>
<td>ROR DUR</td>
<td>dummy</td>
</tr>
<tr>
<td>ROL DUR</td>
<td>Dummy</td>
</tr>
<tr>
<td>DEY</td>
<td>Count down</td>
</tr>
<tr>
<td>BNE DELAY</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>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX TEMP</td>
<td>Restore X</td>
</tr>
<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>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPX DIGITS</td>
<td>All digits shown?</td>
</tr>
<tr>
<td>BNE SHOWLP</td>
<td></td>
</tr>
<tr>
<td>BEQ KEY</td>
<td>Done, get next input</td>
</tr>
</tbody>
</table>

The EVAL Routine

Let us now examine the routine which evaluates the guess of the

153
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 CORRECT    If correct, tell player
```

If there is a match, a branch occurs to location CORRECT; 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    Play it
```

A jump then occurs to location ENDCHECK:

```
BEQ ENDCHECK    Check for end of game
```

**Exercise 8-3:** Examine the BEQ instruction above. Will it always result in a jump to label ENDCHECK? (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:

```
CORRECT    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
CMP GESNO  All digits guessed?
BNE KEY  If not, get next key closure

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 Get next guess sequence

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 Save A on the stack
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":

157
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:

```
PLAY      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:

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

The speaker is then turned off:

```
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.
ADVANCED 6502 PROGRAMMING

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:

<table>
<thead>
<tr>
<th>FL2</th>
<th>LDX DUR</th>
<th>Get duration in # of ½ cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDY FREQ</td>
<td>Get frequency</td>
</tr>
</tbody>
</table>

Next, let us implement the inner loop delay:

<table>
<thead>
<tr>
<th>FL1</th>
<th>DEY</th>
<th>Waste time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCC * + 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNE FL1</td>
<td>Delay loop</td>
</tr>
</tbody>
</table>

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.

EOR #$FF Complement port

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

STA PORT3B

The outer loop can then be completed:

<table>
<thead>
<tr>
<th></th>
<th>DEX</th>
<th>Outer loop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BNE FL2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

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?
<table>
<thead>
<tr>
<th>NOTE</th>
<th>FREQUENCY CONSTANT</th>
<th>DURATION CONSTANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C9</td>
<td>6B</td>
</tr>
<tr>
<td>2</td>
<td>BE</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>A9</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
<td>8F</td>
</tr>
<tr>
<td>5</td>
<td>8E</td>
<td>94</td>
</tr>
<tr>
<td>6</td>
<td>7E</td>
<td>AA</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
<td>BF</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>D7</td>
</tr>
<tr>
<td>9</td>
<td>5E</td>
<td>E4</td>
</tr>
</tbody>
</table>

Fig. 8.6: Frequency and Duration Constants
9. Using Interrupts  
(Mindbender)

INTRODUCTION

Interrupts are generated by using the programmable interrupt timer of the 6522 VIA, a common 6502 I/O chip. The programmable interrupt timer is used in the free-running mode to generate a wave form.

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.

![Fig. 9.1: Enter Length of Sequence](image)

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.

![Fig. 9.2: Enter Your Guess](image)

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 DIG0.

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).
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 “OF” 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-
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 IRQVECL
LDA #$03
STA IRQVECH
```

Low interrupt vector
High interrupt vector
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 waveform from timer 1 is shown in Figure 9.12. In order to compute the exact duration of the pulse, note that the pulse dura-
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:

\[
\begin{align*}
\text{STA DDR1A} & \quad \text{Output} \\
\text{STA DDR1B} & \quad \text{Output} \\
\text{STA DDR3B} & \quad \text{Output}
\end{align*}
\]

All LEDs are then cleared:

<table>
<thead>
<tr>
<th>ACR7 OUTPUT ENABLE</th>
<th>ACR6 INPUT ENABLE</th>
<th>MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (ONE-SHOT)</td>
<td>0</td>
<td>GENERATE TIME OUT INT WHEN T1 LOADED PB7 DISABLED</td>
</tr>
<tr>
<td>0 (FREE RUN)</td>
<td>1</td>
<td>GENERATE CONTINUOUS INT PB7 DISABLED</td>
</tr>
<tr>
<td>1 (ONE-SHOT)</td>
<td>0</td>
<td>GENERATE INT AND OUTPUT PULSE ON PB7 EVERYTIME T1 IS LOADED = ONE-SHOT AND PROGRAMMABLE WIDTH PULSE</td>
</tr>
<tr>
<td>1 (FREE RUN)</td>
<td>1</td>
<td>GENERATE CONTINUOUS INT AND SQUARE WAVE OUTPUT ON PB7</td>
</tr>
</tbody>
</table>

Fig. 9.11: 6522 Auxiliary Control Register Selects Timer 1 Operating Modes
KEY1
LDA #0
STA PORT1A
STA PORT1B

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

STA MASKA
STA MASKB

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 #%00000010 Select LED 10
STA PORT1B Turn it on

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 T1LL
STA RND + 1
STA RND + 4
STA RND + 5

Then a random number is obtained using the RANDOM subroutine:

LDY DIGITS Get # of digits to guess
DEY Count to 0
RAND JSR RANDOM 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 Decimal Adjust
CLD

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
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 PORT1A
STX PORT1B
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        Increment guess # pointer
         CPY DIGITS  All tested?
         BNE ENTRYLP 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        Increment digit # pointer
         CPX DIGITS  All digits checked
```

If the limit has not been reached, a branch occurs back to the beginning of the outer loop at location DIGLP:
BNE 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                      Get ready for add
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":

182
USING INTERRUPTS

<table>
<thead>
<tr>
<th>LDA #0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC</td>
</tr>
<tr>
<td>ROL A</td>
</tr>
<tr>
<td>DEY</td>
</tr>
<tr>
<td>BNE SHIFT</td>
</tr>
<tr>
<td>RTS</td>
</tr>
</tbody>
</table>

Shift into position

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 T1LL

Finally, the accumulator is restored, and a return occurs to the main program:
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?

```
;MINDBENDER PROGRAM
;Plays Mindbender Game: User specifies length of number to be guessed, then guesses digits, and computer tells player how many of the digits guessed were right, and how many of those correct digits were in the correct place, until the player can guess the number. On the board, blinking LEDs indicate correct value & correct digit, and nonblinking LEDs show correct digit value, but wrong place.
;The bottom row of LEDs is used to show the mode of the program: if the leftmost LED is lit, the program expects the user to enter the length of the number to be guessed, if the two leftmost LEDs are lit, the program expects a guess. The program rejects unsuitable values for a number length, which can only be 1-9. A value other than 0-9 for a guess restarts the game. A low tone denotes a bad guess, a high tone, a win. After a win, the program restarts. An interrupt routine is used to blink the LEDs.

