Advanced Macintosh BASIC Programming

Philip Calippe

This reference guide and thorough tutorial to Microsoft BASIC 2.0 shows you the advanced commands and techniques you need to create your own Macintosh applications. Includes software development utilities and an arcade-style game, ready to type in and use.

$16.95

Advanced Macintosh BASIC Programming

Philip Calippe

Contents

Foreword

You already know Microsoft BASIC on the Macintosh, but now you want to tap its full potential. You've studied the manual and you're ready for some practical applications.

This book is not a simplistic guide to elementary programming. Instead, *Advanced Macintosh BASIC Programming* is a tutorial and reference to the advanced commands, statements, and techniques of Microsoft BASIC 2.0. You'll learn how to tackle useful projects and how to get the results you want from your Macintosh and its BASIC.

You'll learn how to use various Macintosh peripherals—such as the disk drive and printer—and how to use the Mac's unique graphic capabilities. Many sample programs are included, illustrating concepts like creating and maintaining a mailing list, drawing business graphics, block graphic animation, creating functional dialog boxes with control buttons, and generating pull-down menus. Countless examples make the transition from intermediate programmer to advanced applications designer painless and worry-free.

Different Abilities, Different Approaches

To accommodate everyone—no matter what their expertise—this book takes three major approaches to illustrating the Macintosh's advanced capabilities.

- The first approach explores the Macintosh-specific features of Microsoft BASIC. With this background, you can utilize the rest of the book. Programming features covered only briefly in the Microsoft BASIC manuals are detailed here. You're also shown the BASIC commands to create programming tools.
- The second approach uses the concepts defined in the first section to create a set of software tools, which are generally added to programs as subroutines. Each tool evolves from a design requirement to a subroutine or subprogram. Where applicable, each of these tools will be defined by purpose and theory of operation, data structure implementation, coding, and example of usage. This mirrors the organizational process of creating applications.
- The third approach deals with programming utilities, which incorporate the tools already defined. Each utility is defined

in terms of purpose, design, and implementation—complete with code and sample application. Often utilities use themselves and each other in their creation. Together they significantly reduce the amount of time it takes to develop an application. The ultimate utility writes most of the application itself.

A programmer's most challenging task is writing a game. The last program included in the book is an arcade-action game which demonstrates the concepts of event interrupts and event processing.

In these pages you'll find a wealth of information—from programming techniques and tricks to practical software utilities. Armed with this information, you'll be able to write sophisticated programs, programs that truly make the most of your Macintosh.

All the example programs included in *Advanced Macintosh BASIC Programming* are ready to type in and run. All you need is your Macintosh and a copy of Microsoft BASIC 2.0. If you prefer not to type in the programs, however, you can order a 3-1/2-inch disk which includes all the programs in this book by calling toll-free 800-334-0868 (in NC, call 919-275-9809), or by using the coupon found in the back of this book.

INTRODUCTION

The Form of the Book

To make this book as easy to use as possible, we need to mention a few things before you begin. Anyone with a reasonable understanding of BASIC will find this section useful, though not essential, for comprehending the programs and program segments which are used as examples.

All BASIC keywords within the text appear in **boldface,** just as they would in the *List* window on your Macintosh screen. Programs are printed just as they show on the screen, with keywords also in boldface. That's one of the niceties of Microsoft BASIC on the Macintosh.

The terms *BASIC command* and *BASIC statement* are often used interchangeably, but they're not necessarily the same. To keep things from getting too confusing, let's define these terms.

- BASIC commands can be keywords like **PRINT, IF-THEN-ELSE**, and **CALL.**
- On the other hand, BASIC statements can be made from BASIC commands like **IF** X>0 **THEN PRINT** X.
- Thus, a command can be equated to a keyword, while a statement is a segment of programming code which may contain one or more commands.

Most elementary BASIC programs, no matter for which computer, use a small assortment of BASIC commands which typically include **PRINT, GOTO, GOSUB, INPUT**, and **IF-THEN-ELSE** within statements. You can program practically anything using just these commands and some algebraic expressions. However, less common commands are available, often machine dependent, which exploit machine-specific hardware features. Obviously, there are many such commands for the Macintosh.

To prepare for later chapters, some of these commands will be examined in detail and categorized into keyboard input and input processing commands, graphic commands, file commands, miscellaneous commands, and ROM routine calls. There are examples associated with most of the commands. It's an excellent idea for you to type in these examples—you'll find it far easier to understand how they function.

BASIC 2.0 requires the adjustment of the *List* and *Output* windows. When starting BASIC 2.0, a *List* window is provided; double click on its title bar to set it to full screen size. The same can be done with the *Output* window. The *Command* window is rarely required and, when it is, the default size is usually sufficient.

Some Terms

The following terms should be explained before we go into any detail about the BASIC commands.

Term	Meaning
⌘-C	The result of pressing the Command key (⌘) with the C key. This usually stops program execution.
⌘-S	The result of pressing the Command key (⌘) with the S key. This usually suspends program execution until another key is pressed.
⌘-period	The result of pressing the Command key (⌘) with the period key (.). This usually stops program execution.
Number	A numeric value usually greater than zero.
Prompt string	A message which appears with a request for input. It usually indicates to the user what input is desired. A typical message would be *Type your name and press RETURN:*. The message may be any string, but not a string variable.
String	A string is a list of characters such as *This is a string!!!*. A string is divided into two components when stored in memory. First is a length which ranges from 0 to 32767 characters; then comes the set of characters that make up the string. A string with five characters, such as <-+->, requires six bytes of storage in memory. A string of zero length looks like " " and is called a *null string* or *null character*. It has special uses, which will be discussed and demonstrated later.
String variable	A variable that may have a string assigned to it. String variables end with a dollar sign ($), for example, *A$* or *NAME$*.
Variable list	A list of variable names separated by commas (though it can conceivably be just a single variable). A variable list may look something like *NAME$, PHONE$, AMOUNT,* or *INDEX(I)*. Note that variable types may be mixed in a variable list and that arrays must be indexed.

CHAPTER

1
Keyboard and String Manipulation

1
Keyboard and String Manipulation

Most programs perform three general tasks: *input, input processing,* and *output.* Some programs may have no input or output—such as those used for benchmarking computer processing time when performing calculations.

Let's take a look at the first of these tasks, input. A set of BASIC commands handles input on the Macintosh, and since a good program provides an easy-to-use interface which checks the input, there's also a set of commands to assist in verifying the integrity of the input.

The most commonly used command for input is, strangely enough, **INPUT**. It has six forms of syntax:

INPUT *variable list*
INPUT;*variable list*
INPUT"*prompt string*";*variable list*
INPUT;"*prompt string*";*variable list*
INPUT"*prompt string*",*variable list*
INPUT;"*prompt string*",*variable list*

The first and simplest form of **INPUT** sends a question mark to the screen and waits for a keyboard response, which is ended with a press of the Return or Enter key (these two Macintosh keys don't always behave the same). Accidentally entered characters can be erased with the Backspace key. Certain characters cannot be entered—these restricted characters depend on the type of variables in the variable list. Numeric variables are delimited by either commas or semicolons; thus, neither of these characters can be entered as part of the input data. String variables use a comma as a delimiter, so commas are forbidden here. Command-period and other Command

3

keystrokes cannot be entered. If a nonnumeric character is entered as data for a numeric variable, a *?Redo from start* message will appear. If this message occurs, all the items of that **INPUT** statement must be reentered. There are two exceptions to this rule of nonnumeric entry: The characters *e* and *d* are interpreted as exponents for single- and double-precision entries, respectively. For example, entering *4e4* for a numeric variable would be the same as entering *40000*.

Although this form of **INPUT** is the simplest to use, it doesn't give the user any information about what to enter, how many entries to make, or even the nature of the entries. One way to get around this deficiency is to use a **PRINT** command prior to the **INPUT**.

The second form of **INPUT** differs from the first only with the addition of a semicolon before the variable list. This prevents Return and Enter from generating a linefeed and carriage return on the screen. This is useful to remember when programming data entry screens that cannot be destroyed by user input. Data entry screens are discussed in more detail later. Program 1-1 demonstrates the use of the added semicolon in **INPUT**. For each response, enter *1* or *2*, according to the prompt. (Because spacing can be important in certain programs, if in doubt, type the statements with spaces as shown.)

Program 1-1. INPUT;

```
CLS
PRINT "Enter 1:";:INPUT A:PRINT "   Enter 2:";:INPUT B
PRINT "  Thank you, let's do it again."
PRINT "Enter 1:";:INPUT ;A:PRINT "   Enter 2:";:INPUT ;B
PRINT "  Thank you."
```

Figure 1-1. *When a semicolon is used before the variable list, Return does not affect the screen.*

```
▤□▥▥▥▥▥▥▥▥▥▥▥▥▥▥▥▥▥ Program 1-1 ▥▥▥▥▥▥▥▥▥▥▥▥▥▥▥▥
Enter 1:?  1
  Enter 2:?  2
  Thank you, let's do it again.
Enter 1:?  1   Enter 2:?  2  Thank you.
```
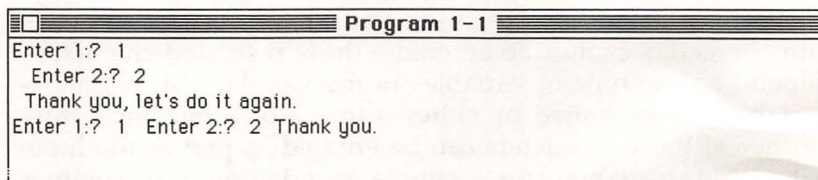
This program asks for the numbers 1 and 2 to be entered. This is done twice; the first time pressing the Return or Enter key generates a linefeed so that the next printed text appears on the line beneath the input. The second time the inputs are requested, pressing Return or Enter *doesn't* generate a linefeed due to the addition of a semicolon after the keyword **INPUT**. The cursor does not move down to the next line.

After clearing the display (**CLS**), the second line displays prompts before trying to get keyboard input. After values for the variables A and B have been entered, a message (*Thank you, let's do it again*) appears. Finally, the fourth line requests that the values for A and B be entered again.

The third form of **INPUT** differs from the first by the addition of a prompt string which is separated from the variable list by a semicolon. This format displays the prompt just prior to the normal **INPUT** question mark. In effect, the prompt string takes the place of a **PRINT** statement prior to the **INPUT**. It would be logical to make the prompt a message which tells the user what input is required.

Figure 1-2. *Prompts can be included with INPUT to eliminate the need for a PRINT statement.*

```
▤▢▬▬▬▬▬▬▬▬▬▬▬▬▬ Program 1-2 ▬▬▬▬▬▬▬▬▬▬▬▬▬
Enter 1:?  1
  Enter 2:?  2
  Thank you, let's do it again.
Enter 1:?  1   Enter 2:?  2  Thank you.
```

The fourth type of **INPUT** has a semicolon *before* the prompt string. This has the same effect as the semicolon before the variable list in the second form of **INPUT**. Try Program 1-2, and enter a 1 or 2 according to the prompt. Compare the resulting display with that of Program 1-1.
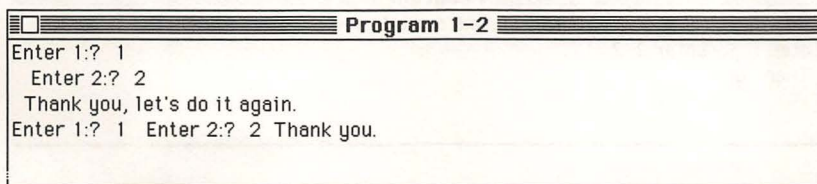
Program 1-2. INPUT Semicolon Prompt

```
CLS
INPUT "Enter 1:";A:INPUT "  Enter 2:";B
PRINT "  Thank you, let's do it again."
INPUT ;"Enter 1:";A:INPUT ;"  Enter 2:";B
PRINT "  Thank you."
```

Prompt strings have replaced the need to use a **PRINT** statement. Again, the use of the semicolons after the **INPUT** keywords suppresses linefeeds, keeping the following text on the same line.

The fifth and sixth forms of **INPUT** are similar to the third and fourth, but have a comma instead of a semicolon before the variable list to prevent the display of the question mark. Try Program 1-3 and enter a *1* or *2*.

Program 1-3. INPUT with Commas
```
CLS
PRINT "Enter 1:";:INPUT ;A:PRINT "  Enter 2:";:INPUT ;B
PRINT "  Great, one more time."
INPUT ;"Enter 1:",A:INPUT "  Enter 2:",B
PRINT "  Thank you."
```

The question marks are omitted by BASIC when a comma is placed in front of the variable list.

Figure 1-3. *Adding a comma after the prompt string in INPUT suppresses the question mark.*

```
▤□▬▬▬▬▬▬▬▬ Program 1-3 ▬▬▬▬▬▬▬▬
Enter 1:?  1   Enter 2:?  2   Great, one more time.
Enter 1: 1   Enter 2: 2
  Thank you.
```
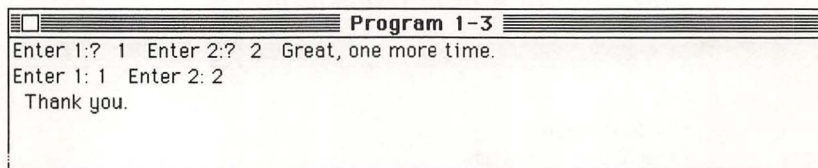
The last four forms of INPUT are more convenient than the first two and enhance the user interface in programs. The ultimate goal of any program is that the program functions properly and that it is easy to use. With this goal in mind, there are some problems with INPUT with respect to entering commas, semicolons, and certain command keystrokes as data. A more flexible means of input is required—the **LINE INPUT** command.

LINE INPUT
The four forms of **LINE INPUT** are

LINE INPUT *string variable*
LINE INPUT;*string variable*

LINE INPUT*"prompt string";string variable*
LINE INPUT;*"prompt string";string variable*

LINE INPUT is restricted to inputting a single string into a string variable. You'll see later that this isn't really a restriction, but instead is a built-in safety measure. Input may consist of any characters—including commas and semicolons—but it cannot consist of certain Command keystrokes. Accidentally entered characters can be erased with the Backspace key. BASIC does not display a question mark when a program uses **LINE INPUT**. Input is terminated with the Return or Enter key. Each of the forms of **LINE INPUT** parallels the first four forms of **INPUT**. Try Program 1-4 and enter $1,200.00 when the prompts display.

Program 1-4. LINE INPUT

```
CLS
LINE INPUT "Testing LINE INPUT: Enter $1,200.00: ";A$
INPUT "Testing INPUT: Enter $1,200.00: ";A$
```

This program asks for the string $1,200.00. This is done twice; the first time it's accepted via **LINE INPUT**. However, there will be a *?Redo from start* message at the second attempt because **INPUT** cannot accept a comma as part of the data.

Figure 1-4. *Unlike INPUT, LINE INPUT can accept punctuation as input. The drawback is that only strings can be entered.*

```
▤▯▤▤▤▤▤▤▤▤▤▤▤▤ Program 1-4 ▤▤▤▤
Testing LINE INPUT: Enter $1,200.00:  $1,200.00
Testing INPUT: Enter $1,200.00: ?  $1,200.00
?Redo from start
Testing INPUT: Enter $1,200.00: ?  $1200.00
```
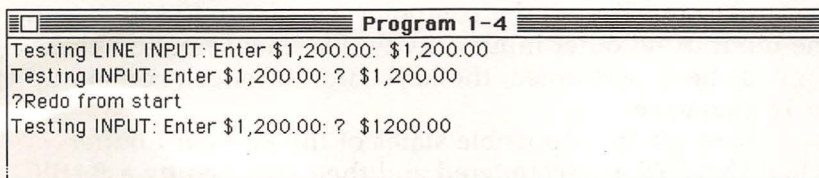
LINE INPUT is the typical input command used within the software tools in the next section. Entering Command keystroke characters as input is not a typical requirement and many cannot be entered with **LINE INPUT**.

INKEY$

INKEY$ is another input command, and has this syntax:

String variable = **INKEY$**

This command is much harder to use than either **INPUT** or **LINE INPUT** when entering data like numbers and strings. It has no prompting provisions, so messages must be provided with a **PRINT** statement. However, it's ideal for realtime keyboard processing required by interactive keyboard graphics, editors, and action games. For these applications, string input processing is rare and prompting is unlikely. After all, you wouldn't want to see a game where the message *Press H to move down* appeared after every movement of a game figure.

INKEY$ functions differently from the previously discussed forms of input in that it doesn't wait for the user to make an entry on the keyboard. When the user does provide input, a **PRINT** command is required to echo the entered data to the screen.

Figure 1-5. *INKEY$ does not wait for input.*

```
▤□▤▤▤▤▤▤▤▤▤▤▤▤▤▤ Program 1-5 ▤▤▤▤▤▤▤▤▤
Press some keys
The first character typed was: q
```
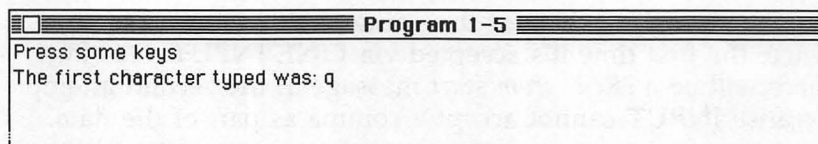
Memory allocated in the Macintosh serves as a *keyboard buffer*, the place in memory where the Macintosh records the keystrokes typed until the computer gets around to processing the input. If no other input activity (like mouse button clicking) has been performed, the keyboard buffer can contain up to 16 characters.

There are three possible states of the keyboard buffer when **INKEY$** is encountered and then executed by a BASIC program. First, the user has not typed a character on the keyboard so the buffer is empty. If this is true, a null character (" ") is stored in the string variable assigned to **INKEY$**. The keyboard buffer remains unchanged. Second, the user has entered one character and so the buffer has one character in it, which is then stored in the string variable. The keyboard buffer then empties. Third, the user has typed more than one character since the last attempted input with **INKEY$** and

therefore the buffer has more than one character in it. The first character in the buffer, which is also the first character typed, is stored in the string variable. The keyboard buffer then contains one fewer character. Try Program 1-5 for a demonstration of how to use **INKEY$**.

Program 1-5. INKEY$

```
CLS
PRINT"Press some keys"
FOR I=1 TO 1000:NEXT I:' A pause
K$=INKEY$
IF LEN(K$)=1 THEN PRINT "The first character typed was: ";K$ E
LSE PRINT "Nothing was typed!"
```

This program waits a short time before trying to read a character from the buffer. If there is at least one character in the buffer, the first one typed will be displayed; otherwise, a message states that *Nothing was typed!*.

The first line clears the *Output* window before the message *Press some keys* appears. After a pause created by the **FOR-NEXT** loop, the fourth line executes **INKEY$**. If a character has been typed, it's stored in the string variable K$; otherwise, a null string is stored. The length of the string stored in K$ is zero if it's a null string. On the other hand, any other character stored will have a length of one. This test for the length of the contents of K$ is performed with an **IF** statement, which then displays an appropriate message. If more than one character was typed, the remaining characters will appear in the *Command* window. (You can see these extra characters by choosing *Show Command* from the *Windows* menu.) These typed characters may cause trouble if they're not deleted with the Backspace key.

INPUT$

Yet another form of keyboard input is available—**INPUT$**—typically used for file input. **INPUT$** has this syntax:

String variable = **INPUT$**(*number*)
String variable = **INPUT$**(*number,filenumber*)
String variable = **INPUT$**(*number,#filenumber*)

INPUT$ differs from the previous forms of input because it

may be used to input data from disk files as well as from the keyboard.

The first form of **INPUT$** is used to read data from the keyboard. It functions similarly to **INKEY$** and **LINE INPUT**. Its input must go to a string variable; all input is accepted except for certain Command keystrokes. It does *not* have any provision for displaying a prompt message. The value of *number* controls how many characters to input. To read exactly five characters, for instance, a statement like $X\$ = INPUT\$(5)$ would be used. **INPUT$** does not echo its input to the screen; therefore, typing cannot be deleted with the Backspace key. Moreover, if Backspace is used, it's included as part of the input data. Unlike **INKEY$**, **INPUT$** waits for user input.

The other two forms of **INPUT$** have the same properties, but are used to read from files. The files are specified by the value of *filenumber*, which represents the buffer associated with the file. This will be discussed in more detail later.

INPUT$ is useful for entering data where a specific number of characters are required, as well as when reading from files or devices attached to the Macintosh. Try Program 1-6; enter your name after you see the prompt. The effects are best illustrated if you type slowly and look at the screen after each character is entered. Don't be concerned if, at first, it seems that the computer is ignoring what you're typing.

Program 1-6. INPUT$

```
CLS
PRINT "Type your full name:";
NA$=INPUT$(3)
PRINT:PRINT "The first three letters in your name are ";NA$
```

This program prompts you to enter your name, reads the first three characters typed, and displays them.

Figure 1-6. *The program seems to ignore what you type because INPUT$ does not echo input to the screen.*


```
================= Program 1-6 =================
Type your full name:
The first three letters in your name are Phi
```
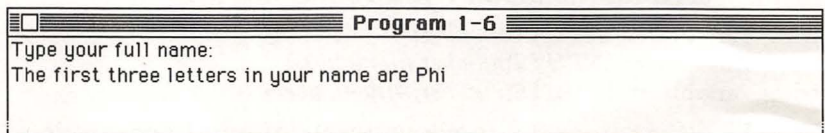
The first and second lines clear the *Output* window and display the prompt, respectively. **INPUT$** has a value of 3. The program waits until three characters are typed on the keyboard and stored as a string in NA$. Notice that the characters typed are not displayed until they're printed by the last line. Again, extra characters you type will be displayed in the *Command* window. Delete them with the Backspace key.

Since characters typed when using **INPUT$** are not displayed, **PRINT** is necessary. Try Program 1-7 to see how this is done.

Program 1-7. INPUT$ with PRINT

```
CLS
PRINT "Type your full name:";
NM$=""
FOR I=1 TO 3
NA$=INPUT$(1)
PRINT NA$;
NM$=NM$+NA$
NEXT I
PRINT:PRINT "The first three letters in your name are ",NM$
```

This is equivalent to Program 1-6, except the third line in Program 1-6 is replaced by six lines in this listing. This causes each character to be displayed as it's being entered. The results, however, are the same.

The third line initializes the string variable NM$ to a null string. It will contain the first three letters typed. The **FOR-NEXT** loop reads three characters from the keyboard and displays them as they're typed. A character is read by **INPUT$**, stored in NA$, and displayed by **PRINT**. This character is appended to the string stored in NM$. After three characters are typed and appended, they're displayed on the screen by the final **PRINT** statement.

O N E

Figure 1-7. *Unlike INKEY$, INPUT$ waits for input. If a loop displays each character after input with PRINT, a file and device-based INPUT is simulated.*

```
▤▢▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬ Program 1-7 ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬
Type your full name:Phi
The first three letters in your name are Phi
```
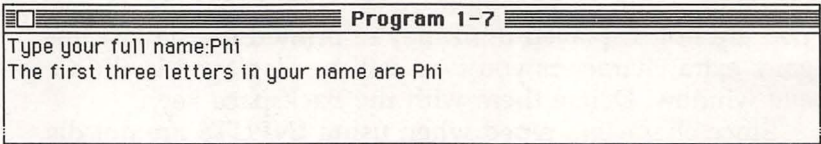
Change the fifth line of Program 1-7 to

```
GETKEY:
NA$=INKEY$:IF LEN(NA$)=0 THEN GETKEY
```

This line simulates **INPUT$(1)** by using **INKEY$** and an **IF-THEN** loop to wait for input. Compare the results of this new program with those of the old. They should be identical.

CHAPTER

2
Input Processing Commands

2
Input Processing Commands

Since it's concerned with validating data entered into a program, input processing is closely connected with input. Data integrity—in other words, making sure that the user of the program cannot enter undesirable data—is a vital aspect of providing a good user interface. For example, a properly constructed invoicing program would not allow a user to enter an item description when only a price was desired. With the proper input commands (see Chapter 1), a program *can* be made to control the data it receives by letting BASIC perform data checking. These controls, however, are very general and inadequate for all but the simplest programs. The only checking that BASIC will perform is to insure that strings are not entered into numeric variables and that the proper number of items are entered when there is a variable list associated with the input command.

Before looking at some methods of programming input routines which check the nature of the data, we need to examine the BASIC commands which assist in this task. In order to allow *any* data to be input—without allowing BASIC to override that input—data should be placed into string variables. Numeric data input is more efficient when placed in numeric variables, but then it's only safe when reading from files with a known and fixed format. Few things are more confusing to a user than having a *?Redo from start* message stamped here and there on a screen which has scrolled due to such a message. When these messages appear, the input fields of the screen may no longer line up with the prompts.

A user interface doesn't directly deal with files; when it does, all input should go into strings for analysis and possible manipulation. Moreover, user input into variable lists of more than one variable is dangerous. It requires the user to become

15

familiar with input delimiters. Confusion is the result when a comma or semicolon needs to be entered as part of the data and as a data delimiter.

For the present, let's assume that all user input is from **LINE INPUT** when strings are to be entered or from the **INKEY$** command when single keystrokes are needed. Therefore, the data could be anything from 32,767 characters long to only a null string. Typically, the input data will be an alphabetic string like *John Smith*, a numeric string like *5525*, or a mixture like *$100.00CR*. Punctuation and symbols are included as alphabetic characters. A dash or period is either an alphabetic character or a numeric character, depending on the context in which it is used.

LEN

The first question to resolve after accepting input with the **LINE INPUT** command is whether or not any data was actually entered. Pressing the Return or Enter key at the time of input does nothing but create a null string. But if there *is* input, how many characters does the string contain? The BASIC command which answers both questions is called **LEN**:

Numeric variable = **LEN**(*string variable*)
Numeric variable = **LEN**(*string*)

LEN returns a numeric value equal to the number of characters in the string variable or in the string. Zero is returned if either is a null string. This input length is useful when trying to parse free-format input or when padding input to a fixed length. Try Program 2-1 and enter your own name at the prompt.
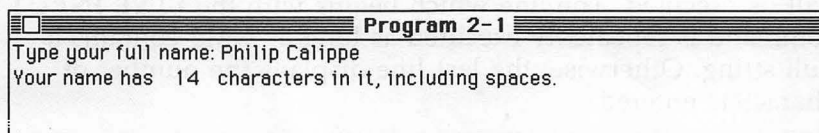
Program 2-1. LEN

```
CLS
LINE INPUT "Type your full name:";NM$
PRINT "Your name has ";LEN(NM$);" characters in it, including sp
aces."
```

This program asks you to type your name, then determines how many characters are in it. Run the program a second time, but try pressing only the Return key when the program asks for your name.

Figure 2-1. *LEN counts spaces and punctuation when determining the length of a string.*

```
≡□≡══════════════ Program 2-1 ≡══════════
Type your full name: Philip Calippe
Your name has  14  characters in it, including spaces.
```

The second line uses **LINE INPUT** to prompt you for your name which, when entered, is stored in NM$. **LEN** appears in the third line to determine the length of NM$, or the number of characters in your name.

WHILE-WEND

If a program requires input, but must reject a null string, a **WHILE-WEND** command can be used to repeat the request until something other than a null string is entered. Type in and run Program 2-2. Press Return the first time you see the prompt and enter your name the second time.

Program 2-2. WHILE-WEND

```
CLS
NM$=""
WHILE LEN(NM$)=0
LINE INPUT "Type your full name:";NM$
WEND
PRINT "Your name has";LEN(NM$);"characters in it, including spa
ces."
```

This does the same thing as Program 2-1 except that it doesn't accept a null string input.

Figure 2-2. *Using LINE INPUT and LEN in a WHILE-WEND loop is an easy way to insure that an input is made.*

```
≡□≡══════════════ Program 2-2 ≡══════════
Type your full name: Philip Calippe
Your name has  14  characters in it, including spaces.
```
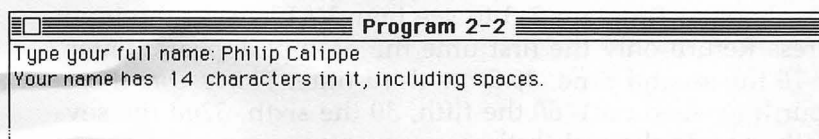
The second line in Program 2-2 initializes NM$ to a null string so that the **WHILE-WEND** loop in the next three lines will be executed. The line which begins with the **LINE INPUT** command is repeatedly executed as long as NM$ contains a null string. Otherwise, the last line displays the number of characters entered.

VAL

When a program concludes that it has some input, it may require that input be of a specific value or range of values. This can be tested with the **VAL** command:

Numeric variable = **VAL**(*string variable*)
Numeric variable = **VAL**(*string*)

- **VAL** returns the *numeric equivalent* of the string variable or string by converting the string into a number. The value of a null string is zero, as is a string starting with a nonnumeric character.
- Strings beginning with numeric characters are evaluated up to the first nonnumeric character. For instance, **VAL**("34.5Z99") returns 34.5. The Z character as well as the two 9s are ignored.
- As usual, there are exceptions—two in this case. The first involves the characters *e* and *d*, which are used for inputting single- and double-precision numbers, respectively. Therefore, **VAL**("4e4") is equal to 40000 and **VAL**("3.26d3") equal to 3260. The second exception is a string beginning with either *&H* or *&O*. These are prefixes for hexadecimal and octal numbers. **VAL**("&H64"), for instance, is equal to 100 (decimal), and **VAL**("&O77") is equal to 63 (decimal).
- **VAL** also ignores leading blanks, tabs, and linefeeds.

If a program requires a numeric input greater than zero, the value of the string can be checked by **VAL**. If the value is zero, the input would have to be rejected. Or if a specific value is required, all other inputs could be checked for and rejected if they don't meet the requirements.

Look at Program 2-3 to see how **VAL** checks the input. Press Return only the first time the prompt appears. Enter −10 the second time, 30.5 the third time, *Thirty-One* the fourth (spelled out), 60 the fifth, 30 the sixth, 32nd the seventh, and 31 the eighth time.

TWO

Figure 2-3. *VAL converts string input into a numeric value.*

```
▤▢▦▦▦▦▦▦▦▦ Program 2-3 ▦▦▦▦▦▦▦
How many days in December:
Please enter a number.
How many days in December: -10
No negative numbers please!
How many days in December: 30.5
No fractional numbers please!
How many days in December: Thirty-One
Please enter a number.
How many days in December: 60
No month has that many days in it.
How many days in December: 30
Check your calendar again.
How many days in December: 32nd
No month has that many days in it.
How many days in December: 31
Correct
```

Program 2-3. *VAL*

```
CLS
DY=0
WHILE DY<>31
LINE INPUT "How many days in December: ";DY$
DY=VAL(DY$)
IF DY=0 THEN PRINT "Please enter a number."
IF DY<0 THEN PRINT "No negative numbers please!"
IF DY<>INT(DY) THEN PRINT "No fractional numbers please!":GOT
0 LOOPEND
IF DY>31 THEN PRINT "No month has that many days in it."
IF DY>0 AND DY<31 THEN PRINT "Check your calendar again."

LOOPEND:
  WEND
  PRINT "Correct"
```

Though this program tests to see how many days there are in December, it has a more important function, that of demonstrating how to perform input checking with the assistance of **VAL**.

The value of the variable DY is set to 0 in the second line. This variable will contain the user-entered input. The **WHILE-WEND** loop inputs the user's reply and checks to see if it's correct. Note that the loop occupies most of the program. **LINE INPUT** is used within this loop to prompt for input and then stores this in DY$. The numeric equivalent of DY$ is calculated with the **VAL** function and stored in DY (not DY$). When only the Return key is pressed, the value (as you saw when you ran the program) of DY is 0. If that's true, the message *Please enter a number* is displayed. This same line detects the input of the string *Thirty-One* since its value is also 0. The erroneous entry of −10 is trapped by the statement IF DY<0 in the following line. The next line detects the entry of fractional numbers by comparing the entered number with the truncation of the entry. If these two differ, the entry has a fractional part and cannot be accepted. Notice that this trapped the 30.5 entry. Another error trap checks to make sure that DY is not greater than 31—this detects the wrong answer of 60—while yet another line confirms that the answer is possibly correct for a different month but not for December. When a value of 31 is finally entered, the loop is exited and the last program line tells you that the answer is correct.

You've probably noticed that there's more program code involved with input checking than with getting the input itself. That's where a software tool for input will become valuable. It does all the work—the programmer simply decides which input tool to use. This saves programming as well as debugging time. Such software tools will be discussed in more detail later.

STR$

After data is entered, a program may check that the value of the data is actually correct. Often, the value entered is acceptable to the program, but is not really what the user had intended. For instance, a program may ask for the part number, and the user types *31450* instead of *31450*—An uppercase letter *O* is typed instead of a zero. This is a common mistake. The program interprets the value as 3145, which happens to be another item number. The display still shows the 31450 typed by the user. Hence, the data entered is accepted and data for the next item is entered. Imagine the customer's surprise when the order arrives.

A possible solution would be to redisplay the actual number entered with a **PRINT USING** and **VAL** statement. If the number 31450 is really the combination of several numbers, such as in department 31, item number 450, the string might be parsed with string commands. Thus, a command to convert the **VAL** of a string *back to a string* is useful. The BASIC command for this task is **STR$**:

String variable = **STR$**(*numeric variable*)
String variable = **STR$**(*number*)

• *Number* can be in forms ranging from 123 or 3d3 to &H64.
• The **STR$** command will expand any of these forms into a string of numeric characters unless there are more than 16 digits in the expansion. **STR$**(3d15), for example, is equal to 3000000000000000, while **STR$**(3d16) is equal to 3d+16.

Type in and run Program 2-4, entering *31450* the first time the prompt for the part number appears. The letter *O* should be entered in uppercase. Respond with *n* to the following question. Then enter *31450* as the part number. This time the 0 *is* a zero. Respond with *y* to the question.

Program 2-4. STR$

```
CLS

LOOP:
  LINE INPUT "Enter the part number: ";PARTNO$
  PARTNO$=STR$(VAL(PARTNO$))
  PRINT "Part number ";PARTNO$;"  [y/n]? :";
  LINE INPUT YN$
  IF YN$="n" OR YN$="N" THEN LOOP
PRINT "Order for item ";PARTNO$;" entered."
```

This program asks for a part number, redisplays the number entered, and determines if this value was what the user meant. This is performed with the aid of the **STR$** command. When the first string is entered as *31450* (uppercase *O*) instead of *31450* (number zero), it's interpreted as the number 3145 instead of 31450. This is displayed and the user has a chance to make the necessary corrections.