=200
GETKEY =100
ACCESS =$8896
DIGITS =$00
NUMBER OF DIGITS TO BE GUESSED
DUR =$01
TONE DURATION CONSTANT
XTEMP =$02
TEMP STORAGE FOR X REG.
YTEMP =$03
TEMP STORAGE FOR Y REG.
CNT =$04
KEEPS TRACK OF # OF MATCHES
MASKA =$05
CONTAINS PATTERN EOR'ED WITH LED STATUS REGISTER A TO CAUSE BLINK
MASKB =$06
ILED PORT B BLINK MASK
FREQ =$07
TEMP STORAGE FOR TONE FREQUENCY
CNT1 =$08
# OF CORRECT DIGITS IN RIGHT PLACE
RND =$09
FIRST OF RANDOM # LOCATIONS
DIG0 =$0F
FIRST OF 9 DIGIT LOCATIONS
ENTRY0 =$10
FIRST OF 9 GUESS LOCATIONS
IROVECL =$A67E
INTERRUPT VECTOR LOW ORDER BYTE
IROVECH =$A67F
...AND HIGH ORDER
6522 VIA #1 REGISTERS!
```

Fig. 9.13: Mindbender Program
IER $=A00E ; INTERRUPT ENABLE REGISTER
ACR $=A008 ; AUXILIARY CONTROL REGISTER
TILL $=A004 ; TIMER 1 LATCH LOW
T1CH $=A005 ; TIMER 1 COUNTER HIGH
PORTIA $=A001 ; VIA 1 PORT A IN/OUT REG
DDR1A $=A003 ; VIA 1 PORT A DATA DIRECTION REG.
PORTIB $=A000 ; VIA 1 PORT B IN/OUT REG
DDR1B $=A002 ; VIA 1 PORT B DATA DIRECTION REG.
PORT3B $=AC00 ; VIA 3 PORT B IN/OUT REG
DDR3B $=AC02 ; VIA 3 PORT B DATA DIRECTION REG

; ROUTINE TO SET UP VARIABLES AND INTERRUPT TIMER FOR
; I.E.D. FLASHING

0200: 20 B6 8B JSR ACCESS ; UNPROTECT SYSTEM MEMORY
0203: A9 EA LDA $EEA LOAD LOW INTERRUPT VECTOR
0205: BD 7E A6 STA IRQVECL ...AND STORE AT VECTOR LOCATION
0208: A9 03 LDA $03 ILOAD INTERRUPT VECTOR,...
020A: BD 7F A6 STA IRQVCH ...AND STORE.
020D: A9 7F LDA $7F ICLEAR INTERRUPT ENABLE REGISTER
020F: BD 0E A0 STA IER
0212: A9 C0 LDA $C0 IFENABLE TIMER 1 INTERRUPT
0214: BD 0E A0 STA IER
0217: A9 40 LDA $40 IFENABLE TIMER 1 IN FREE-RUN MODE
0219: BD 08 A0 STA ACR
021C: A9 FF LDA $FF
021E: BD 04 A0 STA TILL IFSET LOW LATCH ON TIMER 1
0221: BD 05 A0 STA TICH IFSET LATCH HIGH & START COUNT
0224: 5B CLI IFENABLE INTERRUPTS
0225: BD 03 A0 STA DDR1A IFSET VIA 1 PORT A FOR OUTPUT
0228: BD 02 A0 STA DDR1B IFSET VIA 1 PORT B FOR OUTPUT
022B: BD 02 AC STA DDR3B IFSET VIA 3 PORT B FOR OUTPUT
022E: A9 00 KEY1 LDA $0 IFCLEAR LEDS
0230: BD 01 A0 STA PORTIA
0233: BD 00 A0 STA PORTIB
0236: BS 05 STA MASKA IFCLEAR BLINK MASKS
0238: BS 06 STA MASKB

; ROUTINE TO GET NUMBER OF DIGITS TO GUESS, THEN
; FILL THE DIGITS WITH RANDOM NUMBERS FROM 0-9

023A: A9 02 LDA $X00000010 IFLIGHT LED TO SIGNAL USER TO
023C: BD 00 A0 STA PORTIB IFINPUT OF # OF DIGITS NEEDED.
023E: BS 00 01 JSR GETKEY IFGET # OF DIGITS
0242: C9 0A CMP $10 IFKEY# >9, RESTART GAME
0244: 10 E8 BPL KEY1
0246: C9 00 CMP $0 IFCHECK FOR 0 DIGITS TO GUESS
0248: F0 E4 BEQ KEY1 IF...0 DIGITS NOT ALLOWED
024A: BS 00 STA DIGITS IFSTORE VALID # OF DIGITS
024C: AD 04 A0 LDA TILL IFSET RANDOM #.
024F: BS 0A STA RND+1 IFUSE IT TO START RANDOM
0251: BS 0B STA RND+4 IFNUMBER GENERATOR,
0253: BS 0E STA RND+5
0255: A4 00 LDB DIGITS IFGET # OF DIGITS TO BE GUESSED,
0257: 8D DEY IFAND COUNT TO 0, FILLING
025B: 20 FF 02 RAND JSR RANDOM IFGET RANDOM VALUE FOR DIGIT
025F: F8 SED
025C: 69 00 ADC $00 IFDECIMAL ADJUST
025E: D8 CLD
025F: 29 OF AND $X0001111 IFKEEP DIGIT <10
0261: 99 OF 00 STA DIG0-Y IFSAVE IT IN DIGIT TABLE.
0264: 88 DEY
0265: 10 F1 BPL RAND IFFILL NEXT DIGIT

---

Fig. 9.13: Mindbender Program (Continued)
Fig. 9.13: Mindbender Program (Continued)
USING INTERRUPTS

; DIGITS IN CORRECT PLACES = NUMBER OF DIGITS IF WIN, A HIGH PITCHED SOUND IS GENERATED, AND IF ANY DIGIT IS WRONG, A LOW SOUND IS GENERATED.
; 02D4: A6 08 TEST LDX CNT1 LOAD NUMBER OF CORRECT DIGITS
02D6: E4 00 CPX DIGITS ALL GUESSES CORRECT?
02D8: F0 08 BEQ WIN IF YES, PLAYER WINS
02DA: A9 72 BDF LDA ##72
02DC: 85 01 STA DUR SET UP LENGTH OF LOW TONE
02DE: A9 BE LDA ##BE TONE VALUE FOR LOW TONE
02E0: 20 12 03 JSR TONE SIGNAL BAD GUESSES W/TONE
02E3: F0 82 BEQ ENTER GET NEXT GUESSES
02E5: A9 FF WIN LDA ##FF DURATION FOR HIGH TONE
02E7: B5 01 STA DUR
02E9: A9 54 LDA ##54 TONE VALUE FOR HIGH TONE
02EB: 20 12 03 JSR TONE SIGNAL WIN
02EE: 4C 2E 02 JMP KEY1 RESTART GAME

; ROUTINE TO FILL ACCUMULATOR WITH '1' BITS, STARTING AT THE LOW ORDER END, UP TO AND INCLUDING THE BIT POSITION CORRESPONDING TO THE # OF LEADS TO BE LIT OR SET TO BLINKING.
; 02F1: B0 04 LITE BNE STRTSH IF Y NOT ZERO, SHIFT ONES IN
02F3: A9 00 LDA ##0 SPECIAL CASE: RESULT IS NO ONES.
02F5: 18 CLC
02F6: 60 RTS
02F7: A9 00 STRTSH LDA ##0 CLEAR A SO PATTERN WILL SHOW
02F9: 38 SHIFT SEC MAKE A BIT HIGH
02FA: 2A ROL A SHIFT IT TO CORRECT POSITION
02FB: B8 DEY BY LOOPSING TO # OF GUESS/DIGIT MATCHES, AS PASSED IN Y
02FC: D0 FB BNE SHIFT IF LOOP 'TIL DONE
02FE: 60 RTS

; RANDOM NUMBER GENERATOR
; USES NUMBERS A, B, C, D, E, F STORED AS RND THROUGH RND+5: ADDS B+E+F+1 AND PLACES RESULT IN A; THEN
; SHIFTS A TO B, B TO C, ETC. THE NEW RANDOM NUMBER WHICH IS BETWEEN 0 AND 255 INCLUSIVE IS IN THE
; ACCUMULATOR ON EXIT
; 02FF: 38 RANDOM SEC I CARRY ADDS VALUE 1
0300: A5 04 LDA RND+l ADD A+B+E AND CARRY
0302: 65 00 ADC RND+4
0304: 65 0E ADC RND+5
0306: B5 09 STA RND
0308: A2 04 LDX #4 SHIFT NUMBERS OVER
030A: B5 09 LDA RND+X
030C: 95 0A STA RND+1+x
030E: CA DEX
030F: 10 F9 IPL RPL
0311: 60 RTS

; TONE GENERATOR ROUTINE.
; DURATION OF TONE (NUMBER OF CYCLES TO CREATE)
; SHOULD BE IN 'DUR' ON ENTRY, AND THE NOTE VALUE
; (FREQUENCY) IN THE ACCUMULATOR.
; 0312: 85 07 TONE STA FREQ
0314: A9 FF LDA ##FF
0316: 80 00 AC STA PORT3B
0318: A9 00 LDA ##00
031A: A6 01 LDY DUR
031C: 04 07 FL2 LBY FREQ

Fig. 9.13: Mindbender Program (Continued)

187
ADVANCED 6502 PROGRAMMING

031F: 88 FL1 DEY
0320: 18 CLC
0321: 90 00 BCC +2
0323: D0 FA BNE FL1
0325: 49 FF EOR $FF
0327: 8D 00 AC STA PORT3B
032A: CA DEX
032B: D0 F0 BNE FL2
032D: 60 RTS

; INTERRUPT-HANDLING ROUTINE
; COMPLEMENTS LEDS AT EACH INTERRUPT

03EA: 48 PHA ; SAVE ACCUMULATOR
03EB: AD 01 A0 LDA PORTIA ; GET PORT FOR COMPLEMENTING
03EE: 45 05 EOR MASKA ; COMPLEMENT NECESSARY BITS
03F0: BD 01 A0 STA PORTIA ; STORE COMPLEMENTED CONTENTS
03F3: AD 00 A0 LDA PORT1B ; DO SAME WITH PORT1B
03F6: 45 06 EOR MASKB
03F8: BD 00 A0 STA PORT1B
03FB: AD 04 A0 LDA TILL ; CLEAR INTERRUPT BIT IN VIA
03FE: 68 PLA ; RESTORE ACCUMULATOR
03FF: 40 RTI ; DONE, RESUME PROGRAM

SYMBOL TABLE:
GETKEY 0100 ACCESS BBB6 DIGITS 0000
DUR 0001 XTEMP 0002 YTEMP 0003
CNT 0004 MASKA 0005 MASKB 0006
FREQ 0007 CNT1 0008 RND 0009
DIG0 000F ENTRY0 0018 IRQVECL A67E
IRQVECH A67F IER A00E ACR A00B
TILL A004 TICH A005 PORTIA A001
DDR1A A003 PORT1B A000 DDR1B A002
PORT3B AC00 DDR3B AC02 KEY1 022E
RAND 0258 ENTER 0267 KEY2 0273
DIGLP 0295 ENTRYCMP 029F ENTRYL 02A1
NEXTENT 02AA NEXTD 02AF CC 02C7
TEST 02B4 BAD 02B A W 02E5
LITE 02F STRTSH 02F7 SHIFT 02F9
RANDOM 02FF RPL 030A TONE 0312
FL2 031D FL1 031F
DONE

Fig. 9.13: Mindbender Program (Continued)
10. Complex Evaluation Technique (Blackjack)

INTRODUCTION

This problem involves a complex evaluation in a simple input/output environment and a very small amount of memory. The program generates light and sound effects and operates in real time.

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 com-
puter’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. Other-
wise, 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.)

![Diagram of LED display showing player wins and computer wins]

**Fig. 10.1: Indicating the Winner**

**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.
COMPLEX EVALUATION TECHNIQUE

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.

![Figure 10.2: First Hand](image)

![Figure 10.3: Player Receives A Second Card: Blackjack](image)
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 increase 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 computer 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.
COMPLEX EVALUATION TECHNIQUE

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 "11111111." 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 Game 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).

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**
Fig. 10.11: High Memory Map
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:

    JSR BLINKR    Set random blinking LED

and we save the first total for the current player’s hand:

    STA PHAND    Store player’s hand

then we do the same for the computer:

    JSR LIGHTR    Set random steady LED
    STA CHAND    Store computer’s hand

*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:

    ASK        JSR GETKEY

The key value must now be compared to “A” and to “C”:

    CMP #$0A
    BEQ HITPLR
    CMP #$0C    Is it computer’s turn?
    BEQ DEALER

If any other key has been pressed, it will be ignored and a new key will be read:

    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).

**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:

**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:

**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:

**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:

**LDA CHAND** Check hand for limit
**CMP #10**
**BCS WINNER** Yes. Decide winner.
ADVANCED 6502 PROGRAMMING

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  
    CMP PHAND        Compare hands

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 WHOLOW, 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
SCORER       JSR GETKEY

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       Tie: do not change LEDs
BMI SC

204
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
```

Player won: set left LEDs

If the player lost, the three right-most LEDs are lit:

```
SC
LDA #$B0
```

Player lost: set right LEDs

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
```

$01: Jump on win condition

Solid row of LEDs

Return to monitor

In the case of a player win, a blinking square is displayed and the program is terminated:

```
EN1
LDA #$FF
STA MASKA
```
LDA #$01
STA MASKB
RTS

Subroutines

SETBIT 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:

```
SETBIT      TAY     Save logical number
```

It is then compared to the limit value "7."

```
CMP #8
BCC SB0
```

If the value was greater than 7, we subtract 8 from it:

```
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:

```
SB0      TAX
```

A bit will now be shifted into the correct position of the accumulator. Let us first set the carry to "1":

```
SEC     Prepare to roll
```
We clear the accumulator:

\[ \text{LDA} \ #0 \]

then we roll in the bit to the correct position:

\[ \text{SBLOOP} \quad \text{ROL A} \]
\[ \text{DEX} \]
\[ \text{BPL SBLOOP} \]

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:

\[ \text{INY} \quad \text{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:

\[ \text{CPY} \ #9 \quad \text{Set carry for Port B} \]
\[ \text{RTS} \]

**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:

\[ \text{LIGHTR} \quad \text{JSR RANDOM} \]

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:

```assembly
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:

```assembly
BCS LL0

We will assume that Port A must be used. The new bit will be added to the display by ORing it into Port A:

```assembly
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:

```assembly
TYA
RTS

In case Port B is used, the sequence is identical:

```assembly
LL0 ORA PORTB
STA PORTB
TYA RTS

Restore value

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:

```assembly
BLINKR JSR RANDOM
SETMASK JSR SETBIT

Get random number
**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
ADVANCED 6502 PROGRAMMING

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.
DELAY Subroutines

Two delay loops are used by this program: DELAY, which provides approximately a half-second delay and DELAY2, which provides twice this delay or approximately one second. Index registers X and Y are each loaded with the value "FF." A two-level nested loop is then implemented:

```
DELAY2   JSR DELAY
DELAY    LDA #$FF
         TAY
D0       TAX
D1       DEX
         LDA #$FF
         BNE D1
         DEY
         BNE D0
         RTS
```

**Exercise 10-5:** Compute the exact duration of the DELAY subroutines.

**Interrupt Handler**

The interrupt routine is used to blink LEDs on the board, using MASKA and MASKB, every time that the timer generates an interrupt. No registers are changed. The operation of this routine has been described in the preceding chapter:

```
PHA
LDA PORTA
EOR MASKA
STA PORTA
LDA PORTB
EOR MASKB
STA PORTB
LDA TILL
PLA
RTI
```

**SUMMARY**

This program was more complex than most, despite the simple strategy
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 of the game. If you would like to have a more random value in RND at the beginning of the game, can you 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 EN0" 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?

```
; BLACKJACK PROGRAM
ACCESS = $BB98
INTVECL = $A67E
INTVECH = $A67F
IER = $A00E
ACR = $A00B
T1LL = $A004
T1CH = $A005
D0RA = $A003
D0RB = $A002
PORtA = $A001
PORtB = $A000
M0ASKA = $C2
M0ASKB = $C3
CHIPS = $C1
DONE = $C0
PHAND = $C4
CHAND = $C5
TEMP = $C6
RND = $C7
WHOWON = $C0
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-REPETITIVE. NUMERICAL TOTALS ARE KEPT IN ZERO PAGE LOCATIONS '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 'WHOWON' ARE STATUS FLAGS TO DETERMINE END OF GAME AND WHO WON THE CURRENT HAND.
```

--- Fig. 10.12: Blackjack Program ---
PROGRAM STARTS BY INITIALIZING THE TIMER AND THE INTERRUPT VECTOR. THE OUTPUT PORTS ARE TURNED ON, AND THE STATUS FLAGS ARE CLEARED.

```
0200: 20 B6 B8 BLJACK JSR ACCESS ; UNPROTECT SYSTEM MEMORY
0203: A9 EA LDA #$EA ; LOAD LOW INTERRUPT VECTOR
0205: BB 7E A6 STA INTRVL
0208: A9 03 LDA #$03 ; LOAD HIGH INTERRUPT VECTOR
020A: BB 7F STA INTRCH
020D: A9 7F LDA #$7F ; CLEAR TIMER INTERRUPT ENABLE
0210: BB 0E A0 STA IER
0212: A9 0C LDA #$0C ; ENABLE TIMER 1 INTERRUPT
0214: BB 0E A0 STA IER
0217: A9 40 LDA #$40 ; PUT TIMER 1 IN FREE RUN MODE
0219: BB 0B A0 STA ACR
021C: A9 FF LDA #$FF
021E: BB 04 A0 STA TILL ; SET LOW LATCH ON TIMER 1
0221: BB 05 A0 STA TICH ; SET HIGH LATCH & START TIMER
0224: 58 CLI ; ENABLE PROCESSOR INTERRUPTS
0225: BB 03 A0 STA DDRA ; SET LED PORTS TO OUTPUTS
0228: BB 02 A0 STA DDRB
022B: BB 00 A0 CLD
022C: A9 05 LDA #$5 ; SET PLAYER’S SCORE TO 5
022E: BB 05 C1 STA CHIPS
0230: A9 00 LDA #0 ; CLEAR DONE FLAG
0232: BB 05 C0 STA DONE
```

NEW HAND: DISPLAY IS CLEARED, BOTH HANDS ARE SET WITH START VALUES, AND THE CORRESPONDING LED’S ARE SET.

```
0234: BB 05 C2 START STA MASKA ; CLEAR BLINKER MASKS; IT IS
0236: BB 05 C3 STA MASKB
0238: BB 01 A0 STA PORTA ; CLEAR LED’S
023B: BB 00 A0 STA PORTB
023E: BB 05 CD STA WIDNOW ; CLEAR FLAG FOR HAND
0240: 20 0F 03 JSR BLINKR ; SET RANDOM BLINKING LED
0243: BB 05 C4 STA PHAND
0245: 20 F7 02 JSR LIGHTR ; SET A STEADY RANDOM LED
0248: BB 05 C5 STA CHAND ; STORE COMPUTER’S HAND
```

KEY INPUT: ‘A’ IS A HIT, ‘C’ IS COMPUTER TURN. ALL OTHERS ARE IGNORED.

```
024A: 20 00 01 ASK JSR GETKEY ; GET A KEY INPUT
024D: C9 0A CMP #$0A ; DOES PLAYER WANT A HIT?
024F: F0 07 BEQ HITPLR ; YES, BRANCH
0251: C9 0C CMP #$0C ; IS IT ‘COMP TURN’ KEY?
0253: F0 12 BEQ DEALER ; YES
0255: 4C 0A 02 JMP ASK ; BAD KEY, TRY AGAIN
```

```
0258: 20 0F 03 HITPLR JSR BLINKR ; SET A RANDOM LED
025B: 18 CLC
025C: 45 C4 ADC PHAND ; TALLY PLAYER’S HAND
025E: BB 05 C4 STA PHAND
0260: C9 0E CMP #$14 ; CHECK HAND
0262: 90 E6 BCC ASK ; IS <313, OK
0264: 4C 07 02 JMP LOSE ; DUSTED, GO TO LOSE ROUTINE
```

```
0267: 20 5D 03 DEALER JSR DELAY ; DELAY EXECUTION OF ROUTINE
026A: BB 05 C5 LDA CHAND ; IS COMP OVER HOUSE LIMIT?
026C: C9 0A CMP #$10 ; YES, FIGURE WINNER
026E: BB 0F 0F BCS WINNER
0270: 20 F7 02 JSR LIGHTR ; NO, SET RANDOM LED
0273: 18 CLC
```

--- Fig. 10.12: Blackjack Program (Continued) ---

213
ADVANCED 6502 PROGRAMMING

0274: 65 C5 ADC HAND $TALLY COMPUTER'S HAND
0275: 85 C5 STA HAND
0276: C9 0E CMP #14 $IS HAND <=13?
0277: 90 EB BCC DEALER $YES, ANOTHER HIT?
0278: 4C 92 02 JMP WIN $BUSTED, PLAYER WINS

$FIGURE WINNER: 'WIN' AND 'LOSE' TALLY SCORE.
HAND DETERMINE IF THE Player HAS WON OR LOST
THE GAME. THE 'WHOWON' FLAG IS SET TO SHOW WHO
WON THE PARTICULAR HAND. IF THE HANDS ARE
EQUAL, NOTHING IS AFFECTED.

027F: A5 C5 WINNER LDA HAND $COMPARE HANDS
0280: C5 C4 CMP PHAND
0281: F0 19 BEQ SCORER $ARE EQUAL, NO CHANGE
0282: 90 0B BCC WIN $PLAYER'S HAND GREATER
0283: C6 CD DEC WHOSON $LOSE ROUTINE
0284: C6 C1 DEC CHIPS $TALLY SCORE
0285: D0 11 BNE SCORER $IS PLAYER BROKE?
0286: C6 C0 DEC DONE $YES, SET END OF GAME FLAG: LOSE
0287: 4C 9E 02 JMP SCORER
0288: E6 CD WIN INC WHOSON $WIN ROUTINE
0289: E6 C1 INC CHIPS $TALLY SCORE
028A: A5 C1 LDA CHIPS $ADD WINNINGS
028B: C9 0A CMP #10 $IF CHIPS=10, SET END OF GAME FLAG
028C: D0 02 BNE SCORER
028D: E6 C0 INC DONE $SET END OF GAME FLAG: WIN

$DISPLAY SCORE BY LIGHTING 1 OF 10 LED'S. THE
BOTTOM ROW OF LED'S IS SET TO SHOW WHETHER THE PLAYER
FOR THE COMPUTER WON THE HAND. THE DISPLAY IS HELD
THERE, THEN A TEST IS MADE FOR AN END OF GAME CONDITION
IF SUCH A CONDITION EXISTS, THE LED'S ARE
SET ACCORDINGLY, AND THE PROGRAM IS TERMINATED.
IT IS ASSUMED THAT THE ADDRESS OF THE MONITOR IS
IN THE STACK.