Figure 2-4. *VAL is used to certify numeric string inputs.*

```
▤◻▦▦▦▦▦▦▦▦▦▦▦▦▦▦▦▦ Program 2-4 ▦▦▦▦
Enter the part number:  31450
Part number 3145   [y/n]? : n
Enter the part number:  31450
Part number 31450   [y/n]? : y
Order for item 31450  entered.
```

    The first line in the LOOP subroutine (**LINE INPUT**) asks for a part number stored in PARTNO$, which is updated to represent its numeric equivalent in the next line. This new string is displayed by the **PRINT** statement, which also asks if the correct part number was entered. A second **LINE INPUT** accepts the answer. If the answer indicates an incorrect entry, the program jumps back to the beginning of LOOP where it can be reentered. Otherwise, the order for the item with the specified number is acknowledged.

    Again, notice the amount of extra programming required for input processing. If the above example was used as a model for inputting all numeric data, programs would need two responses for each data item entered. It would be preferable if only one entry per item was necessary, and additional entries were required only for corrections. There are ways to verify responses which don't use a mandatory answer to an *Is this correct?* question. These are discussed as part of the functions of some of the software tools later in the book.

LEFT$, RIGHT$, and MID$

Often, string input needs to be parsed, or broken up into smaller pieces of information. For example, if the part number of an item was a code representing the store department, item catalog number, and color number, then other BASIC commands can be used to extract this information. **LEFT$** is one such command you'll find useful:

String variable = **LEFT$**(*string variable,number*)
String variable = **LEFT$**(*string,number*)

- **LEFT$** is a function which returns the leftmost number of characters (as specified by the number within parentheses) from the string or string variable.
- *Number* may have any value from 0 to 32767.

22

For instance, given that the first two characters of a part number indicate its department, Program 2-5 shows how to extract this number from the string. Run it and type in *378903087* for the part number.

Program 2-5. LEFT$

```
CLS
LINE INPUT"Enter the part number:";PARTNO$
DPNO$=LEFT$(PARTNO$,2)
PRINT"The item is found in department: ";DPNO$
```

LINE INPUT takes the part number and stores it in string PARTNO$. Since the department number is composed of the two leftmost characters, the next line extracts it from the string with **LEFT$** and stores it in string DPNO$. The department number, *37*, is then displayed by **PRINT**.

Another parsing command—in fact, the exact opposite of **LEFT$**—is **RIGHT$**, which extracts characters from the rightmost part of a string:

String variable = **RIGHT$**(*string variable,number*)
String variable = **RIGHT$**(*string,number*)

• Unlike **LEFT$**, **RIGHT$** returns the specified rightmost number of characters from the string or string variable. Except for this difference, the command functions identically to **LEFT$**.

To see **RIGHT$** in operation, add the following two statements to Program 2-5. When adding these lines, place the first line below the one which begins with DPNO$, and the second line at the end of the program.

It's assumed that the last two characters of the input string contain the color code. Once you've added these statements, rerun the program, again entering *378903087* as the part number.

```
COL$=RIGHT$(PARTNO$,2)
PRINT"The item color is color code: ";COL$
```

The first line to add extracts the color code from the part number stored in PARTNO$ with **RIGHT$** and stores it in string COL$. The color code is then displayed by the **PRINT** statement.

The most versatile of the three parsing commands, however, is **MID$**. This command lets you extract a string from *any* part of another string:

String variable = **MID$**(*string variable,position number,number of characters*)

String variable = **MID$**(*string,position number,number of characters*)

String variable = **MID$**(*string variable,position number*)

String variable = **MID$**(*string,position number*)

MID$(*target string,position number,number of characters*) = String variable

MID$(*target string,position number,number of characters*) = String

MID$(*target string,position number*) = String variable

MID$(*target string,position number*) = String

The first and second forms correspond to the third and fourth forms, but specify a number of characters to copy. The last four forms correspond to the first four except that they *modify* a part of a string instead of simply copying.

- When using **MID$** to copy characters, *position number* refers to the position of the character at which copying begins. If the position number value is greater than the length of the string, a null string is returned.
- *Number of characters* is a count of how many characters, starting from the position number, will be copied. If this is greater than the number of characters which follow the position number, the rest of the string is returned. Notice that the original string is never modified by these forms of **MID$**. If the number of characters is not specified, the remainder of the string, from the position number on, is returned.
- Using **MID$** to replace a portion of a string has more constraints. The number of characters replaced cannot be less than or greater than the number of characters inserted. In other words, the length of the string being modified cannot change. If fewer characters are being inserted than are replaced, then the smaller number of characters will be replaced. If more characters are being inserted than are replaced, then the smaller number of characters will be inserted.
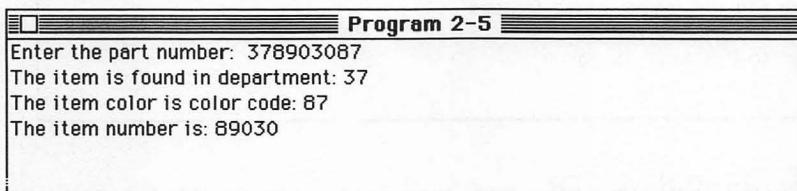
Look again at Program 2-5. The item number can be extracted from the part number by using **MID$**. Add the following lines to Program 2-5 and again type in *378903087* as the part number. Place the first line immediately under the line

which contains the **RIGHT$** command, and the second line at the end of the program.

```
ITEMNO$=MID$(PARTNO$,3,5)
PRINT"The item number is: ";ITEMNO$
```

The item number is encoded as the third through the seventh characters of the part number. This item number is pulled from the part number with **MID$**, stored in ITEMNO$, and then displayed by the **PRINT** statement in the last line.

Figure 2-5. *The department, color, and item numbers are extracted from the input with LEFT$, RIGHT$, and MID$.*

```
▤□▨▨▨▨▨▨▨▨▨▨▨▨▨ Program 2-5 ▨▨▨▨▨▨▨
Enter the part number: 378903087
The item is found in department: 37
The item color is color code: 87
The item number is: 89030
```

INSTR

There will be cases where the position of the substring to be extracted and its length are unknown. For instance, a program may ask for a person's full name and then determine what the first and last names are. All that's known about the data is that there is a space *somewhere* in the middle which can be located with **INSTR**:

Numeric variable = **INSTR**(*offset,target string,search string*)
Numeric variable = **INSTR**(*target string,search string*)

- The *search string* is searched for within the *target string*—
  INSTR returns the position where the match is found. In first and last name decoding, the search string would be a space. If this string is null, then **INSTR** returns the offset or the value 1 if there is no offset.
- The *target string* is the string being analyzed, in other words, the string containing the full name. **INSTR** returns zero if the search string is not within the target string or if the target string is a null string.

• If the *offset* is used (it's optional), it's the starting position within the target string where **INSTR** begins looking for the search string. When an offset greater than the length of the target string is used, **INSTR** returns a value of zero. If no offset is used, then the first character in the target string is used as the starting point.

Program 2-6 shows how to parse the first and last names from an input. Enter your full name when the prompt appears. Rerun the program and enter only your first name.

Figure 2-6. *Parsing input requires looking for delimiters with INSTR.*

```
▤▯▤▤▤▤▤▤▤▤▤▤▤▤▤▤ Program 2-6 ▤▤▤▤▤▤▤▤▤▤
Enter your full name: Philip Calippe
Your last name is: Calippe
Your first name is: Philip
```

Program 2-6. INSTR
CLS

```
GETNAME:
    LINE INPUT "Enter your full name: ";NM$
    P=INSTR(NM$," ")
    IF P=0 THEN PRINT"Please ";:GOTO GETNAME


FIRSTNM$=LEFT$(NM$,P-1)
LASTNM$=RIGHT$(NM$,LEN(NM$)-P)
PRINT "Your last name is: ";LASTNM$
PRINT "Your first name is: ";FIRSTNM$
```

Program 2-6 takes your full name as input and then parses it to determine your first and last names. It requires a space typed between the first and last names. If this criterion is not met, the program asks you to reenter your full name.

The first line of the GETNAME routine uses **LINE INPUT** to place your name into NM$. The next line uses **INSTR** to determine the location of the space that should be somewhere within NM$. The location of the space is stored in P. The last

line of the routine checks to see if P is equal to zero, which would indicate that no space was typed. If that's true, then the program goes back to GETNAME and asks for the information again. If P is *not* equal to zero, the next line extracts your first name with **LEFT$**. The first name would be the first $P-1$ characters of the input. **RIGHT$** finds your last name. It assumes that this is located in the input string after the location of the space. The last two lines of the program display your last and first names, respectively.

CHR$

When you use **INKEY$**, input is stored in a string variable and the length of the string returned is always 1 or 0. If the value is 1, then the string variable is actually a character. Once you know that a character exists in a string, you can process that character. One command for character processing is **CHR$**, which is useful in interactive programs such as keyboard-controlled action games:

Target string = **CHR$**(*number*)

• The value of *number* is the ASCII or non-ASCII value of the character to be returned to the target string. This value ranges from 0 to 255, with values greater than 128 being non-ASCII characters.

To see all the interesting characters in the font BASIC is currently displaying, type in and run Program 2-7.

Program 2-7. CHR$

```
CLS
FOR I=32 TO 255
PRINT I;" = ";CHR$(I);"   ";
IF I MOD 4 = 0 THEN PRINT
NEXT I
```

This prints all the displayable characters in the default font in the *Output* window. Microsoft BASIC 2.0 uses the Geneva font. The characters displayed by each font vary slightly.

Figure 2-7. *CHR$ shows what characters are available in a font (in this case, Geneva). The open boxes indicate that a character is undefined.*

□			Program 2-7
189 = Ω	190 = æ	191 = ø	192 = ¿
193 = í	194 = ¬	195 = √	196 = ƒ
197 = ≈	198 = ∆	199 = «	200 = »
201 = …	202 =	203 = À	204 = Ã
205 = Õ	206 = Œ	207 = œ	208 = –
209 = —	210 = "	211 = "	212 = '
213 = '	214 = ÷	215 = ◊	216 = ÿ
217 = ◄	218 = □	219 = □	220 = □
221 = □	222 = □	223 = □	224 = □
225 = □	226 = □	227 = □	228 = □
229 = □	230 = □	231 = □	232 = □
233 = □	234 = □	235 = □	236 = □
237 = □	238 = □	239 = □	240 = □
241 = □	242 = □	243 = □	244 = □
245 = □	246 = □	247 = □	248 = □
249 = □	250 = □	251 = □	252 = □
253 = □	254 = □	255 = □	

The second line of Program 2-7 begins a **FOR-NEXT** loop which ends at the bottom of the program. This loop iterates the value of variable I, which represents the character number to display. **PRINT** I in the next line displays the character number and the corresponding character using **CHR$**. The semicolon at the end of this line enables the creation of a table, while the statement **IF I MOD 4 = 0** starts a new line after every four items are printed. (This example is not an indication of how **CHR$** is typically used.)

One of the most common uses for **CHR$** is assigning hard-to-type characters, such as Option characters, to a string variable. One of the more interesting characters within the display font of BASIC is **CHR$(217)**. Try Progam 2-8 and find out why.

Program 2-8. CHR$(217)

```
CLS
RBT$=CHR$(217)
FOR I=1 TO 50
CALL MOVETO(5,100)
OT$=SPACE$(51-I)+RBT$
```

```
PRINT OT$
NEXT I
```

The program shows some primitive animation by moving a rabbit from left to right across the screen. Notice how the rabbit slows down. Better animation techniques are discussed later, as well as the **CALL** command and **MOVETO** ROM routines which you see in this example.

The second line stores the character representing the figure in RBT$. This line may have been entered by pressing the key combination of Shift-Option-tilde (tilde is the symbol on the key immediately to the left of the 1 key).

```
RBT$="✿"
```

It's better to use **CHR$(217)**, because a figure of a rabbit doesn't explain to someone which keys to type.

FOR I = 1 **TO** 50 displays the figure 50 times. The next line sets the position where the start of RBT$ is going to be displayed. Here, **CALL** is used to position the cursor at a specific spot for the **PRINT** statement below. By adding more spaces to the front of the contents of OT$ and printing it in the same place each time, animation is simulated. OT$ is padded with spaces before **PRINT** puts the string contents on the screen.

Figure 2-8. *CHR$(217)* draws a rabbit which runs across the Output *window to this final position.*
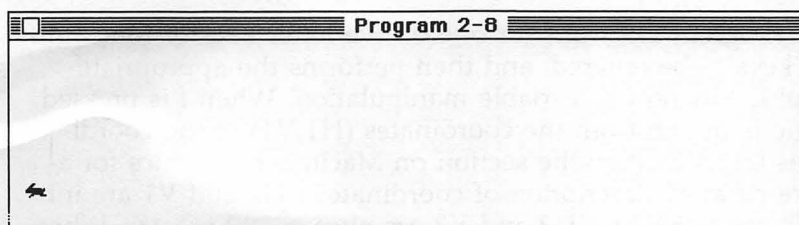
ASC

BASIC commands often have a converse command, and **CHR$** is no exception. Its converse is **ASC**. **ASC** returns the ASCII value of its parameter:

Target numeric = **ASC**(*string variable*)
Target numeric = **ASC**(*string*)

The string variable or string may be of any size, but only the ASCII value of the first character is returned. This command may also be used to test for non-ASCII characters, like the rabbit shape in Program 2-8. A good use of **ASC** is to decode keyboard commands received by **INKEY$**. Program 2-9 is a good demonstration. Enter characters like *l*, *b*, *c*, *f*, or *q*.

Program 2-9. ASC

```
CLS
PRINT "Press one of b,c,f,l,q,1,2,3,4:"
CMD$="":H1=50:H2=150:V1=50:V2=150
WHILE CMD$<>"q"
  CMD$=INKEY$
  IF CMD$="b" THEN LINE(H1,V1)-(H2,V2),33,B:GOTO FINCMD
  IF CMD$="c" THEN CLS:PRINT "Press one of b,c,f,l,q,1,2,3,4:":G
OTO FINCMD
  IF CMD$="f" THEN LINE(H1,V1)-(H2,V2),33,BF:GOTO FINCMD
  IF CMD$="l" THEN LINE(H1,V1)-(H2,V2):GOTO FINCMD
  IF CMD$>="1" AND CMD$<="4" THEN V=(ASC(CMD$)-48)*25:H1=
V:V1=V:H2=100+V:V2=100+V

FINCMD:
  WEND
```

Program 2-9 waits for one of five letter keys or four number keys to be entered, and then performs the appropriate graphic function or variable manipulation. When *l* is pressed, a line is drawn from the coordinates (H1,V1) to the coordinates (H2,V2). (See the section on Macintosh graphics for a more detailed description of coordinates.) H1 and V1 are initially set to 50, and H2 and V2 are always 100 greater. When *b* is pressed, the program draws a box frame. A box filled with black is drawn when *f* is hit. Pressing *c* clears the display and *q* stops the program. Keys 1–4 make H1 and V1 equal to 25 times the value of the key. Pressing *4*, then, makes H1 and V1 equal to 100, and H2 and V2, 200. To increase the values of

H1, V1, H2, and V2, make sure you press one of the number keys *before* pressing *b*, *f*, or *l*.

Figure 2-9. *Interactive keyboard graphic commands are interpreted with INKEY\$ and ASC.*
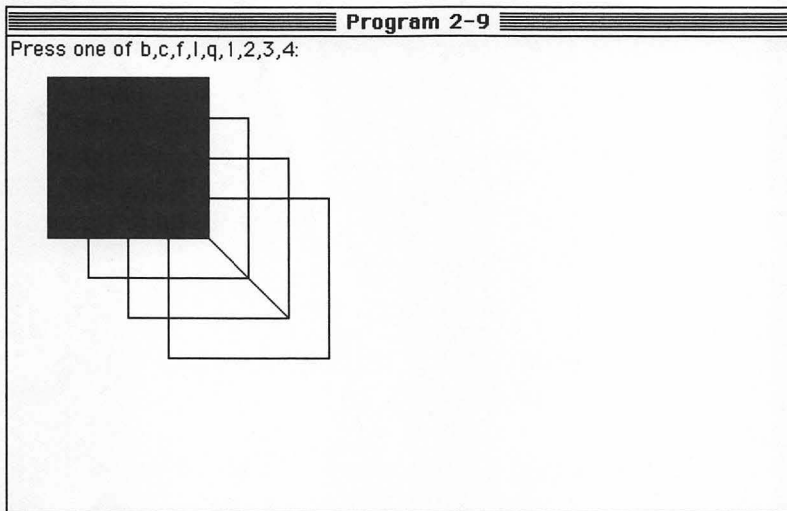
The first three lines clear the screen, display the available keys, initialize H1, H2, V1, and V2, and set CMD\$ to null. It's good programming practice to initialize variables *before* beginning the main portion of a program.

Each time a key is pressed, it's stored into CMD\$ by **INKEY\$**. Since **INKEY\$** does not wait for input, this line repeats until a character is stored. If the character stored in CMD\$ compares with one of those tested for in the remaining lines in the main loop, the **THEN** portion of that **IF** statement is performed. The last line of the **WHILE-WEND** loop uses **ASC** to convert the string input into a number.

When the character stored in CMD\$ is *q*, the test by the **WHILE** fails, and execution of the program continues after the **WEND**. But since there's no more program to execute, it ends.

The **GOTO**s at the end of each **IF** statement serve two functions. First, they speed up program execution. Only those necessary tests are performed—no additional (and unnecessary) statements are executed. A **GOTO** in the last line of the main loop is unnecessary and would only slow down the

program. However, if new commands are added, it would have to be modified. (Note the use of the label FINCMD in each **GOTO**. Remember that with Microsoft BASIC 2.0 on the Macintosh, line number references for such things as **GOTO**s and **GOSUB**s can be replaced with labels like this.)