0290: 20 00 01 SCORER JSR GETKEY $HOLD LAST STANDINGS OF CARDS
0291: A9 00 LDA #0 $CLEAR LED'S
0292: B5 C2 STA MASKA
0293: B5 C3 STA MASKB
0294: 8D 01 A0 STA PORTA
0295: 8D 00 A0 STA PORTB
0296: A6 C1 LDX CHIPS
0297: F0 18 BEQ ENDER $ADJUST SO SUBROUTINE SETS
0298: CA DEX $THE RIGHT LED
0299: BA TXA
029A: B3 12 03 JSR SETMASK

029B: A5 C0 LDA WHOSON $SEE WHO WON HAND
029C: BF F0 BEQ ENDER $TIE- DO NOT AFFECT LED'S
029D: 30 05 LDA #$05 $PLAYER WIN- SET THREE LEFT LED'S
029E: A9 0E LDA #$0E $PLAYER WIN- SET THREE LEFT LED'S
029F: 4C C3 02 JMP SCO
02A0: A9 80 SC LDA #$B0 $PLAYER LOST- SET THREE RIGHT LED
02A1: 0D 00 A0 SCS ORA PORTB $SET LED PORT
02A2: BD 00 A0 STA PORTB
02A3: 20 5A 03 ENDER JSR DELAY2 $HOLD DISPLAY

02A4: A5 C0 LDA DONE $CHECK FOR END OF GAME CONDITION
02A5: D0 03 BNE ENO
02A6: 4C 34 02 JMP START $ZERO, START NEW HAND
02A7: 10 06 ENO BPL ENI $NEXT, WIN CONDITION
02A8: A9 8E LDA #$8E $SET SOLID ROW LEDS
02A9: BD 00 A0 STA PORTB
02AA: 60 RTS $RETURN TO MONITOR

Fig. 10.12: Blackjack Program (Continued)
<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>02E8</td>
<td>A9 FF</td>
<td>ENI</td>
</tr>
<tr>
<td>02D0</td>
<td>B5 C2</td>
<td>LDA #$FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STB MASK</td>
</tr>
<tr>
<td>02D8</td>
<td>A9 01</td>
<td>STA #$01</td>
</tr>
<tr>
<td>02E1</td>
<td>B5 C3</td>
<td>STA MASKB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
</tr>
<tr>
<td>02E4</td>
<td>AB</td>
<td>SETBIT TAY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;SAVE LOGICAL NUMBER</td>
</tr>
<tr>
<td>02E5</td>
<td>C9 08</td>
<td>CMP $8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;BRACKET 0-7 VALUE</td>
</tr>
<tr>
<td>02E7</td>
<td>90 02</td>
<td>BCC SB0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;...SUBTRACT IF &gt;7</td>
</tr>
<tr>
<td>02E9</td>
<td>E9 08</td>
<td>SBC $8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;SET INDEX REG</td>
</tr>
<tr>
<td>02EB</td>
<td>AA</td>
<td>SB0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;MOVE BIT TO POSITION</td>
</tr>
<tr>
<td>02EC</td>
<td>38</td>
<td>SEC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;PREPARE BIT TO ROLL</td>
</tr>
<tr>
<td>02ED</td>
<td>A9 00</td>
<td>LDA $0</td>
</tr>
<tr>
<td>02EF</td>
<td>2A</td>
<td>SBLLOOP ROL A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;MAKE Y NUMERICAL, NOT LOGICAL</td>
</tr>
<tr>
<td>02F0</td>
<td>CA</td>
<td>DEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;PREPARE BIT TO ROLL</td>
</tr>
<tr>
<td>02F1</td>
<td>10 FC</td>
<td>DPL SBLLOOP</td>
</tr>
<tr>
<td>02F3</td>
<td>CB</td>
<td>CPY $9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;SET CARRY. FOR PORTB C=1</td>
</tr>
<tr>
<td>02F5</td>
<td>60</td>
<td>RTS</td>
</tr>
</tbody>
</table>

Fig. 10.12: Blackjack Program (Continued)
0321: 9B      TYA
0322: 60      RTS

; GENERATES A RANDOM NUMBER FROM 0 TO 9 THAT IS NOT
; THE NUMBER OF AN LED ALREADY SET. RESULT IS IN ACC ON
; EXIT.

0323: 20 47 03 RANDOM JSR RANDER ; GET 0-255 NUMBER
0325: 29 0F      AND #$0F      ; MASK HIGH NIBBLE
0326: C9 0A      CMP $10      ; BRACKET 0-9
0327: 32A1 80 7F BCS RANDM
032C: 20 44 02 JSR SEBBIT ; SET BIT IN POSITION
032E: 85 C6      STA TEMP ; SAVE IT
0330: 00 08      BCS RNO ; DETERMINE PORT A OR B
0332: A5 C2      LDA MASKA ; COMBINE PORT AND MASK
0334: 00 01 0A ORA PORTA
0336: 33B8 4C 40 03 JMP RNI
033B: A5 C3      RNO LDA MASKB ; COMBINE PORT AND MASK
033D: 00 00 0A ORA PORTB
033F: 25 C6      RNI AND TEMP ; LOOK AT SPECIFIC BIT
0341: D0 DF      BNE RANDOM ; IF BIT SET ALREADY, TRY AGAIN
0343: 8B      DEY ; MAKE Y LOGICAL
0345: 9B      TYA ; EXIT WITH VALUE IN ACCUMULATOR
0346: 60      RTS

; GENERATES A RANDOM NUMBER FROM 0-255. USES NUMBERS
; i+a+b+c+d+e+f stored as RND through RND+5. Adds RHEFF1
; and puts result in A, then shifts A to B, B to C, etc.
; RANDOM NUMBER IS IN ACCUMULATOR ON EXIT.

0347: 3B      RANDER SEC ; CARRY ADDS 1
0348: A5 CB      LDA RND+1 ; ADD B+D+F
034A: 65 CB      ADC RND+4
034C: 65 CC      ADC RND+5
034E: A5 C7      STA RND
0350: A2 04      LDX $4 ; SHIFT NUMBERS DOWN
0352: B5 C7      LDA RND+1,X
0354: 95 CB      STA RND+1,X
0356: CA      DEX
0357: 10 F9      BPL RDLOOP
0359: 60      RTS

; DELAY LOOP: DELAY2 IS SIMPLY TWICE THE TIME DELAY
; FOR DELAY. GIVEN LOOP IS APPROX .5 SEC. DELAY.

035A: 20 5D 03 DELAY2 JSR DELAY
035D: A9 FF      DELAY LDA $$FF ; SET VALUE FOR LOOPS
035F: A8      TAY
0360: 00      D0 TAX
0361: CA      D1 DEX
0362: A9 FF      LDA $$FF
0364: D0 FF      BNE D1
0366: 8B      DEY
0367: D0 F7      BNE D0
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.
; NO REGISTERS ARE CHANGED, AND THE INTERRUPT
; FLAG IS CLEARED BEFORE EXIT.

03EA: 4B      PHA ; SAVE ACCUMULATOR
03EB: AD 01 A0 LDA PORTA ; COMPLEMENT PORTS WITH MASKS

---

Fig. 10.12: Blackjack Program (Continued)
**Fig. 10.12: Blackjack Program (Continued)**
11. Artificial Intelligence
(Tic-Tac-Toe)

INTRODUCTION

This chapter presents the complete design of a complex algorithm that solves the strategy and implementation problems of the Tic-Tac-Toe game. This is a long program using sophisticated evaluation techniques, table look-up algorithms, as well as complex data structures such as chained lists. It deserves a close examination and will bring you to a true competence level when programming the 6502.

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.

![Figure 11.3: Our First Move](image)

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.

![Figure 11.4: Second Computer Move](image)

![Figure 11.5: After the Computer's Third Move](image)
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.

![Figure 11.6: After the Computer's Fourth Move](image)

![Figure 11.7: After the Computer's Fifth Move](image)

**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
right and we move into position 9. The computer responds by moving to square 1, thus preventing us from completing a diagonal. This situation is shown in Figure 11.10. We must prevent the computer from completing a winning row on top; therefore we occupy position 2. The computer responds by occupying position 8. This is shown in Figure 11.11. We make our final move to square 7 to finish the game. This is a draw: we did not beat the computer.
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.

Fig. 11.14: Move 3

Fig. 11.15: "We Win!"

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 × 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
Fig. 11.16: The Six Combinations

Fig. 11.17: Evaluation Grid
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.

![Fig. 11.18: Test Case 1](image)

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
Fig. 11.21: Evaluating the Vertical Potential

Fig. 11.22: Evaluating the Diagonal Potential

Fig. 11.23: The Final Potential
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.

![Evaluation Grid for "O"

Fig. 11.24: Evaluation for "O"

231
Fig. 11.25: Potential Evaluation

Fig. 11.26: Move for Highest Score

Fig. 11.27: Finishing the Game
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.

![Tic-Tac-Toe game setups](image)

**Fig. 11.28: An Alternative Ending for the Game**

*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
Fig. 11.29: Test #1 Evaluated for "X"

Fig. 11.30: Test #1 Evaluated for "O"

Fig. 11.31: Test #2
“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.

![Diagram of a tic-tac-toe board with moves indicated]

**Fig. 11.32: Moving to the Center**

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, using only a two-ply analysis technique to design a simple chess game would not make it very interesting or very good. It would be necessary to use three-ply, four-ply or even more detailed analysis in order to make the game challenging.

If it is not possible to push the evaluation to a sufficient depth, the algorithm must be equipped with specific procedures that can detect special cases. This is the case with _ad hoc_ programming, which can be considered "unclean" but actually results in a much shorter program and/or a lesser memory requirement. In other words, if the special situations in a game can be recognized in advance, then it is
possible to write a special-purpose program which will take these situations into account. The resulting program will usually be shorter than the completely general one. This type of program, however, can only be constructed if the programmer has an excellent initial understanding of the game.

In the game of Tic-Tac-Toe, the number of combinations is limited. This makes it possible to examine all possible combinations that can be played on the board and to devise a procedure that takes all of these cases into account. Since we are primarily limited here by the amount of available memory, we will construct an ad hoc algorithm that fits within 1K of memory. Alternative techniques will be proposed as exercises.

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
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,"

240
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

Fig. 11.40: Board Analysis Flowchart (Continued)
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 illus-
tion, 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. 11.43: 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.
<table>
<thead>
<tr>
<th>COMPUTER</th>
<th>PLAYER</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>(DRAW)</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(DRAW)</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(DRAW)</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>(DRAW)</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>(DRAW)</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>(DRAW)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>(LOSS)</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

**Fig. 11.44: Actual Game Sequences**

246
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 having provided 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.
Fig. 11.45: Tic-Tac-Toe Flowchart
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.
11. Next, the computer's move is displayed.
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-
quence 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.
Fig. 11.46: Tic-Tac-Toe Row Sequences in Memory
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
Fig. 11.47: Tic-Tac-Toe: Low Memory
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A000</td>
<td>PORT1B</td>
</tr>
<tr>
<td>A001</td>
<td>PORT1A</td>
</tr>
<tr>
<td>A002</td>
<td>DDR1B</td>
</tr>
<tr>
<td>A003</td>
<td>DDR1B</td>
</tr>
<tr>
<td>A004</td>
<td>T1LL</td>
</tr>
<tr>
<td>A005</td>
<td>T1CH</td>
</tr>
<tr>
<td>A006</td>
<td>UNUSED</td>
</tr>
<tr>
<td>A007</td>
<td></td>
</tr>
<tr>
<td>A008</td>
<td></td>
</tr>
<tr>
<td>A009</td>
<td></td>
</tr>
<tr>
<td>A00A</td>
<td></td>
</tr>
<tr>
<td>A00B</td>
<td>ACR</td>
</tr>
<tr>
<td>A00C</td>
<td></td>
</tr>
<tr>
<td>A00D</td>
<td></td>
</tr>
<tr>
<td>A00E</td>
<td>IER</td>
</tr>
<tr>
<td>A67E</td>
<td>IRQVL</td>
</tr>
<tr>
<td>A67F</td>
<td>IRQVH</td>
</tr>
<tr>
<td>ACO0</td>
<td>PORT3B</td>
</tr>
<tr>
<td>ACO1</td>
<td>UNUSED</td>
</tr>
<tr>
<td>ACO2</td>
<td>DDR3B</td>
</tr>
</tbody>
</table>

*Fig. 11.48: Tic-Tac-Toe: High Memory*
will be raised by one point. Each time that the player loses, it will be
decremented by one point. It is initially set at the value 12 decimal:

```
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 loca-
tions 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  Low vector
LDA $>INTVEC
STA IRQVH  High vector
```

As usual, the interrupt-enable register must first be cleared, then the appropriate interrupt must be enabled:

```
LDA #$7F
STA IER    Clear register
LDA #$C0
STA IER    Enable interrupt
```

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:

```assembly
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:

```assembly
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:

```assembly
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:

```assembly
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  LDY #7
TSTLP   CMP ROWSUM,4
       BEQ WIN
       DEY
       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:
WIN CMP#12
BEQ INTDN

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

260
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
ADVANCED 6502 PROGRAMMING

A delay must be provided:

DLY JSR DELAY

We are now ready to start a new game with the new intelligence level of the computer:

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:

LDA #30
STA 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:

LDA INTEL
CMP #$0F
BEQ GTMSK
INC 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:

COMPMV INC PLAYR
JSR DELAY

The computer move is determined by the ANALYZ routine described
below:

    JSR ANALYZ

The computer's move is entered as a "four" at the appropriate location on the board:

    LDA #04
    JSR UPDATE

Next, we check all of the rows for the possibility of a computer win, i.e., for a total of "twelve":

    LDA #12
    WINTST
    LDY #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."

    ANALYZ
    LDA #0
    STA 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:
LDA #08

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:

\[
\begin{align*}
\text{LDA} & \ #04 \\
\text{LDX} & \ #02 \\
\text{JSR} & \ \text{FINDMV} \\
\text{BNE} & \ \text{DONE}
\end{align*}
\]

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:

\[
\text{JSR RANDOM}
\]

We truncate the random number to its right byte so that it does not exceed fifteen:

\[
\text{AND} \ #0F
\]

and we compare it to the current IQ of the computer:

\[
\begin{align*}
\text{CMP} & \ \text{INTEL} \\
\text{BEQ} & \ \text{OK} \\
\text{BCS} & \ \text{RNDMV}
\end{align*}
\]

If the random number is higher than the IQ level stored in INTEL, we branch to RANDMV 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:

\[
\begin{align*}
\text{OK} & \ \text{LPX MOVNUM} \\
& \ \text{CPX} \ #1
\end{align*}
\]
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 ad-
dress 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:
Fig. 11.49: FINDMV Flowchart
FINDMV STX TEMP2
STA TEMP1

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:
UPDATE     DEX

The value must now be stored in the appropriate location of the GMBRD table which contains the internal representation of the board:

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 computer’s move. It does this by examining the code contained in the accumulator. 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:

CMP #04
BEQ NOBLNK

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:

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:

ORA LTMSKL
STA LTMSKL

If the carry was “zero” upon completion of LIGHT, one of the bits zero through seven had been set and we are done:

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:

LDA #01
STA LTMSKH

At this point, the LED masks are properly configured and we can give the order to light the LEDs:

NOBLNK    JSR LEDLTR

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:

LDX #7
ADDROW    CLC

The first square of row eight is examined first:

LDY RWPT1,X

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:

LDA GMBRD,Y

The GMBRD table is accessed using index register Y as a displacement. Note that the two instructions shown above implement a two-level indexing operation. This is a most efficient data retrieval technique. 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 ADDRLOW

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 in-
deed 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":