CHAPTER

3
Graphics and Animation

3
Graphics and Animation

Since the Macintosh display is so graphics oriented, it's not surprising to find a set of graphics commands in Microsoft BASIC. However extensive these commands may be, they're not the complete set you have in hand for graphic manipulation. Another series of commands, even more powerful, is available through calls to ROM routines. Since we're starting out with the simplest and working toward more advanced programming practices, let's stick with the BASIC commands for now. Calls to ROM routines can wait a bit.

The Macintosh Screen

The Macintosh screen is divided into 342 horizontal lines, numbered 0 to 341, with line 0 at the top of the screen and line 341 at the bottom. Each line is divided into 512 points, called *pixels*, which are numbered from 0 to 511. Pixel 0 is on the far left, while pixel 511 is on the far right. Any pixel on the screen can thus be specified or addressed by using a pair of numbers separated by a comma.

These coordinates are often put in parentheses to make it plain that they are indeed coordinates; (0,0) and (255,171) would be two possible sets of coordinates. The first number in the pair is the horizontal position of the point, while the second is the vertical position. The Macintosh shows all points bounded by coordinates (0,0), (511,0), (511,341), and (0,341)—the top left, top right, bottom right, and bottom left corners, respectively. Graphics can be drawn outside this area, but they'll not be displayed on the screen. In fact, the Macintosh allows graphics to be located anywhere from coordinates (-32768,-32768) to coordinates (32767,32767). All points addressed outside this range will cause an overflow error and message.

35

THREE

Microsoft BASIC confines graphic output to the *Output* window. In pre–2.0 versions, part of the screen area was consumed by the menu bar at the top of the screen, by the window title of the *Output* window below the menu bar, and by the grow box and border on the right and lower edge of the output window. However, the *Output* window in 2.0 has a greater drawing area because it does not have a window border at the bottom. By double clicking on the *Output* window's title bar in BASIC 2.0, the program creates a drawing area of 299 lines with 491 pixels. Pixel (0,0) is at the top left and (490,298) at the lower right. Since the *Output* window may move about the screen, the coordinates used when drawing graphics are relative to the *Output* window, not to the screen. Therefore, the *Output* window point (0,0) may actually be the screen point (5,40). Take a look at Figure 3-1 for a graphic representation.

Figure 3-1. *The Macintosh Screen*
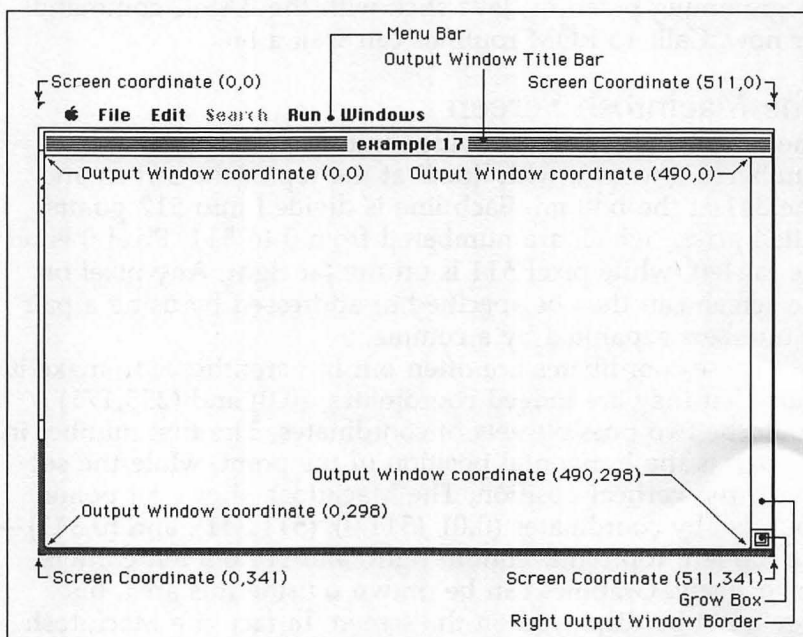
It's easiest to run a BASIC program from the *Finder* so that you don't have to stop and adjust the *Output* window

size. To facilitate this, programs, software tools, utilities, and applications in this book will restrict graphic output to the area bounded by the points (0,0) and (490,298).

CLS
You've already typed in one of the BASIC graphics commands. In fact, it was used in all the examples in Chapters 1 and 2. **CLS**, the BASIC keyword which clears the screen, appeared at the beginning of each example program. Its syntax is simply

CLS

This clears the *Output* window to the background color (white by default). The background pattern can be changed so that the *Output* window is cleared to any pattern, but this will be explained later in the section dealing with the ROM routine BACKPAT.

PSET
To actually plot a point on the screen, you'll be using another BASIC command—**PSET**:

PSET (*horizontal location,vertical location*)
PSET (*horizontal location,vertical location*),*color*
PSET STEP (*horizontal offset,vertical offset*)
PSET STEP (*horizontal offset,vertical offset*),*color*

- The *horizontal* and *vertical locations* are the coordinates of the point to be plotted.
- *Color* is equal to 33 for black, or 30 for white. If no color is specified, 33 (black) is used. Points plotted with color 33 are erased by plotting points at the same location with color 30.
- *Horizontal offset* is the horizontal distance from the last point plotted. A positive number plots a point to the right, while a negative number plots a point to the left.
- *Vertical offset* is the vertical distance from the last point plotted. This can be a positive number to plot a point closer to the bottom of the screen or a negative number to plot a point toward the top of the screen. If no point has been previously plotted, the last point is assumed to be the center of the *Output* window.

    Type in and run Program 3-1 for an illustration of where the *Output* window boundaries are located. Choose *Show Output* from the *Windows* menu, then double click the title bar of

the *Output* window to make it a full-sized screen. The last line creates a pause—if the pause is too short, change 10000 to a larger number.

Program 3-1. PSET

```
CLS
PSET(0,0)
PSET (490,0)
PSET(490,298)
PSET (0,298)
FOR I= 1 TO 10000:NEXT I
```

The program displays the four extreme points of the *Output* window display. Once the screen is cleared (**CLS**), the next four lines draw the points located in the top left, top right, bottom right, and bottom left corners, respectively. Setting points with the **STEP** and color options of **PSET** is a bit more involved. Program 3-2 uses both extensively, however, and will give you an idea of how they can be used. To end the program, press any key.

*Program 3-2. PSET STEP*

```
START:
  CLS
  FOR I= 0 TO 9
  PSET (I,I),33
  PSET (I,200-I),33
  PSET (205-I,I),33
  PSET (205-I,200-I),33
  NEXT I
  FOR I=0 TO 189
  PSET (I+10,I+10),33
  PSET STEP (-10,-10),30
  PSET (I+10,190-I),33
  PSET STEP (-10,10),30
  PSET (195-I,I+10),33
  PSET STEP (10,-10),30
  PSET (195-I,190-I),33
  PSET STEP (10,10),30
```

```
IF INKEY$<>"" THEN END
NEXT I
GOTO START
```

This program employs some cunning methods to produce animation. Four lines move toward each other from the four points of a box on the screen. When they meet, they don't stop, but continue toward the opposite corners of the box. This process is cycled endlessly until the program is stopped by a keypress.

Figure 3-2. *Animating four lines with PSET is a matter of plotting one point and erasing another.*
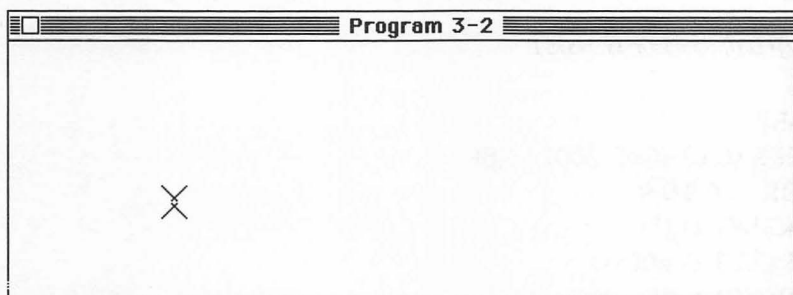
The lines are initially drawn by the first **FOR-NEXT** loop. The main body of the program is another **FOR-NEXT** loop which animates the lines by plotting a point at one end of the line and erasing a point at the other end. Because this happens so quickly, it appears that the line moves. The four two-line combinations of **PSET** and **PSET STEP** first add a point to the line in the appropriate direction to make it appear to move, then erase a point at the line's end. The first combination handles the line at the top left, plotting and erasing points so that the line seems to head for the bottom right corner. Similarly, the other three combinations animate the lines initially located at the bottom left, top right, and bottom right. **INKEY$** is used to determine if a key was pressed. If so, the program stops. When the lines reach their respective corners, the animation sequence is repeated by the **GOTO** START command.

T H R E E

PRESET

Another BASIC command for plotting points is **PRESET**:

PRESET (*horizontal location,vertical location*)
PRESET (*horizontal location,vertical location*),*color*
PRESET STEP (*horizontal offset,vertical offset*)
PRESET STEP (*horizontal offset,vertical offset*),*color*

Except for one difference, **PRESET** functions exactly like **PSET**.

- **PRESET** selects the background color (white) if a color is not specified.

    Run Program 3-3, and compare the result with that of Program 3-2. Again, press any key to stop the program.

Program 3-3. PRESET

```
CLS
START:
  LINE (0,0)-(205,200),33,BF
  FOR I= 0 TO 9
  PRESET (I,I)
  PRESET (I,200-I)
  PRESET (205-I,I)
  PRESET (205-I,200-I)
  NEXT I
  FOR I=0 TO 189
  PRESET (I+10,I+10)
  PSET STEP (-10,-10)
  PRESET (I+10,190-I)
  PSET STEP (-10,10)
  PRESET (195-I,I+10)
  PSET STEP (10,-10)
  PRESET (195-I,190-I)
  PSET STEP (10,10)
  IF INKEY$<>"" THEN END
  NEXT I
  GOTO START
```

    Except for the black background, this is the same as Program 3-2. Notice that the color is never specified because, by

default, **PRESET** plots in the opposite color of **PSET**. Here, points are drawn and erased by first plotting the point with **PRESET** on a black background, then plotting a point with **PSET**.

Figure 3-3. *These lines are animated by extending one end with PRESET while erasing the other with PSET. The roles of PSET and PRESET would be reversed if the lines were drawn on white.*
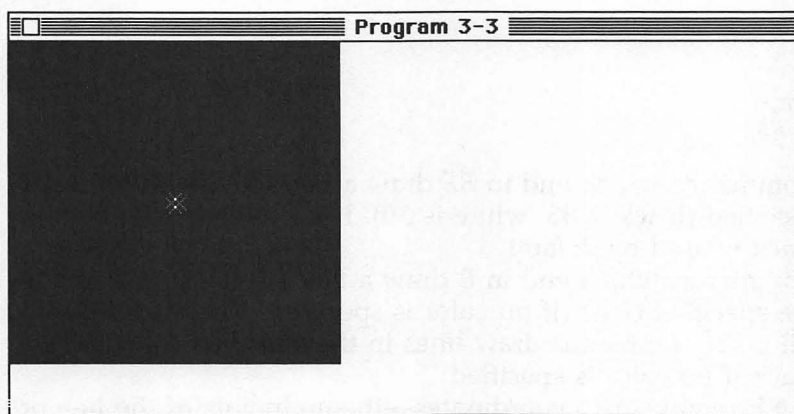
When you want to animate a point, two steps are involved. You must first draw an initial point, then cycle through the operations of drawing a new point and erasing the old. Since shapes are really just a set of points in an organized pattern, animating shapes can be done the same way. This can be a slow process (as you might have guessed after seeing the demonstrations in Programs 3-2 and 3-3), especially if the shapes are large. Other BASIC commands useful for animating shapes will be discussed later.

LINE

Along with drawing simple points, a command exists which draws entire lines, box frames, and filled boxes. The command is called **LINE**:

LINE –(*horizontal location 2,vertical location 2*)
LINE (*horizontal location 1,vertical location 1*)–(*horizontal location 2,vertical location 2*)
LINE – **STEP** (*horizontal offset 2,vertical offset 2*)

LINE STEP (*horizontal offset 1,vertical offset 1*)–(*horizontal location 2,vertical location 2*)
LINE (*horizontal location 1,vertical location 1*)– **STEP** (*horizontal offset 2,vertical offset 2*)
LINE STEP (*horizontal offset 1,vertical offset 1*)– **STEP** (*horizontal offset 2,vertical offset 2*)

Each of these formats has five additional variations, created by appending one of the following immediately after the final parenthesis:

,color
„B
„BF
,color,B
,color,BF

- Commands which end in *BF* draw a box filled with the color specified (black is 33, white is 30). If no color is specified, black is used by default.
- Commands which end in *B* draw a box outline or a frame in the specified color (if no color is specified, black is used).
- All other commands draw lines in the specified color, or black if no color is specified.
- **LINE** requires two coordinates—the endpoints of the line or opposite corners of the box. The corners which the coordinates represent depend on where they are located relative to each other. For example, if the first coordinate has a greater *horizontal location* value than the second coordinate, and a smaller *vertical location* value, the first coordinates indicate the upper right corner, while the second coordinates describe the lower left corner. If a box or filled box is drawn either with both horizontal locations at the same value or with both vertical locations at the same value, the result is a line. If both coordinates are identical, the result is a point.
- **LINE** commands which have only *horizontal location 2* and *vertical location 2* for coordinates are specifying just the second point. The first point is the last point specified prior to this command. If no point has yet been specified, the center of the *Output* window is used.
- The first point is indicated with forms using *horizontal location 1* and *vertical location 1*.
- Specifying *horizontal offset 1* and *vertical offset 1* means that the first point is relative to the position of the last plotted point. For instance, if the command **LINE** (20,20)–(40,40) has

just been executed, and the next command is **LINE STEP**
(20,20)–(90,200), the first point is located 20 pixels to the
right and 20 pixels below the last plotted point (40,40). It's
thus located at (60,60).
• When **LINE** includes *horizontal offset 2* and *vertical offset 2*,
the second point is located relative to the first. Type

```
CLS:FOR I=0 TO 100 STEP 10:LINE (I,I)-STEP(I+10,I+10),,B:NEX
T I
```

in the *Command* window and press Return.

Program 3-4 has a multitude of **LINE**s in its various
forms.

Program 3-4. LINE
```
CLS
LINE -(280,240)
LINE -(400,220),33
LINE -(350,170),,B
LINE -(380,200),,BF
LINE -(370,190),30,B
LINE -(360,180),30,BF
LINE (300,100)-(340,190)
LINE (300,200)-(340,100),33
LINE (200,20)-(440,90),,B
LINE (210,25)-(430,85),,BF
LINE (220,30)-(420,80),30,B
LINE (230,35)-(410,75),30,BF
LINE - STEP (-10,-5)
LINE - STEP (-10,0),33
LINE - STEP (-150,-30),,B
LINE - STEP (140,25),,BF
LINE - STEP (-130,-20),30,B
LINE - STEP (120,15),30,BF
LINE STEP (-360,-50)-(20,20)
LINE STEP (-10,0)-(20,25),33
LINE STEP (5,5)-(190,220),,B
LINE STEP (-5,-5)-(30,35),,BF
LINE STEP (5,5)-(180,210),30,BF
```

```
LINE (40,45) - STEP (5,5)
LINE (40,50) - STEP (5,5),33
LINE (45,55) - STEP (130,150),,B
LINE (50,60) - STEP (120,140),,BF
LINE (55,65) - STEP (110,130),30,B
LINE (60,70) - STEP (100,120),30,BF
LINE STEP (-5,-5) - STEP (-5,-5)
LINE STEP (5,0) - STEP (-5,-5),33
LINE STEP (-85,-100) - STEP (80,95),,B
LINE STEP (-75,-90) - STEP (70,85),,BF
LINE STEP (-65,-80) - STEP (60,75),30,B
LINE STEP (-40,-55) - STEP (20,20),30,BF
FOR I=1 TO 4000:NEXT I
```

Figure 3-4. *All these shapes and lines were drawn with the LINE command.*
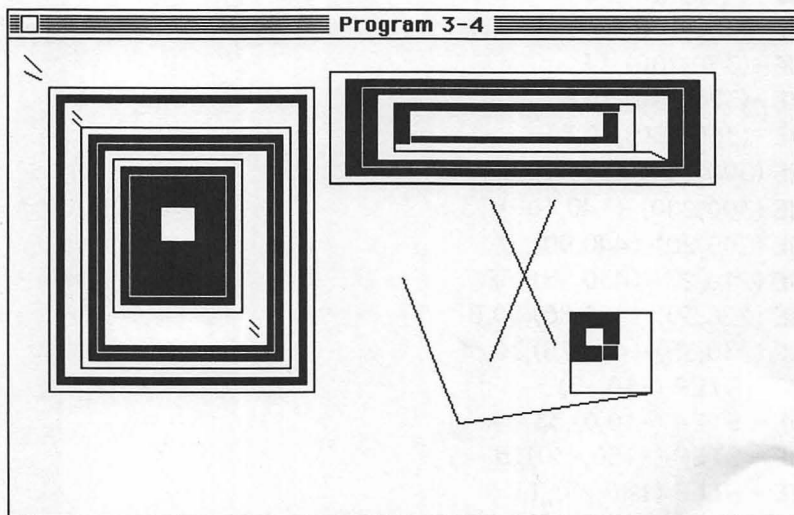
Program 3-4

The best way to see the different uses of **LINE** at work is to enter Program 3-4, one line at a time, running it after each line is typed in. This way you'll be able to see which line does what, and watch the creation of the drawing take place. You may want to resize the *List* window to make it fairly thin, and move it down to the bottom of the screen, where it will be out of the way.