277
ADVANCED 6502 PROGRAMMING

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 #$FF
DL1   LDX #$FF
DL2   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 T1LL
   PLA
   RTI
```

*Exercise 11-7: Notice the LDA T1LL 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 T1LL, 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.

---

**Fig. 11.50: Tic-Tac-Toe Program**
; \VARIABLE STORAGES:

0037  0011  07
0037  0012  08
0037  0013  02
0037  0014  05
0037  0015  08
0037  0016  08
0037  0017  06
0038  0018
0039  0018
0040  0018
0041  0018 CLKBST: 1ST LOC. TO BE CLEARED BY 'INIT'.
0042  0019 OMNED: GAME BOARD: PLAYER'S POSITIONS ON
0043  0020 1BOARD AS $01=PLAYER, $04=COMPUTER.
0044  0021 SOSTAT: SQUARE'S TACTICAL STATUS.
0045  0022 ROWSUM: SUM OF VALUES OF SQUARES IN
0046  0023 ROW WHERE 1-PLAYER.
0047  0024 14=COMPUTER, 0-EMPTY.
0048  0025 RNDSCR: $16 GEN. SCRATCHPAD.
0049  0026 TEMP1: $17
0050  0027 TEMP2: $18
0051  003A MOVNUM: NUMBER OF CURRENT MOVE.
0052  003B PLAYR: WHO'S TURN IT IS.
0053  003C LMTSKN: HIGH ORDER Blink Mask for LED'S
0054  003B LMTSK: LOW ORDER SAME.
0055  003E DUR: DURATION FOR TONES.
0056  003F FREQ: FREQUENCY OF TONES.
0057  0040 CLREND: CLEAR LOC TO BE CLEARED BY 'INIT'.
0058  0040 ODMSK: PRODUCT OF RANDOM MOVE
0059  0041 INTEL: GENERATOR ODD TO PICK CORNER.
0060  0041 INTEL: INTELLIGENCE QUOTIENT.
0061  0042
0062  0042 ; \***** MAIN PROGRAM *****
0063  0042
0064  0042
0065  0050
0066  0050 START LDA $12
0067  0051 STA INTEL SET I.O. AT 75%
0068  0052 RESTR JSR INIT INITIALIZE PROGRAM.
0069  0053 0027 0001 JSR GETKEY GET FIRST MOVE DETERMINER.
0070  0054 002A C9 0F CMP $9 IS IT 'F'?
0071  0055 002C D0 04 BNE PLAYF
0072  0056 002E A9 01 LDA $01 YES, PLAYER FIRST.
0073  0057 0033 18 85 STA PLAYR
0074  0057 0012 E6 3A PLAYLP INC MOVNUM COUNT THE MOVES.
0075  0058 0024 A5 38 LDA PLAYR WHO'S TURN?
0076  0059 0021 F0 0E BNE COMPVV IF O, COMPUTER'S MOVE.
0077  0059 0021 C6 3B DEC PLAYR PLAYER'S TURN, COMPUTER NEXT.
0078  0059 0024 20 80 03 JSR PLRMV GET PLAYER'S MOVE.
0079  0059 0021 8F 01 LDA $01 STORE PLAYER'S PIECE.
0080  0059 0021 20 40 03 JSR UPDATE PLAY IT, AND UPDATE ROWSUMS.
0081  0059 0022 A9 03 LDA $03 LOAD PATTERN FOR WIN SEARCH.
0082  0059 0021 20 00 0F BNE WINTST CHECK FOR WIN.
0083  0059 0021 E6 3B COMPVV INC PLAYR COMPUTER'S TURN, PLAYER NEXT.
0084  0059 0021 20 64 03 JSR DELAY TIME FOR COMPUTER TO 'THINK'.
0085  0059 0021 20 9B 02 JSR ANALYZ FIND COMPUTER'S MOVE.
0086  0059 0021 8F 04 LDA $04 STORE COMPUTER'S PIECE.
0087  0059 0021 20 40 03 JSR UPDATE PLAY IT.
0088  0059 0021 A9 0C LDA $12 LOAD PATTERN FOR WIN SEARCH.
0089  0059 0021 35 07 WINTST LDY $7 LOOP 7 X TO CHECK ROWSUMS
0090  0059 0021 27 0A 00 TSTL: CMP ROWSUMS Y FOR WINNING PATTERN.
0091  0059 0021 35 0F REY WIN IF PATTERN FOUND.
0092  0059 0021 3C 88 REY LOOP AND
0093  0059 0021 3D 0F BPL TSLP TRY AGAIN.
0094  0059 0021 2F 3A LDA MOVNUM IF MOVE NUMBER = 9.
0095  0059 0021 2D C9CMP $9 THEN GAME IS TIE.
0096  0059 0021 2C DD BNE PLAYF KEEP PLAYING IF NOT.
0097  0059 0021 2F 0D LDA $0F SET ALL LIGHTS TO BLINKING.
0098  0059 0021 26 3D STA LMTSKL
0099  0059 0021 26 3C STA LMTSK
0100  0059 0021 2D 4A BNE PLAYF KEEP THEM BLINKING A WHILE.
0101  0059 0021 2F 04 CMP $12 COMPUTER WIN?
0102  0059 0021 2F 00 F0 INTDN IF YES: I.O. DOWN.

Fig. 11.30: Tic-Tac-Toe Program (Continued)
003 0251 A9 1E  LDA #30  LOAD FRED. CONST FOR WIN TONE.
004 0253 B5 3F  STA FRED
005 0255 A5 41  LDA INTEL
006 0257 C9 0F  CMP #00  F.I.O. AS HIGH AS POSSIBLE?
007 0259 B5 0E  BEQ GTMSK  IF YES, DON'T CHANGE IT.
008 025B E6 41  INC INTEL  RAISE I.O.
009 025D D0 0A  BNE GTMSK  IF SO FLASH ROW.
010 025F A9 FF  INTH redirected LDA #FF  LOAD FRED. CONST. FOR LOSE TONE.
011 0261 B5 3F  STA FRED
012 0263 A5 41  LDA INTEL  IF I.O. - 0?
013 0265 F0 02  BEQ GTMSK  IF YES, DON'T DECREMENT!
014 0267 C6 41  BEQ INTEL  IF I.O. DOWN.
015 0269 A9 00  GTMSK  LDA #0  CLEAR ALL I.E.S.
016 026B BD 01 A0  STA PORTA
017 026E BD 00 A0  STA PORTB
018 0271 B6 00  LDX RPTY  Y  SET BIT IN ACCUM. TO LIGHT
019 0273  HLED CORRESPONDING TO 1ST: SQUARE
020 0273  8N WINNING ROW.
021 0273  00F 03  JSR LEDTLR
022 0276 B6 00  LDX RPT3,Y  SET SECOND BIT.
023 0278 B6 40  JSR LEDTLR
024 027B B6 10  LDY WPT3,Y  GET 3RD BIT.
025 027E  00F 03  JSR LEDTLR
026 0280 AD 01 A0  LDA PORTA  MASK OUT UNNECESSARY BITS IN
027 0283  25 3D  AND LIKMK
028 0285 B5 3D  STA LIKMK
029 0287 AD 00 A8  LDA PORTI
030 028A B5 3C  AND LIKMK
031 028C B5 3C  STA LIKMKH
032 028E A9 FF  LDA #FF  SET WIN/LOSE TONE DURATION.
033 0290 B5 3E  STA DUR
034 0292 A5 3F  LDA FRED  SET FREQUENCY.
035 0294  20 AB 00  DLY JSR TONE  PLAY TONE.
036 0297  20 A4 03  DLY JSR DELAY  DELAY TO SHOW WIN OR TIE
037 029A 4C 04  02 JMP RETSTR  START NEW GAME, DON'T CHNG. I.O.
038 029D 029D  029D  029D  029D
039 029D  029D
040 029D  029D
041 029D  029D
042 029D  029D
043 029D  029D  ANALYZ  LDA #0  SET MASK THAT MAKES RANDOM MOVES
044 029F B5 40  STA ODDMSK  SET SIDES TO 0.
045 02A1 A9 08  LDA #06  CHECK FOR WINNING MOVE FOR
046 02A3 AD 03  LDX #03  COMPUTER.
047 02A5  20 04 03  JSR FINDMV
048 02A8 D0 59  BNE DONE  IF FOUND, RETURN.
049 02AA A9 02  LDA #02  CHECK FOR WINNING MOVE FOR
050 02AC A2 03  LDX #03  PLAYER.
051 02AE  20 04 03  JSR FINDMV
052 02B1 D0 50  BNE DONE  IF FOUND, RETURN.
053 02B3 A9 04  LDA #04  ICAN COMPUTER SET A TRAP?
054 02B5 A2 02  LDX #02
055 02B7  20 04 03  JSR FINDMV
056 02B9 D0 47  BNE DONE  IF YES, PLAY IT.
057 02BC  20 9A 00  JSR RANDOM  GET A RANDOM NUMBER...
058 02BF  29 0F  AND #0F  ...AND MAKE IT 0-15...
059 02C1 C5 41  CMP INTEL  IF FOR USE AS SCRAMBLE DETERMINER.
060 02C3 F0 02  BNE OK  IF BOTH ARE EQUAL, SKIP TEST
061 02C5 AB 28  BCS RANDMV  IF RAND > INTEL, PLAY A DUMB MOVE.
062 02C7 A6 3A  OK LDX MOUNM
063 02C9 E0 01  CPX #1  1ST MOVE?
064 02CB F0 25  BNE RANDMV  IF YES, PLAY ANY MOVE.
065 02CD E0 04  CPX #4  4TH MOVE.
066 02CF D0 0C  BNE TRAPCK  IF NOT, CONTINUE.
067 02D1 A2 06  LDX #6  LOAD INDEX TO 1ST DIAG. ROWSUM.
068 02D3 8A 05  TAX
069 02D4  B5 2A  CMP ROWSUM+X  CHECK IF 1ST DIAG. IS P-C-P
070 02DB F0 16  BNE ODDBRND  IF YES, PLAY SIDE.
071 02DB 4B 0A  INX
072 02DB B5 2A  CMP ROWSUM+X  CHECK NEXT DIAG. ROWSUM
073 02DB F0 11  BNE ODDBRND
074 02DD A9 01  TRAPCK LDA #1  CAN PLAYER SET A TRAP?
0175 020F A2 02  LDX $2
0176 02E1 20 04  JSR FNDMV  ;IF YES, PLAY BLOCK.
0177 02E4 D0 1D  BNE DONE  ;IF YES, PLAY BLOCK.
0178 02E6 A6 1C  LDX GMBRD+4  ;IS CENTER.
0179 02EB D0 08  BNE RNDMV  ;IS OCCUPIED?
0180 02EA A2 05  LDX $5  ;IND PLAY IT.
0181 02EC D0 15  BNE DONE
0182 02EE A9 01  ODDRND LDA $1  ;SET ODDMASK TO 1: SO
0183 02F0 B5 40  STA ODDMSK  ;MOVE WILL BE A SIDE.
0184 02F2 20 9A 00  RNDMV JSR RANDOM  ;GET RANDOM # FOR MOVE.
0185 02F5 29 0F  AND $4F  ;MAKE IT 0-15.
0186 02F7 05 40  ORA ODDMSK  ;MAKE ODD # IF CORNER HEADED.
0187 02F9 C9 09  CMP $9  ;NUMBER TOO HIGH?
0188 02FB D0 5F  BCS RNDMV  ;YES: GET ANOTHER.
0189 02FD AA 04  TAX
0190 02FE B5 18  LDA GMBRD,X  ;SPACE OCCUPIED?
0191 02F0 D0 0F  BNE RNDMV  ;YES: GET ANOTHER MOVE.
0192 0302 EB 0D  INX  ;INCREMENT X TO MATCH OUTPUT OF FNDMV.
0193 0303 60  DONE  RTS  ;RETURN W/ MOVE IN Y.
0194 0304 1  SUBROUTINE 'FIND MOVE' *******
0195 0304 1  FINDS A SQUARE MEETING SPECIFICATIONS
0196 0304 1  PASSED IN IN X AND Y.
0197 0304 1  INDEX REGISTER Y CONTAINS
0198 0304 1  MASK THAT, WHEN OR'ED WITH
0199 0304 1  NUMBER OF TIMES A SQUARE FITS ROWS WITH
0200 0304 1  ROWSUM IN ACCUM.: MUST YIELD A ONE
0201 0304 1  FOR SQUARE TO QUALIFY.
0202 0304 1
0203 0304 1  FNDMV STY TEMP  ;SAVE REGISTERS.
0204 0304 86 39
0205 0306 85 38  STA TEMP1  ;STX TEMP1
0206 0308 A9 00  LDA $0  ;CLEAR SQUARE STATUS REGISTERS.
0207 030A A0 08  LDY $8
0208 030C 99 21 00  CLRLP STA Sbsta:y
0209 030F 88  DEY
0210 0310 10 FA  BPL CLRLP
0211 0312 A0 07  LDY $7  ;LOOP 7X
0212 0314 A5 38  CHEKLP LDA TEMP1  ;DOES ROWSUM
0213 0316 D9 2A 00  CMP ROWSUM,Y  ;MATCH PARAMETER?
0214 0318 D0 0F  BNE NOCHEK  ;IF NOT: TRY NEXT.
0215 031A B6 00  LDX RNPT1,Y  ;CHECK 1ST SQUARE IN ROW.
0216 031B 20 39 03  JSR CNTSUB  ;INCREMENT ITS STATIF IT'S EMPTY.
0217 0320 B6 08  LDX RNPT2,Y  ;DO 2ND SQUARE.
0218 0322 20 39 03  JSR CNTSUB
0219 0325 B6 10  LDX RNPT3,Y  ;AND THIRD.
0220 0327 20 39 03  JSR CNTSUB
0221 0329 B8  NOCHEK DEY  ;TRY NEXT ROW.
0222 032B 19 E7  BPL CHEKLP
0223 032B D0 09  LDX $9  ;LOAD PARAMETER...
0224 032F A5 39  FNHTCH LDA TEMP2  ;LOAD PARAMETER...
0225 0331 35 20  AND Sbsta-1,X  ;SQUARE STATUS=AND(PARAM)=0?
0226 0333 D0 03  BNE FOUND  ;IF YES, PLAY X AS MOVE.
0227 0335 CA 04  BEX  ;INCREMENT AND TRY NEXT SOSTAT.
0228 0336 D0 F7  BNE FNHTCH
0229 0338 60  FOUND RTS
0230 0339 1  SUBROUTINE 'COUNTSUB' *******
0231 0339 1  INCREMENTS SOSTAT OF EMPTY SQUARES.
0232 0339 1
0233 0339 B5 19  CNTSUB LDA GMBRD,X  ;GET SQUARE.
0234 033B D0 02  BNE NOCNT  ;IF FULL, SKIP.
0235 033D F6 21  INC SOSTAT,X  ;INCREMENT SOSTAT
0236 033F 60  NOCNT RTS  ;DONE.
0237 0340 1  SUBROUTINE 'UPDATE' *******
0238 0340 1  PLAYS MOVE BY STORING CODE PASSED IN IN ACCUM.
0239 0340 1  HAT SQUARE SPECIFIED BY X REG.
0240 0340 1  ALSO LIGHTS/SETS BLINKING PROPER LED.
0241 0340 1  HAND COMPUTES ROWSUMS.
0242 0340 1
0243 0340 F9 EA  UPDATE DEX  ;DECREMENT MOVE TO MATCH INDEXING.
0244 0344 95 18  STA GMBRD,X  ;PLAY MOVE.

Fig. 11.50: Tic-Tac-Toe Program (Continued)
0247 0343 C9 04  
0248 0345 FO 00  
0249 0347 20 9B 03  
0250 034A  
0251 034A 05 3D  
0252 034C B5 3D  
0253 034E 90 04  
0254 0350 A9 01  
0255 0352 B5 3C  
0256 0354 20 6F 03  
0257 0357 A2 07  
0258 0359 18  
0259 035A B4 00  
0260 035C 89 18 00  
0261 035E A9 0A  
0262 0361 99 19 00  
0263 0364 B4 10  
0264 0366 79 18 00  
0265 0368 25 2A  
0266 036B CA  
0267 036C 10 EB  
0268 036E 60  
0269 036F  
0270 036F  
0271 036F  
0272 036F  
0273 036F  
0274 036F 20 9B 03  
0275 0372 00 01 0A  
0276 0376 B0 00 0D  
0277 037B 90 05  
0278 037A A9 01  
0279 037C BD 00 0D  
0280 037F 60  
0281 0380  
0282 0380  
0283 0380  
0284 0380  
0285 0380 A9 80  
0286 0382 85 3E  
0287 0384 69 10  
0288 0388 20 AD 00  
0289 0388 20 00 01  
0290 038C C9 0A  
0291 038E B0 F9  
0292 0390 AA  
0293 0391 F0 F6  
0294 0393 BB 17  
0295 0395 D0 F2  
0296 0397 60  
0297 0398  
0298 0398  
0299 0398  
0300 0398  
0301 0398  
0302 0398  
0303 0398  
0304 0399 86 3B  
0305 039A A9 00  
0306 039C 3B  
0307 039D 2A  
0308 039E CA  
0309 039F 10 FC  
0310 03A1 A6 3B  
0311 03A3 60  
0312 03A4  
0313 03A4  
0314 03A4  
0315 03A4 A0 FF  
0316 03A6 A2 FF  
0317 03A8 26 3E  
0318 03AA 66 3E  

; COMPUTER’S MOVE
: 0247 0343 C9 04  
: 0248 0345 FO 00  
: 0249 0347 20 9B 03  
: 0250 034A  
: 0251 034A 05 3D  
: 0252 034C B5 3D  
: 0253 034E 90 04  
: 0254 0350 A9 01  
: 0255 0352 B5 3C  
: 0256 0354 20 6F 03  
: 0257 0357 A2 07  
: 0258 0359 18  
: 0259 035A B4 00  
: 0260 035C 89 18 00  
: 0261 035E A9 0A  
: 0262 0361 99 19 00  
: 0263 0364 B4 10  
: 0264 0366 79 18 00  
: 0265 0368 25 2A  
: 0266 036B CA  
: 0267 036C 10 EB  
: 0268 036E 60  
: 0269 036F  
: 0270 036F  
: 0271 036F  
: 0272 036F  
: 0273 036F  
: 0274 036F 20 9B 03  
: 0275 0372 00 01 0A  
: 0276 0376 B0 00 0D  
: 0277 037B 90 05  
: 0278 037A A9 01  
: 0279 037C BD 00 0D  
: 0280 037F 60  
: 0281 0380  
: 0282 0380  
: 0283 0380  
: 0284 0380  
: 0285 0380 A9 80  
: 0286 0382 85 3E  
: 0287 0384 69 10  
: 0288 0388 20 AD 00  
: 0289 0388 20 00 01  
: 0290 038C C9 0A  
: 0291 038E B0 F9  
: 0292 0390 AA  
: 0293 0391 F0 F6  
: 0294 0393 BB 17  
: 0295 0395 D0 F2  
: 0296 0397 60  
: 0297 0398  
: 0298 0398  
: 0299 0398  
: 0300 0398  
: 0301 0398  
: 0302 0398  
: 0303 0398  
: 0304 0399 86 3B  
: 0305 039A A9 00  
: 0306 039C 3B  
: 0307 039D 2A  
: 0308 039E CA  
: 0309 039F 10 FC  
: 0310 03A1 A6 3B  
: 0311 03A3 60  
: 0312 03A4  
: 0313 03A4  
: 0314 03A4  
: 0315 03A4 A0 FF  
: 0316 03A6 A2 FF  
: 0317 03A8 26 3E  
: 0318 03AA 66 3E  

--- Fig. 11.50: Tic-Tac-Toe Program (Continued) ---
0319 03AC CA  DEY
0320 03AD DD F9  BNE DL2
0321 03AF 88  DEY  BNE DL1
0322 03B0 DD F4  RTS
0323 03B2 60  
0324 03B3  *
0325 03B3  
0326 03B3  
0327 03B3  ;THE BLINK MASKS HAVE ONES IN THEM ARE TURNED
0328 03B3  
0329 03B3  48  INTVEC PIA
0330 03B4 AD 01 A0  LDA PORT1A
0331 03B7 85 33  EOR LTMSKL
0332 03B9 AD 01 A0  STA PORT1B
0333 03BC AD 00 A0  LDA PORT1B
0334 03BF 85 3C  EOR LTMSKH
0335 03C1 BD 00 A0  STA PORT1B
0336 03C4 AD 04 A0  LDA TILL
0337 03C7 68  PLA
0338 03C8 40  RTI
0339 03C9  
0340 03C9  
0341 03C9  ;SUBROUTINE "INITIALIZE" 
0342 03C9  ;INITIALIZES PROGRAM.
0343 03C9  ;
0344 03C9  * = $50
0345 03C9  0050  INIT  LDA #0  ICLRSTORAGES.
0346 03C9  0052 A2 28  LDX @CLREM-CLRST
0347 03C9  0054 95 18  CLRALL STA CLRM+X
0348 03C9  0056 CA  DEY
0349 03C9  0057 19 FF  BPI CLRMALL
0350 03C9  0059 AD 04 A0  LDA TILL  ISET RANDOM NUMBER GENERATOR SEED.
0351 03C9  005C 85 33  STA RNDSCR+1
0352 03C9  005E 85 3A  STA RNDSCR+4
0353 03C9  0060 A9 FF  LDA #FF
0354 03C9  0062 BD 03 A0  STA DDR1A  ISET UP 1/0
0355 03C9  0065 BD 02 A0  STA DDR1B
0356 03C9  0068 BD 02 AC  STA DDR3B
0357 03C9  006B A9 00  LDA #0  ICLRST ORANS
0358 03C9  006D BD 00 A0  STA PORT1B
0359 03C9  0070 BD 00 A0  STA PORT1B
0360 03C9  0073  ISET UP TIMER FOR INTERRUPTS WHICH
0361 03C9  0073  BLINK LEDS.
0362 03C9  0073  20 86 8B  JBR ACCESS  IUSERPROTECT SYM1 SYSTEM MEMORY TO
0363 03C9  0076  -  ISET UP INTERRUPT VECTORS.
0364 03C9  0076  A9 B3  LDA #INTVEC  FLOAD LOW BYTE INTERRUPT VECTOR.
0365 03C9  0076  BD 7E A6  STA IRDV  ISTORE AT INTERRUPT VECTOR LOCATION.
0366 03C9  0076  A9 03  LDA $INTVEC  FLOAD HI BYTE INTERRUPT VECTOR.
0367 03C9  007C BD 7F A6  STA IRDVH  ISTORE.
0368 03C9  0080 A9 7F  LDA #7F  ICLEAR INTERRUPT ENABLE REGISTER.
0369 03C9  0082 BD 0E A0  STA IER
0370 03C9  0085 A9 C0  LDA #CO  IENABLE TIMER1 INTERRUPT.
0371 03C9  0087 BD 0E A0  STA IER
0372 03C9  0089 A9 40  LDA #40  IENABLE TIMER1 IN FREE-RUN MODE.
0373 03C9  008C BD 09 A0  STA ACR
0374 03C9  008F A9 FF  LDA ###  ISET LOW LATCH ON TIMER 1.
0375 03C9  0091 BD 04 A0  STA TILL  ISET HIGH LATCH START INTERRUPT COUNT.
0376 03C9  0094 BD 05 A0  STA TICH  IENABLE INTERRUPTS.
0377 03C9  0097 58  CLI
0378 03C9  0098 DB  
0379 03C9  0099 40  RTS
0380 03C9  
0381 03C9  
0382 03C9  ISETUP RANDOM GENERATOR LAST NEW
0383 03C9  IRANDOM NUMBER IN ACCUMULATOR.
0384 03C9  
0385 03C9  38  RANDOM SEC
0386 03C9  0092 A5 33  LDA RNDSCR+1
0387 03C9  0099 65 36  ADC RNDSCR+4
0388 03C9  009F 65 37  ADC RNDSCR+5
0389 03C9  00A1 85 32  STA RNDSCR
0390 03C9  00A3 A2 04  LDX #

---Fig. 11.50: Tic-Tac-Toe Program (Continued)---
ADVANCED 6502 PROGRAMMING

0391 0045 85 32 RNDLP LDA RNDSCR+X
0392 0047 95 33 STA RNDSCR+1,X
0393 0049 CA DEX
0394 00AA 10 F9 BPL RNDLP
0395 00AC 60 RTR
0396 00AD ;
0397 00AD ;
0398 00AD ;
0399 00AD ;
0400 00AD ;
0401 00AD ;
0402 00AD 85 3F TONE STA FRED
0403 00AF 89 FF LDA $FF
0404 00B1 BD 00 AC STA PORT3B
0405 00B4 A9 00 LDA $00
0406 00B6 A6 3E LDX DUR
0407 00BB A4 3F FL2 LDY FRED
0408 00BA 88 FL1 DEY
0409 00BB 18 CLR
0410 00BC 90 00 BEQ #+2
0411 00BE D0 FA BNE FL1
0412 00C0 49 FF EOR $FF
0413 00C2 BD 00 AC STA PORT3B
0414 00C5 CA DEX
0415 00C6 D0 F0 BNE FL2
0416 00CB 60 RTS
0417 00C9 .END

SYMBOL TABLE

SYMBOL VALUE

ACCESS 886 ACH A00B ADDRESA 0359 ANAYZ 0390
CHECKP 0314 CLRALL 0054 CLEND 0040 CLRPL 030C
CLKST 0018 CNTRUB 0339 CMCMV 0226 DDBR 0003
DDR1B A002 DDDR3B AC02 DELAY 03A4 DL1 03A6
DL2 0306 DLY 0297 DONE 0307 DUR 003F
FINDMV 0304 F1L 008A FL2 008B FMATCH 032F
FOUND 0338 FREQ 003F GETKEY 0100 GMDBD 0018