44

T H R E E

CIRCLE

Creating straight lines and box shapes is fine, but it's not all the Macintosh can do in BASIC. A similar command in format, **CIRCLE** generates curves and circles:

CIRCLE (*horizontal location,vertical location*),*radius*
CIRCLE STEP (*horizontal offset,vertical offset*),*radius*

Each of these two basic formats has an additional 11 forms, which are produced simply by appending one of the following immediately after the value for *radius*:

,color
,,,end
,color,,end
,,start,end
,color,start,end
,,,,aspect
,color,,,aspect
,,,end,aspect
,color,,end,aspect
,,start,end,aspect
,color,start,end,aspect

- **CIRCLE** creates circles, ellipses, curves, and pie sections. All these shapes have a center reference point defined by *horizontal* and *vertical locations*, or as an *offset* from the last plotted point. If the last command was a **CIRCLE** command, the offset is relative to the center point of that circle. All subsequent commands which define a first point as an offset will have the point defined relative to the center specified by **CIRCLE**.
- A *radius*, the distance from the center to the edge of the circle, must be specified when using **CIRCLE**.
- If *color* is omitted or is 33, the circle is drawn black (30 indicates white).
- If an ellipse is drawn, then an *aspect* ratio must be defined (for circles, this is 1). If no aspect ratio is given, 1 is used. A value greater than 1 will form an ellipse that is tall and thin with its height equal to twice the radius; a value less than 1 will form an ellipse that is short and wide with its width equal to twice the radius.
- A curve is either a part of a circle or an ellipse. By specifying the *start* and *end* of the circle or ellipse, a curve can be

45

drawn. The start and the end are measured in radians—a circle has approximately 6.28 radians for one complete revolution (2 * π radians). A circle has 360 degrees for one complete revolution—degrees can be converted to radians by multiplying the degrees by π, or 3.14, and then dividing by 180. The end can be given without the start. In that case, zero will be used as the start. The zero degree or zero radian position is at the right and its measurement increases clockwise from that point. Pie sections can be drawn by making the start and end values negative. They are treated as though they were positive, but lines are drawn from the endpoints to the center of the circle or ellipse.

Program 3-5 includes several demonstrations of the different forms of the **CIRCLE** command.

Program 3-5. CIRCLE

```
CLS
CIRCLE (120,120),100
CIRCLE (120,120),95,33
CIRCLE (120,120),90,,,4
CIRCLE (120,120),85,,.3,2
CIRCLE (120,120),80,33,.3,4.3
CIRCLE (120,120),75,,,,.8
CIRCLE (120,120),70,,,5,.8
CIRCLE (120,120),65,,1,5,.8
CIRCLE (120,120),60,33,2,4,.8
CIRCLE (120,120),40,33,,-2
CIRCLE (120,120),40,33,-2,-4
CIRCLE (120,120),40,33,-4,-4.5
CIRCLE (120,120),40,33,-4.5,-5.5
CIRCLE (120,120),40,33,-5.5,-6.28
CIRCLE (300,100),50
FOR I=1 TO 15
CIRCLE STEP (5,5),50
NEXT I
FOR I=0 TO 4000:NEXT I
```

THREE

Figure 3-5. *Circles, ovals, arcs, and pie shapes can be drawn with CIRCLE.*
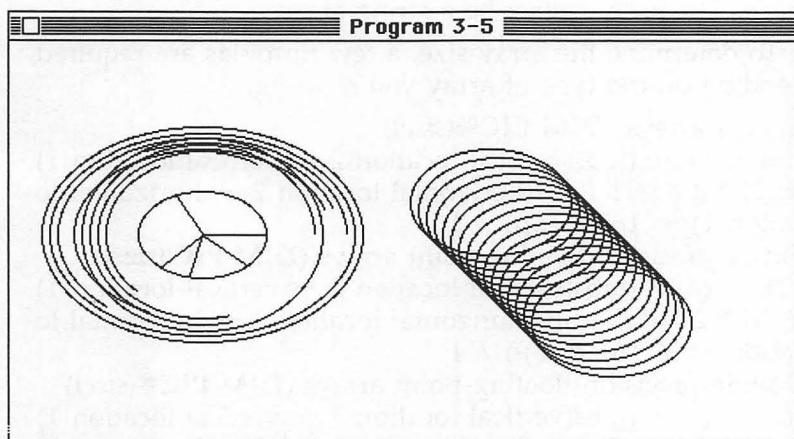
As with **LINE** in Program 3-4, the best way to see **CIR-CLE**'s power is to enter each line, pausing just long enough to run the program. Then type in another line and rerun the program. By doing this, you'll be able to note what each **CIRCLE** statement accomplishes. Experiment by changing some of the values and see what you come up with.

GET

Animation, as you've already seen, is possible with the Macintosh. But by using a pair of BASIC commands, you'll be able to animate shapes quickly and easily. The first such command is called **GET**:

GET (*horizontal location 1,vertical location 1*)–(*horizontal location 2,vertical location 2*),*array name*

- **GET** is used to read a bit image or shape already on the screen and store that picture in the array specified by *array name*.
- The shape must be within the area bounded by the coordinates specified by (*horizontal location 1,vertical location 1*) and (*horizontal location 2 ,vertical location 2*). These coordinates specify the two opposite corners of the area of the *Output* window to save into the array.

THREE

- The array must be defined with a **DIM** statement before using **GET**, and dimensioned large enough to contain the bit image. The array *cannot* be a string array.

  To determine the array size, a few formulas are required, depending on the type of array you're using:

  1. Integer arrays (**DIM PIC%(***size***)**)
     size = (4 + ((abs(vertical location 2 − vertical location 1) + 1) * 2 * INT ((abs(horizontal location 2 − horizontal location 1) + 16) / 16))) / 2
  2. Single-precision floating-point arrays (**DIM PIC!(***size***)**)
     size = (4 + ((abs(vertical location 2 − vertical location 1) + 1) * 2 * INT ((abs(horizontal location 2 − horizontal location 1) + 16) / 16))) / 4
  3. Double-precision floating-point arrays (**DIM PIC#(***size***)**)
     size = (4 + ((abs(vertical location 2 − vertical location 1) + 1) * 2 * INT ((abs(horizontal location 2 − horizontal location 1) + 16) / 16))) / 8

PUT

GET's counterpart is the **PUT** command:

PUT (*horizontal location 1,vertical location 1*),*array name*
PUT (*horizontal location 1,vertical location 1*),*array name,draw mode*
PUT (*horizontal location 1,vertical location 1*)–(*horizontal location 2,vertical location 2*),*array name*
PUT (*horizontal location 1,vertical location 1*)–(*horizontal location 2,vertical location 2*),*array name,draw mode*

- **PUT** draws the bit image stored in the array specified by *array name* at the location given by the coordinates defined by *horizontal location 1* and *vertical location 1*, which represent the upper left corner of the bit image.
- A *draw mode* can be specified to control how the bit image is to be drawn. The default draw mode is called **XOR**, where all background pixels are inverted wherever there is a black pixel in the image. Drawing images *twice* in the same place *before* moving the image to another location preserves the background—very useful for animation. Another mode is **AND**, where only the pixels black in both the background and the image are drawn black. All other pixels are drawn white. The third mode is **OR**, which is used to superimpose the bit image onto the background of the *Output* window. All black background pixels remain black while all black image

48

pixels are drawn on top of the background. Another mode, **PSET**, can be used when no consideration for the background is required. It causes the bit image to replace whatever was previously located at the point where the image is drawn. The **PRESET** mode is also used without concern for the background. It, however, draws the inverse of the bit image onto the screen.

- A second set of coordinates may be specified in **PUT** with *horizontal location 2* and *vertical location 2*. This is necessary only when the bit image is going to be drawn in a size different from that of the original read with the **GET**. The second coordinates specify the lower right corner of the area to draw the bit image into. The bit image will be scaled and drawn to fit that area.

   Program 3-6 is a demonstration of the **GET** command and the **PUT** command.

Program 3-6. GET and PUT

```
DIM MN(20)
CLS
PRINT CHR$(217)
GET (0,0)-(15,16),MN
PUT (0,0),MN
FOR I=300 TO 150 STEP -1
PUT (I,50),MN
PUT (I,50),MN
NEXT I
FOR I =2 TO 14
PUT (150-5*I,50)-(155,58+8*I),MN
PUT (150-5*I,50)-(155,58+8*I),MN
NEXT I
PUT (150-5*I,50)-(155,58+8*I),MN
```

   The program animates the figure in two stages, first by moving it across the screen, then by making it seem to move closer by drawing it larger and larger until it's several times its original size.

THREE

Figure 3-6. *Enlarging a bit image with PUT causes jagged edges on the shape.*
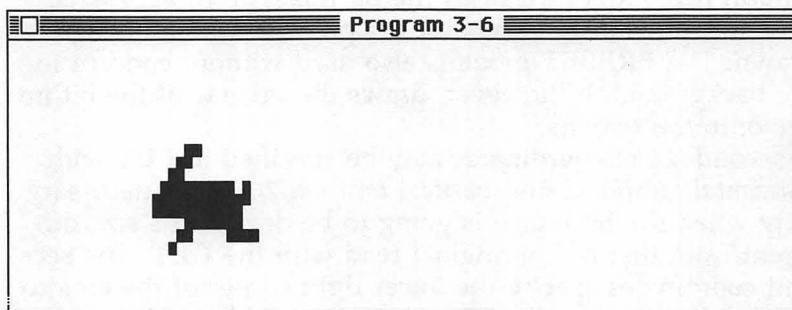
The program's first line dimensions the array MN which will contain the bit image of the figure. The figure itself is displayed using **CHR$(217)**, then stored with **GET** in the following line. **PUT** erases the shape by drawing it atop itself. Note that the *draw mode* is not indicated—by default, **XOR** is thus used.

The initial stage of the animation is performed by the first **FOR-NEXT** loop: I represents the horizontal position of the figure. The figure is repeatedly drawn and erased with the two successive **PUT** commands.

The second part of the animation process takes place in the second **FOR-NEXT** loop. This time I represents the proportionate size of the figure. Again, the two **PUT**s draw and erase the figure until it's seven times its original size. The program's last **PUT** draws the shape in its final size.

POINT

One of the more complicated tasks associated with animation is determining collisions between shapes. The BASIC command **POINT** greatly assists in this job:

Numeric variable = **POINT**(*horizontal location,vertical location*)

POINT examines the pixel defined by the *horizontal* and *vertical locations*. A value of 30 is returned if that pixel is white; 33 is returned if the pixel is black.

Program 3-7 illustrates one method of using **POINT** to detect collisions during movement and animation.

Program 3-7. POINT

```
DEFINT A-Z
DIM EXPL(20),ROCKET(10)
CLS
FOR I=0 TO 17:READ EXPL(I):NEXT I
FOR I=0 TO 8:READ ROCKET(I):NEXT I
LINE (0,250)-(489,250)
LINE (0,254)-(489,254)
LC=70
FOR I=1 TO 34
  READ HT
  LINE (LC,250)-(LC+10,200-10*HT),33,BF
  FOR J=1 TO HT
    XLC=LC+2+INT(RND(1)*9)
    YLC= 245-INT(RND(1)*(HT+4)*10)
    LINE (XLC,YLC)-(XLC+1,YLC+1),30,BF
  NEXT J
  LC=LC+10
NEXT I
HL=470
WHILE POINT (HL-1,40)<>33
  PUT (HL,40),ROCKET
  HL=HL-2
  PUT (HL+2,40),ROCKET
WEND
PUT (HL-2,36),EXPL,OR
FOR I=1 TO 600:NEXT I
PUT (HL-2,36),EXPL
FOR I=1 TO 3000:NEXT I
END
DATA 16,16,82,648,288,1152,6240,29344,-17144,-14080,-478
4,20480,11680,2576,832,16,72,34
DATA 16,7,3,782,4095,-2,4095,782,3
DATA 5,5,5,3,18,18,3,2,2,2,2,4,7,7,7,9,8,6,6,1,1,1,1,1,2,3,6,6,3
,3,2,2,1,4
```

When you run Program 3-7, you'll see a rocket moving from right to left. In a matter of seconds, the rocket smashes into one of the taller buildings, and you'll see an explosion.

Figure 3-7. *The* Output *window's display just prior to the rocket's impact. Animation is performed with a sequence of PUTs.*
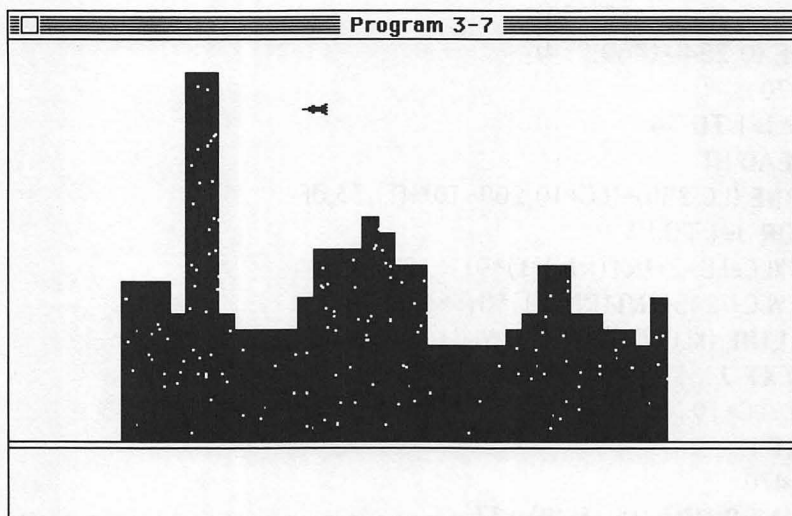
Program 3-7

Let's take a more detailed look at how this program works. The first line sets all the variables to integer variables, increasing the program's speed and minimizing memory requirements. The next line dimensions the arrays EXPL and ROCKET, which will be used to store the bit images of the explosion and rocket shapes. Both arrays are dimensioned a little larger than necessary. That's not a bad idea, for then, if you wanted, you could change the shape somewhat without redimensioning the arrays. This practice also allows a more modular program structure. Conceivably, one programmer could write the program while another drew the figures.

The two **READ** statements in the next couple of lines read in the data which defines the figures' bit images from **DATA** statements near the end of the program. (The procedure for creating the data images is discussed a bit later.) The city baseline is created with the two **LINE** statements, and the next few lines define the buildings (the first one is defined by LC), cre-

ate them, and place and display their windows. Variables such as HT (building height), XLC (horizontal window coordinates), and YLC (vertical window coordinates) are defined and used. The **RND** function is used to pick random locations for the windows. Near the end of the first indented **FOR-NEXT** loop, notice that LC is incremented by 10; this is the location of the next building.

The **WHILE-WEND** loop draws the rocket until it crashes into a building. HL represents the rocket's horizontal position—it starts out at 470. Its vertical position is always 40.

Checking for a collision, the purpose of all of this, is handled by the **WHILE POINT** $(HL-1,40)<>33$ statement. The point immediately to the left (in front of, in other words) is always at $HL-1$. If there is no building in the way—if the pixel to the rocket's left is anything *but* black—the two **PUT**s erase and redraw the rocket, causing the animation. When the rocket finally crashes into a building, the **PUT**s after the **WEND** are executed, drawing the explosion, then erasing it, leaving a gap in the building. Since the explosion is larger than the rocket, it's drawn at higher coordinates to center itself with the rocket's trajectory.

LCOPY

Only one Microsoft BASIC command used by the utilities discussed later still needs to be mentioned, and mentioned only briefly:

LCOPY

This command prints the entire screen display on the ImageWriter (or LaserWriter) printer. If you have your printer connected and its power turned on, insert **LCOPY** immediately before the **END** command in Program 3-7 (remember to separate them with a colon). You'll have a hardcopy of the final cityscape.

Play around with Program 3-7 and **LCOPY**; experiment with placing it in different parts of the program and seeing what printouts result.

CHAPTER

4
Bit Image Design

4
Bit Image Design

The rocket shape in Program 3-7 was designed with the help of a software utility which lets you draw any desired shape in magnified view and then calculates the numbers that must be stored in the bit image array. It also shows the size of the array required for shapes up to 64 pixels wide and 48 pixels high. This software utility, "The Shape Editor," is in Chapter 11.

Let's take a look at how bit image arrays are constructed. We'll concentrate on a smaller image, one only 16 pixels wide × 16 pixels high.

In the middle of Figure 4-1 there's a grid, 16 squares across and 16 squares high. Each square represents a pixel on the screen. Each row is numbered on the left, row 1 being at the top. The leftmost square of row 1 is the pixel specified when using **PUT**. The rest of the shape is drawn relative to that point. Each row is divided into columns with values as indicated above the grid.
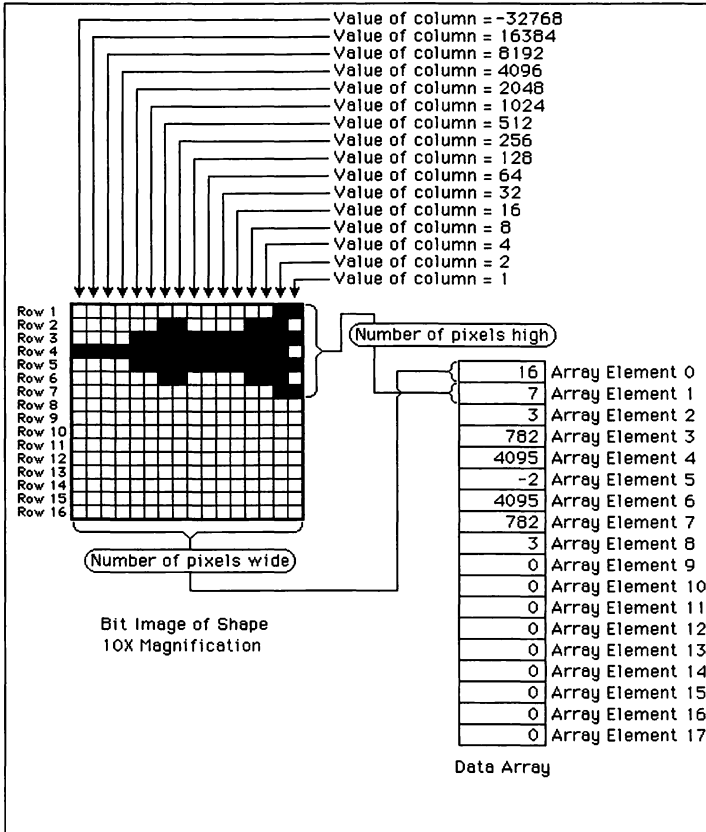
To create a shape (say, if you're using graph paper instead of the Shape Editor), draw the bit image by darkening the squares to be turned on, or set to black.

In the case of the rocket in Figure 4-1, the width of the shape is 16 pixels and its height is 7 pixels. These two numbers go into the array in positions 0 and 1, respectively. The array must be an integer array, done by using **DEFINT** A–Z which defines all variables as integers, or by explicitly defining the array with **DIM** *array%(size)*. The percent symbol (%) makes it an integer array.

Each row of the shape which includes a blackened square (pixel) has a value equal to the sum of the column values. For example, row 1 has black pixels in columns which have values of 2 and 1. Thus, the value of row 1 is 2 + 1, or 3. Similarly, the value of row 2 is 512 + 256 + 8 + 4 + 2, or 782. Calculate the values of all 16 rows. Store the values for rows 1 to 16 in the array elements 2 to 17, respectively. For shapes with

more than 16 rows, simply extend the grid and dimension the array to the number of rows + 1.

Figure 4-1. *Data definition of bit image graphics.*

The figure contains the following data:

Column values (top to bottom):
- Value of column = -32768
- Value of column = 16384
- Value of column = 8192
- Value of column = 4096
- Value of column = 2048
- Value of column = 1024
- Value of column = 512
- Value of column = 256
- Value of column = 128
- Value of column = 64
- Value of column = 32
- Value of column = 16
- Value of column = 8
- Value of column = 4
- Value of column = 2
- Value of column = 1

Rows 1 through 16 (Bit Image of Shape, 10X Magnification), Number of pixels high, Number of pixels wide.

Data Array:

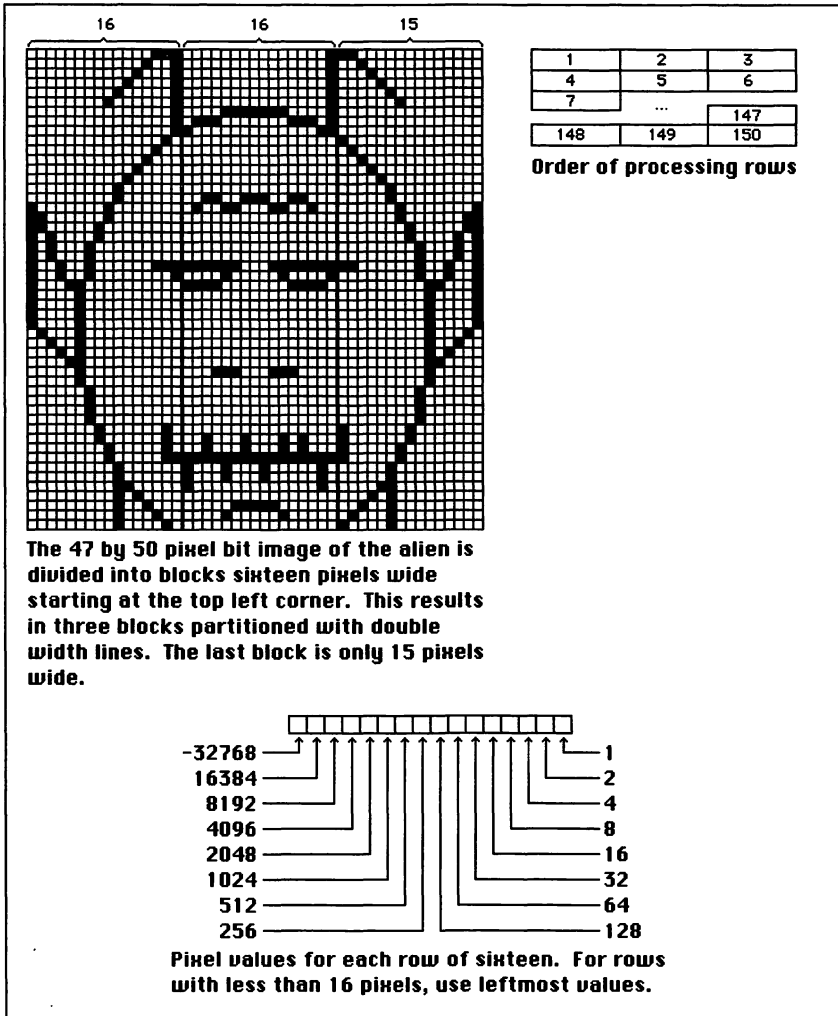| Value	Element
16	Array Element 0
7	Array Element 1
3	Array Element 2
782	Array Element 3
4095	Array Element 4
-2	Array Element 5
4095	Array Element 6
782	Array Element 7
3	Array Element 8
0	Array Element 9
0	Array Element 10
0	Array Element 11
0	Array Element 12
0	Array Element 13
0	Array Element 14
0	Array Element 15
0	Array Element 16
0	Array Element 17

Data Array

Larger Images

The process for generating larger shapes is somewhat similar. The major difference is in understanding the order of processing rows when you're designing a multiple-row shape. (Each row consists of 16 pixels across.) Take a look at Figure 4-2.

Figure 4-2. *The above process is used to calculate the block graphics data for large, irregular shapes.*

16	16	15

1	2	3
4	5	6
7	...	
		147
148	149	150

Order of processing rows

The 47 by 50 pixel bit image of the alien is divided into blocks sixteen pixels wide starting at the top left corner. This results in three blocks partitioned with double width lines. The last block is only 15 pixels wide.

-32768	1
16384	2
8192	4
4096	8
2048	16
1024	32
512	64
256	128

Pixel values for each row of sixteen. For rows with less than 16 pixels, use leftmost values.

As before, the first step is drawing the bit image—in this case, a 47 × 50 pixel image of an alien. The width is divided into units of 16 pixels (the number of bits in an integer), starting from the top left. Figure 4-2 is divided into three columns, the first two 16 pixels wide and the last 15 pixels wide.

FOUR

The data for this image is generated by evaluating the bit strips of each column left to right and top to bottom. The pixel values for each bit in the bit strip are -32768 for the leftmost and 1 for the rightmost pixel. Bit strips with less than 16 pixels are evaluated with the leftmost pixel equal to -32768.

The image in Figure 4-2 requires 150 integers for its data, plus two more for the image width and height. An array of 152 elements is thus needed, with the first element containing the width, the second the height, and the remaining elements containing the actual image data.

Program 4-1 animates the bit image in Figure 4-2. Click the mouse to stop the program.

Program 4-1. Bit Image

```
CLS:DEFINT A-Z
DIM PIC%(151)
FOR I=0 TO 151:READ PIC%(I):NEXT I
LINE (0,0)-(489,200),33,BF
FOR I=10 TO 190 STEP 20:LINE (0,I)-(489,I+1),30,BF:NEXT I
FOR I=0 TO 4:FOR J=52 TO 452 STEP 50:LINE (J,I*40-10)-STE
P(2,19),30,BF:NEXT J,I
FOR I=0 TO 4:FOR J=26 TO 476 STEP 50:LINE (J,I*40+10)-STE
P(2,19),30,BF:NEXT J,I
FOR I=0 TO 225 STEP 15
   LINE (245-I,200)-(245-4*I,300):LINE (245+I,200)-(245+4*I,3
00)
NEXT I
LINE (40,140)-(100,199),30,BF:LINE (130,140)-(190,199),30,BF
:LINE (220,140)-(280,199),30,BF:LINE (310,140)-(370,199),30,
BF:LINE (400,140)-(460,199),30,BF
LINE (40,140)-(100,199),,B:LINE (130,140)-(190,199),,B:LINE (
220,140)-(280,199),,B:LINE (310,140)-(370,199),,B:LINE (400,1
40)-(460,199),,B
FOR I=1 TO 2000:NEXT I
FOR I=1 TO 50
   PIC%(1)=I
   PUT (45,199-I),PIC%,PSET:PUT (135,199-I),PIC%,PSET:PUT (
225,199-I),PIC%,PSET:PUT (315,199-I),PIC%,PSET:PUT (405,19
9-I),PIC%,PSET
```

```
NEXT I
WHILE MOUSE(0)<1:WEND
END
DATA 47,50
DATA 7,1,-16384
DATA 9,1,8192
DATA 17,1,4096
DATA 33,1,2048
DATA 65,1,1024
DATA 129,1,512
DATA 1,4065,0
DATA 1,28701,0
DATA 1,-32765,0
DATA 2,0,-32768
DATA 4,0,16384
DATA 8,0,8192
DATA 16,0,4096
DATA 32,0,2048
DATA 64,0,1024
DATA 64,13208,1024
DATA -32640,19556,514
DATA -16256,0,518
DATA -16128,0,262
DATA -24320,0,266
DATA -24064,0,138
DATA -24064,0,138
DATA -28153,-897,-16238
DATA -28159,2081,146
DATA -29696,-4066,98
DATA -31744,0,66
DATA -31744,0,66
DATA -31744,0,66
DATA -31744,0,66
DATA 17408,0,68
DATA 9216,0,72
DATA 5120,0,80
DATA 3072,0,96
DATA 1024,7280,64
```

DATA 1024,0,64
DATA 512,0,128
DATA 512,0,128
DATA 512,0,128
DATA 512,0,128
DATA 258,0,-32512
DATA 258,8740,-32256
DATA 130,8740,-32256
DATA 131,-1,-32256
DATA 64,-30574,1024
DATA 64,-30574,1024
DATA 96,-32766,3072
DATA 80,0,5120
DATA 72,1984,9216
DATA 68,2080,17408
DATA 66,0,-31744

This program draws a brick wall with five square mouse holes. After a short pause, five aliens rise from their hiding places.

Figure 4-3. *Five aliens rise from their hiding places.*

F O U R
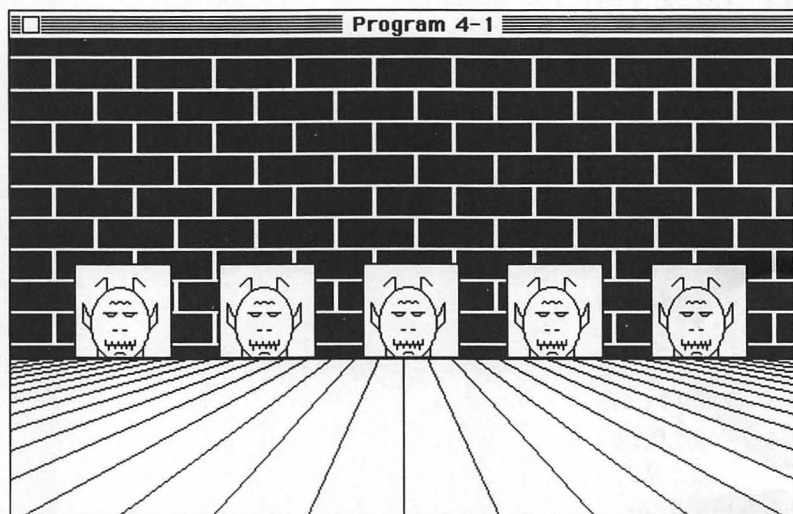
The second line dimensions PIC%(151) to contain the bit image in Figure 4-2. The data elements for the bit image are stored in **DATA** statements at the end of the program, read by the third line. A number of **LINE** statements then draw the wall, floorboards, mouse holes, and frames around those holes.

The last **FOR-NEXT** loop animates the five aliens. I controls the height of the bit image stored in the second element of the bit image array, PIC%(151). The image is drawn, starting one pixel above the base of the wall and redrawn one pixel higher with each iteration of the loop using **PUT** with the **PSET** option. **WHILE MOUSE**(0)<1:**WEND** loops until the mouse button is pressed, which terminates the program.

The first **DATA** statement contains the width and height of the bit image, and each of the remaining statements represents the bit image data for one row of pixels.

CHAPTER

5
File
Commands

5
File Commands

Microsoft BASIC on the Macintosh has a large set of commands which deal with disk file manipulation. One of the simplest is **FILES**:

FILES
FILES *"filename"*
FILES *"volume name:filename"*

    FILES lists the names of the files on the disk in the Macintosh's internal drive. If *filename* is given, Microsoft BASIC will list that particular file if it exists, or it will display the message *File not found* if it doesn't. A file on another disk drive may be listed by giving the *volume name* before the filename. For instance, if there's a disk named *Data Disk* in the external drive, then the file *DB.DATA* may be looked for with the command **FILES** "Data Disk:DB.DATA".

    A file can be deleted from within BASIC by using the **KILL** command:

KILL *filename*

Filename indicates the file to be deleted. It may include the name of the volume where the file is located, again separated from the filename by a colon.

    Files can also be renamed with the command **NAME**:

NAME *current filename* **AS** *new filename*

These names may include the volume name of the file if they are not on disks in the internal drive.

File Creation

Two types of files can be created and used by Microsoft BASIC—*sequential access files* and *random access files*. Sequential access files may contain any number of variable-length data items, but random access files are divided into records of equal length. Each of the records may contain a set of data items, all of which are also of a predetermined length.

FIVE

Input from a sequential file has to begin from the first item of the file, while input from a random access file can begin with any record. Output to a sequential file must begin at the start of the file or after the last item. Output to a random access file can be placed into any record inside the file or after the last record. All files have a size restriction of 16 megabytes or fewer.

Before any file access can be performed, a memory buffer has to be allocated. Associated with this buffer is the file number and type of input or output to be used. The BASIC command **OPEN** does this:

Sequential access output
OPEN *"O,"#filenumber,filespec*
OPEN *"O,"filenumber,filespec*
OPEN *filespec* **FOR OUTPUT AS** *#filenumber*
OPEN *filespec* **FOR OUTPUT AS** *filenumber*

Sequential access output by appending to the file
OPEN *"A,"#filenumber,filespec*
OPEN *"A,"filenumber,filespec*
OPEN *filespec* **FOR APPEND AS** *#filenumber*
OPEN *filespec* **FOR APPEND AS** *filenumber*

Sequential access input
OPEN *"I,"#filenumber,filespec*
OPEN *"I,"filenumber,filespec*
OPEN *filespec* **FOR INPUT AS** *#filenumber*
OPEN *filespec* **FOR INPUT AS** *filenumber*

Random access input and output
OPEN *"R,"#filenumber,filespec*
OPEN *"R,"filenumber,filespec*
OPEN *"R,"#filenumber,filespec,record length*
OPEN *"R,"filenumber,filespec,record length*
OPEN *filespec* **AS** *#filenumber*
OPEN *filespec* **AS** *filenumber*
OPEN *filespec* **AS** *#filenumber,***LEN** = *record length*
OPEN *filespec* **AS** *filenumber,***LEN** = *record length*

- *Filenumber* can be any number from 1 to 255. If sufficient memory is available for buffers, 255 files could be opened at the same time.
- *Filespec* is a string containing the name of the file. It may include the name of the volume where the file is located, separated from the filename by a colon. *Filespec* may also be a

string containing a volume name only, as in the case of a device like *COM1:*, which is the serial port.
* Random access files require a *record length* value defining the size of a record in bytes. The maximum size of a record is 32,767 bytes. If no record length is given, 128 is used.

Sequential File Output

Depending on what kind of control you want over the format of the final product, output to sequential files can be performed with a variety of BASIC commands. One such command is **PRINT#**, which writes data into a sequential file:

PRINT#*filenumber,printlist*
PRINT#*filenumber,***USING** *"using string,";printlist*

* **PRINT#** outputs data to a sequential file opened with the buffer specified by *filenumber*.
* *Printlist* is a list of any combination of numbers, strings, and variables which represent the data items to be stored in that file.
* A **USING** option can be included to format the output to the file.

To write the data items into the correct variables, it's necessary to note a few guidelines when using **PRINT#** with more than one item in the *printlist.* There are some complications saving the correct delimiters between items in the file because delimiters differ for numeric and string variables. For numbers, delimiters are carriage returns, linefeeds, commas, semicolons, and spaces. Leading spaces are ignored and imbedded tabs are ignored when reading numbers. To be safe, separate numeric values and variables with commas or semicolons (semicolons conserve space). For example, a typical **PRINT#** statement would be **PRINT#**1,SUBTOTAL;TAX; TOTAL. A **PRINT#** command with a form similar to **PRINT#**1, **USING** "######,";A;B;C;D is also a good example to follow when storing numerics.