GMSK 0269 IER A00E INIT 0050 INTHN 025F
INTEL 0041 INTVEC 03B3 IROVH 047F IROVL 047E
KEYIN 03B9 LEDLTR 036F LIGHT 0399 LSMSK 003C
LIFSKL 003D LTRDN 037F MOVNUM 003A NOBLNK 0024
NCHER 032A NOCMT 033F ODMASK 0040 O2DNNV 02FF
OK 02C7 PLAYLP 0212 PLAYR 003B PLRNV 0380
PORT1A A001 PORT1B 0000 PORT3B AC00 RANDOM 009A
RESTRT 0204 RNDLP 0045 RNDMV 02F2 RNSCR 0032
ROWSUM 002A RWPST 0000 RWPST 0028 RWPST 0010
SHIFT 039D SSSTAT 0021 START 0200 TCH 0005
TILL 0404 TEMP1 003B TEMPO 0039 TONE 000D
TRAPCK 02DD TRLP 0237 UPDATE 0340 WIN 024D
END OF ASSEMBLY

Fig. 11.50: Tic-Tac-Toe Program (Continued)
### Appendix A

**6502 INSTRUCTIONS—ALPHABETIC**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>JSR</td>
<td>Jump to subroutine</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>LDA</td>
<td>Load accumulator</td>
</tr>
<tr>
<td>ASL</td>
<td>Arithmetic shift left</td>
<td>LDX</td>
<td>Load X</td>
</tr>
<tr>
<td>BCC</td>
<td>Branch if carry clear</td>
<td>LDY</td>
<td>Load Y</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch if carry set</td>
<td>LSR</td>
<td>Logical shift right</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch if result = 0</td>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>BIT</td>
<td>Test bit</td>
<td>ORA</td>
<td>Logical OR</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch if minus</td>
<td>PHA</td>
<td>Push A</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch if not equal to 0</td>
<td>PHP</td>
<td>Push P status</td>
</tr>
<tr>
<td>BPL</td>
<td>Branch if plus</td>
<td>PLA</td>
<td>Pull A</td>
</tr>
<tr>
<td>BRK</td>
<td>Break</td>
<td>PLP</td>
<td>Pull P status</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch if overflow clear</td>
<td>ROL</td>
<td>Rotate left</td>
</tr>
<tr>
<td>BVS</td>
<td>Branch if overflow set</td>
<td>ROR</td>
<td>Rotate right</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear carry</td>
<td>RTI</td>
<td>Return from interrupt</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear decimal flag</td>
<td>RTS</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear interrupt disable</td>
<td>SBC</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>CLV</td>
<td>Clear overflow</td>
<td>SEC</td>
<td>Set carry</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare to accumulator</td>
<td>SED</td>
<td>Set decimal</td>
</tr>
<tr>
<td>CPX</td>
<td>Compare to X</td>
<td>SEI</td>
<td>Set interrupt disable</td>
</tr>
<tr>
<td>CPY</td>
<td>Compare to Y</td>
<td>STA</td>
<td>Store accumulator</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement memory</td>
<td>STX</td>
<td>Store X</td>
</tr>
<tr>
<td>DEX</td>
<td>Decrement X</td>
<td>STY</td>
<td>Store Y</td>
</tr>
<tr>
<td>DEY</td>
<td>Decrement Y</td>
<td>TAX</td>
<td>Transfer A to X</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>TAY</td>
<td>Transfer A to Y</td>
</tr>
<tr>
<td>INC</td>
<td>Increment memory</td>
<td>TSX</td>
<td>Transfer SP to X</td>
</tr>
<tr>
<td>INX</td>
<td>Increment X</td>
<td>TXA</td>
<td>Transfer X to A</td>
</tr>
<tr>
<td>INY</td>
<td>Increment Y</td>
<td>TXS</td>
<td>Transfer X to SP</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>TYA</td>
<td>Transfer Y to A</td>
</tr>
</tbody>
</table>
### Appendix B

#### 6502 INSTRUCTION SET—HEX AND TIMING

<table>
<thead>
<tr>
<th>ANEMONIC</th>
<th>IMPLIED</th>
<th>ACCUM.</th>
<th>ABSOLUTE</th>
<th>ZERO PAGE</th>
<th>IMMEDIATE</th>
<th>ABS. X</th>
<th>ABS. Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(1)</td>
<td>OA</td>
<td>6D</td>
<td>65</td>
<td>69</td>
<td>7D</td>
<td>79</td>
</tr>
<tr>
<td>B</td>
<td>(1)</td>
<td>OE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>C</td>
<td>(2)</td>
<td>CE</td>
<td>6D</td>
<td>65</td>
<td>69</td>
<td>2D</td>
<td>79</td>
</tr>
<tr>
<td>D</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>F</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>OE</td>
<td>6D</td>
<td>65</td>
<td>69</td>
<td>2D</td>
<td>79</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>B</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
<tr>
<td>E</td>
<td>(2)</td>
<td>CE</td>
<td>2D</td>
<td>25</td>
<td>29</td>
<td>2D</td>
<td>89</td>
</tr>
</tbody>
</table>

(1) Add 1 to n if crossing page boundary
### APPENDIX

#### PROCESSOR STATUS CODES

| OP n | # | OP n | # | OP n | # | OP n | # | OP n | # | OP n | # | N | V | D | I | Z | C | MNEMONIC |
|------|---|------|---|------|---|------|---|------|---|------|---|---|---|---|---|---|---------|
| 81   | 6 | 2    | 71| 5    | 2 | 35   | 4 | 2    | 75| 4    | 2 | 16| 6 | 2 | 90 | 2 | 2 | A      |
| 21   | 6 | 2    | 31| 5    | 2 | 35   | 4 | 2    | 35| 4    | 2 | 16| 6 | 2 | 90 | 2 | 2 | D      |
| 0    | 2 | 2    | 30| 2    | 2 | 2    | 2 | 10   | 2 | 2    | 2 | 2 | 10 | 2 | 0    | 1   | 1 | 1      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | B      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | R      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | P      |
| 61   | 6 | 2    | 31| 5    | 2 | 55   | 4 | 2    | 55| 4    | 2 | 16| 6 | 2 | 90 | 2 | 2 | 1      |
| 1    | 1 | 2    | 11| 5    | 2 | 15   | 4 | 2    | 15| 4    | 2 | 1   |    |   | 1    | 0    | 0 | 0      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | C      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | L      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | S      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | T      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | C      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | E      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | A      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | 1      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | S      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | T      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | X      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | Y      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | A      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | S      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | T      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | X      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | 1      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | T      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | S      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | T      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | X      |
|      |   |       |   |       |   |       |   |       |   |       |   |   |    |   |     |     |    | Y      |

(2) Add 2 to n if branch within page
Add 3 to n if branch to another page

---

**289**
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
Blip 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
COUNTSUB, 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
Hexadecimal, 41
Hexguess Program, 63
IER, 171

290
INDEX

IFR, 171
Illegal key closure, 95
Index, 159
Indexed addressing, 37, 39, 122, 126
Initialization, 198
INITIALIZE, 279
Intelligence level, 252, 260
Interconnect, 4
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
JMP, 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
LIGHTR, 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
RANDEL, 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
RDNSCR, 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 tunes, 21
Siren, 100
Slot Machine, 99
Slot Machine Program, 113
Software filter, 175
Special decimal mode, 150
Spinner, 87
Spinner 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
VIA memory map, 66
Visual feedback, 163
WAIT, 98
Wheel pointer, 103, 120
WIN, 128
Win, 259
Win potential, 225
WINEND, 129