String output can be delimited by carriage returns, linefeeds, commas, or a pair of quotation marks. Quotation marks are required if, upon input, the string is to contain carriage returns, linefeeds, commas, or leading spaces. Leading spaces are otherwise ignored at the time of input.

PRINT# is useful when complete control of the format of the output is required. However, the BASIC command WRITE# makes the task of writing to sequential files simpler:

WRITE#*filenumber,printlist*

WRITE# stores the items listed in the *printlist* to a sequential file opened with the buffer number specified by *filenumber*. The items in the printlist must be separated by commas. WRITE# automatically inserts commas between output items and also places quotes around strings.

A carriage return and linefeed are inserted into a file at the end of a WRITE# command and at the end of a PRINT# command (if there is no semicolon at the end of the PRINT#). However, a carriage return and linefeed may be forced into the file at increments defined by WIDTH#. Its syntax is

WIDTH#*filenumber*
WIDTH#*filenumber,linewidth*
WIDTH#*filenumber,,printzonewidth*
WIDTH#*filenumber,linewidth,printzonewidth*

- WIDTH# adjusts the line width and print zone width of a sequential file opened with the buffer number specified by *filenumber*.
- *Linewidth* defines the maximum width of a line of output. If *linewidth* is not given, 255 is used. This differs from values of 1 to 254 because a carriage return and linefeed will *never* be automatically inserted into the output.
- *Printzonewidth* is the default spacing between data items when printed with PRINT# and WIDTH# commands. PRINT# is affected by WIDTH# when the items in the printlist are separated by commas. WIDTH# is particularly useful when the tabular output is going to the printer.

Sequential File Input

You have just as much control over format when you're reading items from a sequential file as when you're writing. A number of BASIC commands are available, and you can choose which you'll use depending on what you want to see as the result:

INPUT#*filenumber,variable list*

- INPUT# reads data items from a sequential file opened with the buffer number specified by *filenumber*.

- The data items are read into the variables specified by *variable list*.
- Numeric items are delimited by carriage returns, linefeeds, commas, semicolons, and spaces. Leading spaces and imbedded tabs are ignored when reading values into numeric variables. Items in the variable list are separated by commas if the input data is delimited by commas, semicolons, or spaces. Data items delimited by carriage returns and linefeeds must be read with separate **INPUT#** commands.
- If the data is in the form of strings, it may be delimited by carriage returns, linefeeds, commas, or a pair of quotation marks. BASIC terminates string input after it encounters a string delimiter on the end of the file.

You can also use **LINE INPUT#**, which reads an entire line and doesn't need delimiters other than carriage returns and linefeeds:

LINE INPUT#*filenumber,string variable*

LINE INPUT# reads a string from the sequential file specified by *filenumber* and stores that string in *string variable*. The command reads all the characters in the file until it comes across either a carriage return or linefeed, the two delimiters. Each subsequent **LINE INPUT#** reads another string from the file, unless the end of the file is encountered. **LINE INPUT#** is particularly useful for reading data into BASIC programs from applications which can save data in ASCII format.

Another sequential file BASIC input command is **INPUT$**:

String variable = **INPUT$**(*number*)
String variable = **INPUT$**(*number,filenumber*)
String variable = **INPUT$**(*number,#filenumber*)

- **INPUT$** reads a string of characters from a sequential file specified by *filenumber*.
- *Number* controls the number of characters to read. When no file number is specified, the keyboard is used (in this case, the file is a device), a common use for **INPUT$**. For example, the serial port can be checked for a character and then read with a command like CH$ = **INPUT$**(1,#1) after a buffer has been opened with a command like **OPEN** "COM1:" **FOR INPUT AS** #1.

F I V E

Random Access Input and Output

Using random access input and output is a bit more compli-
cated and requires some preparation. Fields within a record
must be defined after opening a random access file and before
reading or writing to the file. The command for this is **FIELD**:

FIELD #*filenumber,fieldwidth 1* **AS** *field variable 1*
FIELD *filenumber,fieldwidth 1* **AS** *field variable 1*

- **FIELD** specifies the sizes of the fields within a record in the
  file marked by *filenumber*.
- The value of *fieldwidth 1* is the width of the first field—the
  number of characters—accessed with *field variable 1*. Field
  variables are string variables.
- If more fields are required, add *,fieldwidth 2* **AS** *field variable
  2* or as many more field declarations as needed. If it was nec-
  essary to define three equal fields within a record with a
  length of 90 bytes, for instance, a command like **FIELD**
  #1,30 **AS** SUBTOTAL$,30 **AS** TAX$,30 **AS** TOTAL$ could be
  used. A command like **OPEN** "R,"#1,"CASH SALES,"LEN
  = 90 would precede the above **FIELD** command. The sum of
  the field widths cannot exceed the record length. Attempting
  to do so causes a *Field overflow* error.
- Multiple **FIELD** commands can be defined for a file which
  contains records that are to be interpreted differently.

Before a record can be written to a file, the field variables
must be assigned their values. There are two commands for
doing this—**LSET** and **RSET**:

LSET *field variable* = *string*
LSET *field variable* = *string variable*
RSET *field variable* = *string*
RSET *field variable* = *string variable*

Both commands store strings into field variables. If the strings
are not the same length as the width of the field variable,
strings too long are truncated on the right until they're the
same size as the field variable, and strings too short are pad-
ded with spaces. **LSET** pads spaces to the right of the string to
left justify it in the field variable, while **RSET** pads spaces to
the left to right justify the string. These commands can be
used with strings which are not part of a **FIELD** declaration
for justifying strings for tabular output.

Sometimes strings being assigned to field variables origi-
nate from numeric values. There are three commands that

pack numbers into strings: **MKI$**, **MKS$**, and **MKD$**.
String variable = **MKI$**(*integer*)
String variable = **MKS$**(*single-precision number*)
String variable = **MKD$**(*double-precision number*)

These commands pack numbers stored according to their types. Integers are packed into strings two characters long, single-precision numbers are packed into strings four characters long, and double-precision numbers are packed into strings eight characters long. Single- and double-precision numbers must be in packed decimal format as if they were created in Financial Microsoft BASIC 2.0.

Since BASIC 2.0 comes in two versions, one with packed decimal representation of single- and double-precision numbers and one with binary representation of single- and double-precision numbers, two more numeric-to-string conversion commands are required. These commands are **MKSBCD$** and **MKDBCD$**:

Single-precision binary numbers
String variable = **MKSBCD$**(*single-precision number*)

Double-precision binary numbers
String variable = **MKDBCD$**(*double-precision number*)

These commands pack binary-represented numbers into strings which require four bytes for single-precision numbers and eight bytes for double-precision numbers. BCD stands for Binary Converted to Decimal prior to being converted into a string format. These commands are typically used in Engineering BASIC 2.0 programs or for updating data within files in Financial BASIC 2.0.

After a record has had values assigned to all of its appropriate fields, it can be written to the file. This is done with **PUT**:

PUT *filenumber*
PUT #*filenumber*
PUT *filenumber,record number*
PUT #*filenumber,record number*

PUT writes the buffer (opened with the number specified by *filenumber*) to the record specified by *record number* (of its associated random access file). If no record number is given, the value used depends upon whether or not a **PUT** to the file has already been performed. If a **PUT** has been executed, the

record after the last record written to will be updated. Otherwise, record 1, the first record, will be written to.

Conversely, the command to read a record from a random access file is **GET**:

GET *filenumber*
GET *#filenumber*
GET *filenumber,record number*
GET *#filenumber,record number*

GET reads the record specified by *record number* (of its associated random access file) into the buffer (opened with the number specified by *filenumber*). If no record number is given, the value used depends upon whether or not a **GET** from the file has already been performed. If a GET from that file has been executed, the record after the last record read will be input. Otherwise, record 1, the first record, will be read.

If the data read into the fields of the record are numeric and packed into strings with **MKI$**, **MKS$**, or **MKD$**, the corresponding commands to unpack them are **CVI**, **CVS**, and **CVD**:

Integer variable = **CVI**(*packed integer string*)
Single-precision variable = **CVS**(*packed single-precision string*)
Double-precision variable = **CVD**(*packed double-precision string*)

Counterparts to the **MKSBCD$** and **MKDBCD$** commands are **CVSBCD** and **CVDBCD**:

Single-precision variable = **CVSBCD**(*packed single-precision string*)
Double-precision variable = **CVDBCD**(*packed double-precision string*)

When a file is no longer needed, its buffer memory is freed with **CLOSE**:

CLOSE
CLOSE *filenumber*
CLOSE *#filenumber*
CLOSE *filenumberlist*

- **CLOSE** closes the buffer opened with the number specified by *filenumber*.
- More than one file can be closed by using a *filenumberlist*, where all file buffer numbers to be closed are listed and separated by commas. File buffers 1, 2, and 5 could be closed with a command such as **CLOSE** #1,#2,#5. If no filenumber is specified, then all files are closed.

Another command to close all files is **RESET**:

RESET

F I V E

Trying Things Out

Now that you've seen the various commands provided by BASIC for file creation and manipulation, let's take a look at some programs.
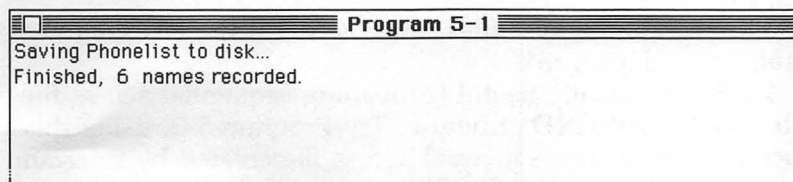
Creating a sequential file is perhaps the easiest, so let's start there. Type in and run Program 5-1.

Program 5-1. Creating a Sequential File

```
CLS
PRINT "Saving Phonelist to disk..."
OPEN "O",#1,"Phonelist"
READ COUNT
FOR I=1 TO COUNT
  READ LASTNAME$,FIRSTNAME$,PHONENUMBER$
  WRITE #1,LASTNAME$,FIRSTNAME$,PHONENUMBER$
NEXT I
CLOSE #1
PRINT "Finished,";COUNT;"names recorded."
END
DATA 6
DATA "Taylor","Cathy","348-7219"
DATA "Smith","Sam","246-1821"
DATA "Cheslam","Elizabeth","624-2843"
DATA "Jones","Fred","843-0922"
DATA "Bell","Jane","762-6452"
DATA "Hoden","Paul","923-4475"
```

This program takes a list of names and phone numbers and saves them in a sequential file called *Phonelist*.

Figure 5-1. *This message indicates that the file was successfully created.*

```
┌──────────────────── Program 5-1 ────────────────────┐
│ Saving Phonelist to disk...                          │
│ Finished, 6 names recorded.                          │
│                                                      │
│                                                      │
│                                                      │
└──────────────────────────────────────────────────────┘
```

The file buffer is created with **OPEN** and assigned the file number 1. **READ** reads a value of 6 (from the first **DATA** statement) into the variable COUNT, which thus contains the number of names and phone numbers to be placed in the file. Each of these names and phone numbers is in **DATA** statements at the end of the program. You can modify these to create a list of any number of names and phone numbers simply by changing the information in the **DATA** statements. Make sure that the first piece of data read into COUNT is the number of name/phone number combinations. Add **DATA** statements if you like.

The **FOR-NEXT** loop reads the information from the **DATA** statements and then writes it to the file. **READ** LASTNAME$,FIRSTNAME$,PHONENUMBER$ does the reading, while the next line (which uses **WRITE**) stores this information to the disk file. **CLOSE** #1 closes the *Phonelist* file and updates the last output to the buffer onto the disk. Failure to close a sequential file may result in not having the last output items recorded to disk and in not having the disk directory updated (on the Macintosh, the directory is an invisible file called the Desktop). The final line before the program ends displays the number of names saved to the disk. If the program stops before this message is displayed, you can assume that there was some problem with writing to the file and that some or all of the information may not be stored.

End of File

Often, there are problems when a program trys to read data beyond the end of a file. To determine when the end of the file has been reached, **EOF** can be used:

Boolean variable = **EOF**(*filenumber*)

EOF returns a value of true or false for the file specified by *filenumber*. After the last item of a sequential file has been read, **EOF** is true (-1).

With random access files, **EOF** is true if the last **GET** was unable to read a record.

EOF is especially useful for reading sequential access files with a **WHILE-WEND** structure. Try Program 5-2, using this structure to read the sequential access file created by Program 5-1. (Make sure that the file *Phonelist* is on the disk in the internal drive.)

F I V E

Program 5-2. *EOF*

```
CLS
COUNT =0
PRINT "Reading Phonelist from disk..."
OPEN "I",#1,"Phonelist"
WHILE NOT EOF(1)
  INPUT#1,LASTNAME$,FIRSTNAME$,PHONENUMBER$
  PRINT LASTNAME$,FIRSTNAME$,PHONENUMBER$
  COUNT = COUNT +1
WEND
CLOSE #1
PRINT "Finished,";COUNT;"names listed."
END
```

Figure 5-2. *The list of names stored in the phone list is displayed.*

```
▤□▦▦▦▦▦▦▦▦▦▦▦▦▦▦ Program 5-2 ▦▦▦▦▦▦▦▦▦▦▦▦▦▦
Reading Phonelist from disk...
Taylor         Cathy          348-7219
Smith          Sam            246-1821
Cheslam        Elizabeth      624-2843
Jones          Fred           843-0922
Bell           Jane           762-6452
Hoden          Paul           923-4475
Finished, 6 names listed.
```

The second line of the program initializes COUNT, which is used to count the number of names read from the phone list file, to 0. The **OPEN** statement opens the file *Phonelist* for sequential input, and the next four lines first read, then print the last name, first name, and phone number of each **DATA** entry. The **WHILE** command at the beginning of this loop instructs the program to continue to read the file until the end of file returns a -1. That's the purpose of the Boolean NOT. If **EOF** is false (0), the program keeps reading data. When **EOF** returns -1, the **WEND** part of the loop executes, closing the file and printing a message on the screen.

F I V E

Appending to a Sequential File

More names can be added to the phone list by appending them to those names already in the sequential access file. Program 5-3 adds more data to the *Phonelist* sequential access file.

Program 5-3. Appending Data

```
CLS
PRINT "Saving additional Phonelist names to disk..."
OPEN "A",#1,"Phonelist"
READ COUNT
FOR I=1 TO COUNT
  READ LASTNAME$,FIRSTNAME$,PHONENUMBER$
  WRITE #1,LASTNAME$,FIRSTNAME$,PHONENUMBER$
NEXT I
CLOSE #1
PRINT "Finished,";COUNT;"names added and recorded."
END
DATA 3
DATA "Brown","Jim","886-7337"
DATA "Marks","Donna","225-9029"
DATA "Kalahan","Tom","787-5681"
```

This program is quite similar to Program 5-1. The main difference is in the third line (**OPEN "A",#1,"Phonelist"**) which opens the file *Phonelist* with an append option. All output to the file will be placed *after* the last record already there.

Appending data to sequential access files is simpler than appending information to random access files. Overall, in fact, sequential access files are easier to deal with.

Figure 5-3. *This message indicates that the new names were successfully added to the phone list file.*

```
▤☐▦▦▦▦▦▦▦▦▦▦▦ Program 5-3 ▦▦▦▦▦
Saving additional Phonelist names to disk...
Finished, 3 names added and recorded.
```

F I V E
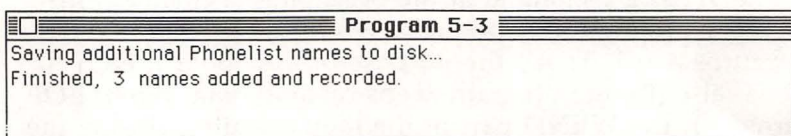
Searching Through a Sequential File

One problem with sequential access files, however, is that
they're not efficient when accessing specific information from
within the file. In the next example, Program 5-4, you'll see
that the accessing information requires that the file be re-
opened each time a search is made. (Make sure that the up-
dated *Phonelist* file is on the disk in the internal drive before
running Program 5-4.)

Program 5-4. Searching

```
CLS
NM$ = " "
WHILE LEN(NM$)>0
  LINE INPUT "Enter last name or RETURN to quit: ";NM$
  IF LEN( NM$)=0 THEN LOOPEND
  OPEN "I",#1,"Phonelist"
  WHILE (NOT EOF(1)) AND (NM$<>LASTNAME$)
    INPUT#1,LASTNAME$,FIRSTNAME$,PHONENUMBER$
  WEND
  IF NM$=LASTNAME$ THEN PRINT LASTNAME$,FIRSTNAME$,PHO
NENUMBER$ ELSE PRINT NM$;" is not in the file!"
  CLOSE #1 ·

LOOPEND:
WEND
END
```

After asking you to enter a last name, this program
searches through the *Phonelist* file and tries to find it. If it does
find the name, the program displays the last name, first name,
and phone number. Otherwise, it tells you that there's no such
name in the list. The program ends when no name is entered
and the Return key is pressed.

F I V E

Figure 5-4. *If the name you search for is on file, then the phone number is displayed. Otherwise, a* Name is not in the file! *message is given.*

```
▤▢▨▨▨▨▨▨▨▨▨▨▨▨▨  Program 5-4  ▨▨▨▨▨▨▨▨
Enter last name or RETURN to quit: Swift
Swift is not in the file!
Enter last name or RETURN to quit: Bell
Bell          Jane          762-6452
Enter last name or RETURN to quit:
```

Once the **CLS** is done, the program initializes NM$ to contain a space, required so that the test on the next line will be passed and execution of the program will continue after that. If the length of NM$ is 0, the **WHILE** command forces the program to jump to the last line, causing the program to end. NM$ will contain the name to search for.

The search portion of the program is contained within the **WHILE-WEND** loop (it's indented as a highlight). **LINE IN-PUT** inputs the search name into NM$, while the next line (**IF LEN....**) verifies that indeed a name was entered. When Return is the only response to **LINE INPUT**, the variable NM$ will be a null string. If a name is entered, it's searched for within the file *Phonelist*.

Notice that this search process must open the file each time a search is made. This is necessary to set the search position to the beginning of the file. Otherwise, input commences from the next data item after the last data item read.

The file is opened with **OPEN**. *Phonelist* (created by Program 5-1 and appended to by Program 5-3) is not sorted. If the method of maintaining a file when adding information is limited simply to appending new data, there's really no point in sorting the new data which will be added. If the new data was sorted before being added, there would be no guarantee that the entire file would still be sorted after the appending process. Thus, the entire file must be read until either the end of the file is reached or the name being searched for is found.

The second **WHILE-WEND** loop does this. The **WHILE** statement tests for the end of the file or a match of the last name with the name stored in NM$. Next, **INPUT#** reads the data from the file. The final **IF-THEN-ELSE** statement makes

the final test and prints either the full name and phone number or a message that the name was *not* in the file. **CLOSE** closes the file, even though more information may be required from it, since **OPEN** cannot reopen a file unclosed. If **CLOSE** was deleted, searching again (which necessitates reopening the file) would generate a *File already open* error message.

Note the label (LOOPEND) before the second-to-last line. This simply serves as a point for the program to continue if there's no name to look for.

Producing Random Access Files

The phone list generated by Program 5-1 as a sequential access file could have been created as a random access file instead. Take a look at Program 5-5, and compare it with Program 5-1.

Program 5-5. Random Access File

```
CLS
PRINT "Saving Phonelist to disk..."
OPEN "R",#1,"RPhonelist",40
FIELD #1,8 AS COUNT$
READ COUNT
LSET COUNT$ = MKD$(COUNT)
PUT #1
FIELD #1,15 AS LASTN$,15 AS FIRSTN$,8 AS PHONEN$
FOR REC=1 TO COUNT
  READ LASTNAME$,FIRSTNAME$,PHONENUMBER$
  LSET LASTN$ = LASTNAME$
  LSET FIRSTN$ = FIRSTNAME$
  LSET PHONEN$ = PHONENUMBER$
  PUT #1
NEXT REC
CLOSE #1
PRINT "Finished,";COUNT;"names recorded."
END
DATA 6
DATA "Bell","Jane","762-6452"
DATA "Brown","Jim","886-7337"
DATA "Cheslam","Elizabeth","624-2843"
```

```
DATA "Hoden","Paul","923-4475"
DATA "Jones","Fred","843-0922"
DATA "Kalahan","Tom","787-5681"
```

   Program 5-5 creates a random access file called *RPhonelist* and saves the last name, first name, and phone number in each of its records. The first record is an exception, however, since it contains the number of items in the file. This isn't necessary for creating a random access file, but it will be useful for performing a binary search a bit later.
   The **OPEN** statement in the third line opens the file and specifies the record length as 40 bytes. Note that there are only 8 bytes set aside for the first record (number of file items), but 38 bytes are needed for all other records. A couple of extra bytes were added as a safety precaution in case one of the field sizes is changed later. For example, area codes could be included in the phone numbers. By removing the dash in the 8-byte phone number, 3 bytes would be available for the area code.
   The field of record 1 is defined by the fourth line (**FIELD #1,8 AS** COUNT$) to be 8 bytes long. The field variable COUNT$ is given 8 bytes because it will be a packed string for the double-precision variable COUNT (remember that all double-packed variables require 8 bytes). All variables are double-precision by default. The size of the list is stored in a **DATA** statement later in the program (**DATA 6**) and stored into COUNT via a **READ**.
   The sixth line records the count into the field variable COUNT$ with an **LSET** command and packs the number with **MKD$**. The next line records the count in record 1. The field variables have to be redefined for all subsequent records, done by another **FIELD** command. The last and first names are set to a maximum size of 15 characters each, while the phone number has a maximum of 8 characters. The **FOR-NEXT** loop which follows records the names and phone numbers in the random access file. Each record is initially read from **DATA** statements and placed into the appropriate fields with **LSET**. **PUT** stores the record in the phone list file. The file is closed, and the number of records stored is displayed before the program terminates.
   Again, the data can be changed by manipulating the **DATA** statements. The first should contain the count, while

all following statements should contain names and phone numbers. Unlike Program 5-1, the names are sorted alphabetically in ascending order by last name. This is essential to the program examples which follow.

Reading Random Access Files
Once the file has been created, the next logical step is to come up with a method for reading it.

Program 5-6. Reading Random Files

```
CLS
PRINT "Reading Phonelist from disk..."
OPEN "R",#1,"RPhonelist",40
FIELD #1,8 AS COUNT$
GET #1
COUNT = CVD(COUNT$)
FIELD #1,15 AS LASTN$,15 AS FIRSTN$,8 AS PHONEN$
FOR REC = 1 TO COUNT
   GET #1
   PRINT LASTN$,FIRSTN$,PHONEN$
NEXT REC
CLOSE #1
PRINT "Finished,";COUNT;"names listed."
END
```

The first line of importance in Program 5-6 opens the file with a record length of 40 bytes. If the program is to be able to read the information, the record length must be the same as that with which the file was created. Record 1 contains the value of the number of records in the file. There's only one field in this record and it's defined in the fourth line. The following line **GET**s record 1 as COUNT$, which is then unpacked and stored in COUNT by **CVD**. The remaining records have their fields redefined by another **FIELD** command. They're then read (with another **GET**) and their contents are printed to the screen. The file is then closed.

FIVE

Figure 5-5. *The random access file is listed with each record retrieved by GET.*

```
▤□▤▤▤▤▤▤▤▤▤▤▤▤▤▤▤▤▤ Program 5-6 ▤▤▤▤▤▤▤▤▤
Reading Phonelist from disk...
Bell          Jane          762-6452
Brown         Jim           886-7337
Cheslam       Elizabeth     624-2843
Hoden         Paul          923-4475
Jones         Fred          843-0922
Kalahan       Tom           787-5681
Finished, 6 names listed.
```

Adding Records

The purpose of saving the number of records in the file becomes evident when more names and phone numbers have to be added. Look carefully at Program 5-7 and see if you can find the reasons for recording the record count.

Program 5-7. More Records

```
CLS
PRINT "Adding more names and numbers to RPhonelist..."
OPEN "R",#1,"RPhonelist",40
FIELD #1,8 AS COUNT$
GET #1
COUNT = CVD(COUNT$)
FIELD #1,15 AS LASTN$,15 AS FIRSTN$,8 AS PHONEN$
READ NTA
FOR REC = COUNT+2 TO COUNT+1+NTA
  READ LASTNAME$,FIRSTNAME$,PHONENUMBER$
  LSET LASTN$ = LASTNAME$
  LSET FIRSTN$ = FIRSTNAME$
  LSET PHONEN$ = PHONENUMBER$
  PUT #1,REC
NEXT REC
FIELD #1,8 AS COUNT$
COUNT = COUNT + NTA
LSET COUNT$ = MKD$(COUNT)
PUT #1,1
```
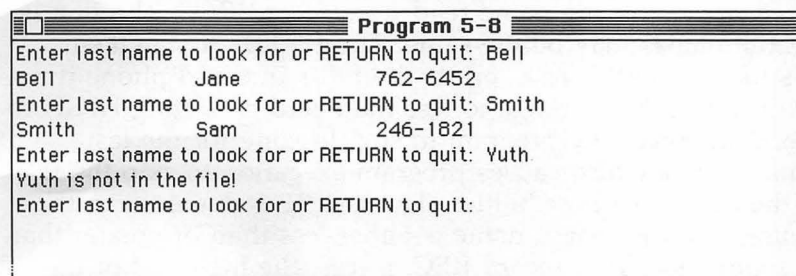
```
CLOSE #1
PRINT "Finished,";NTA;"names added."
END
DATA 3
DATA "Marks","Donna","225-9029"
DATA "Smith","Sam","246-1821"
DATA "Taylor","Cathy","348-7219"
```

Much of what's in Program 5-7 you've already seen. What's new, however, is important. As with the previous program, COUNT$ is read by the first **GET**, then unpacked and converted to the numeric variable COUNT. The **READ** NTA statement in the eighth line puts the number of new records (3) in the variable NTA.

At this point, COUNT contains a value of 6 (which was the number placed in record 1 by Program 5-5). Since the first record of the as-yet-not-updated file holds a count (6) of the file's records, the last name/number record is actually COUNT + 1. Therefore, the first new record to add to the file will be record COUNT + 2.

The first **FOR-NEXT** loop must iterate **READ**, **LSET**, and **PUT** commands within the loop until record number COUNT + 1 + NTA is read. Note that the field variables are prepared with **LSET** before the **PUT** writes to the file. After the new records have been added, the stored count is incorrect. The field variables are redefined again in preparation for updating record 1. COUNT is modified, put into the appropriate field with an **LSET**, and then stored in record 1 by a last **PUT**. The file buffer is no longer needed and therefore is closed.

You'll notice that the names in the **DATA** statements are sorted alphabetically in ascending order by last name. So that the following programs will function properly, the last name in the first **DATA** statement must be alphabetically greater than the last name in the last **DATA** statement in Program 5-5.

LOC and LOF

When adding records to the end of a random access file, the question often comes up as to which record is the last record. One method would be to read each record and count them until the end of the file is reached. An easier way, one where a count doesn't have to be kept, is to use the **LOC** command:

Numeric variable = **LOC**(*filenumber*)

LOC returns the number of the last record written to or read
from the random access file opened with the buffer number
specified by *filenumber.* This command functions differently
with sequential files—it counts the number of items written to
or read from since the file has been open.

　　Reading all the records can take awhile for large files. Not
all records may contain valid information if some have been
discarded and the remaining have been moved to the front of
the file. If it is certain that all records are in use, then the end
of the file can be calculated with **LOF**:

Numeric variable = **LOF**(*filenumber*)

LOF stands for Length Of File and returns the number of
bytes in the file *filenumber.* The number of records in the file
can be calculated by dividing the length of the file in bytes by
the record size of the file. For example, there are ten records
stored in the updated *RPhonelist* file. The record size of the file
is 40 bytes. Thus, it can be assumed that the **LOF** command
would return a value of 400 bytes. When there is a possibility
that the length of the file may not be an indication of the
number of used records within it, the actual number of used
records can be stored in record 1.

Accessing Randomly

Random access files get their name from the fact that *any*
record can be accessed at random. Program 5-8 illustrates the
use of a record 1 count to randomly access names in the file
RPhonelist. (Check to make sure that file is on the disk cur-
rently in the internal drive before you run this program.)

Program 5-8. Random Search

```
CLS
OPEN "R",#1,"RPhonelist",40
FIELD #1,8 AS COUNT$
GET #1
COUNT = CVD(COUNT$)
FIELD #1,15 AS LASTN$,15 AS FIRSTN$,8 AS PHONEN$
NM$=" "
WHILE LEN(NM$)>0
```

```
LINE INPUT "Enter last name to look for or RETURN to quit: ";
NM$
  IF LEN(NM$)=0 THEN FINPROG
  CMPNM$ = SPACE$(15)
  LSET CMPNM$ = NM$
  LOW = 2:HIGH = COUNT + 1
  WHILE HIGH >= LOW
    REC = INT((LOW + HIGH)/2)
    GET #1,REC
    IF CMPNM$ = LASTN$ THEN PRINT LASTN$,FIRSTN$,PHONEN
$:HIGH=0:GOTO STOPLOOK
    IF CMPNM$ < LASTN$ THEN HIGH = REC -1 ELSE LOW = REC +
1

STOPLOOK:
  WEND
  IF HIGH<>0 THEN PRINT NM$;" is not in the file!"

FINPROG:
WEND
CLOSE #1
END
```

This program uses a binary search method of record re-
trieval to find a name and phone number. A binary search is
fast, but requires that the search key be sorted. In this case,
the search key is the last name, which is presorted alphabeti-
cally in ascending order.

Figure 5-6. *A binary search through a random access file
quickly determines whether or not a name is on file.*

```
▣══════════════ Program 5-8 ══════════════
Enter last name to look for or RETURN to quit: Bell
Bell        Jane        762-6452
Enter last name to look for or RETURN to quit: Smith
Smith       Sam         246-1821
Enter last name to look for or RETURN to quit: Yuth
Yuth is not in the file!
Enter last name to look for or RETURN to quit:
```

Again, *RPhonelist* is opened with a record length of 40 bytes. The format of the first record is defined and read. After being unpacked by **CVD**, COUNT contains the number of records in the phone list which include names. The format of the fields within the remaining records is defined by **FIELD**. The name to search for is stored in NM$, initialized to a space so that the test by the statement **WHILE LEN(NM$)>0** will be true and allow execution to continue. The main portion of the program lies between this first **WHILE-WEND** loop.

The program prompts the user to input the last name with **LINE INPUT**. If Return is pressed without entering a name, the length of the input is 0. The **IF** statement makes this check which, when true, causes execution to continue at the final **WEND**, and then **END**. Otherwise, the search string NM$ is padded into the comparison string CMPNM$ with **LSET** after CMPNM$ is given a length of 15 bytes with **SPACE$**. This padding is required in order to be able to compare the name being looked for and the name being read. A match will occur only if the lengths of each are identical.

The binary search routine is next. The variable LOW contains the lowest record number where the name being requested might be found. HIGH contains the highest record number. (Remember that record 1 is simply the total count of the records in the file.) The binary search process repeatedly halves the range of possible record numbers where the name being looked for may be found, until either the name is found or the range is zero. The starting values of LOW and HIGH insure that the range of record numbers to search through is greater than zero. If this is true, the statement REC = INT((LOW + HIGH)/2) calculates the approximate middle of the range and stores that value in the variable REC. Record number REC is read by the next line with a **GET**, after which there can be only three possible relationships between the search name and the name read from record REC. The first is that the names may be the same. The **IF-THEN** statement tests for this and, if true, prints the full name and phone number. It then sets the range to less than zero by setting HIGH to zero. This forces the program to stop looking for the last name. This line also causes program execution to skip the test for the other two possibilities. The next **IF** statement can then assume that the search name is either less than or greater than the name read from record REC. If less, the high end of the

record number search range becomes REC −1, else the low end of the record number search range becomes REC +1. This process halves the record number search range with each iteration.

Even lists of several thousand names can be scanned with a dozen disk accesses or less. When execution reaches the last **IF** statement, either the search name was found or the record search range has become zero or less. Only if the name was found could HIGH be equal to zero. If HIGH does not equal zero, the message that the search name was not in the file is valid. The file is closed after the last **WEND**.

When you run this program, try entering last names which you know are in the file as well as ones you know are not.

CHAPTER

6
Program File
Oriented
Commands

6
Program File Oriented Commands

The commands we'll look at here are not concerned with data files, as were those in the previous chapter, but instead with *program files*. These are simply the programs you create with the Microsoft BASIC application. You've already typed in a number of program files in the preceding chapters.

After typing in some of the examples, you may wish to have a permanent copy of the programs for later use. Choosing *Save* or *Save As* from the *File* menu is one way to save a program file in BASIC. Another way to do this, from within your own BASIC applications, is with the **SAVE** command:

SAVE
SAVE *"program name"*
SAVE *"program name"*,**A**
SAVE *"program name"*,**B**
SAVE *"program name"*,**P**
SAVE *"volume name:program name"*
SAVE *"volume name:program name"*,**A**
SAVE *"volume name:program name"*,**B**
SAVE *"volume name:program name"*,**P**

- The **SAVE** command saves a program file as *program name*, a quoted string.
- When no program name is given, it will be taken from the title of the *Output* window. A dialog box will appear to verify this. (After opening the Microsoft BASIC application from the *Finder*, the *Output* window is initially named *Untitled*.)
- **SAVE** checks to see if a file with the same name already exists—another dialog box confirms the replacement of the existing file if one already exists with the same name.

• *Volume name* specifies the device (necessary if the program
file is to be saved to anything other than the disk in the in-
ternal drive) to which the file is to be saved. Typically, this is
the name of the disk in the external drive. If no volume
name is included, then the device from which the BASIC
originated will be chosen.

Programs can be saved in one of three forms: ASCII, bi-
nary, and protected. These formats are selected from the dia-
log boxes which appear. When a program name is typed, the
default format is binary unless the file was previously saved
with a different format, in which case that format will be cho-
sen. The format can also be specified by appending ,*A* or ,*B* or
,*P* to the **SAVE** command. These represent ASCII, binary, and
protected, respectively. ASCII format is needed when merging
BASIC programs, but is not as efficient a method of storage as
binary. Protected format is a binary format which doesn't
allow the program to be listed or edited when later loaded
into memory.

LOAD
When you wish to retrieve a file already saved, use the BASIC
command **LOAD**:

LOAD
LOAD *"program name"*
LOAD *"program name"*,**R**
LOAD *"volume name:program name"*
LOAD *"volume name:program name"*,**R**

• *Program name* is the file the program was stored in. When no
program name is given, a dialog box appears to prompt you.
• *Volume name* specifies the device from which the file will be
read. If no volume name is included, the device from which
the BASIC originated is selected.
• An autorun option can be specified by appending ,*R* after the
program name. This causes the program to begin execution
after it's loaded. (You can also load programs by selecting
Open from the *File* menu.)

RUN
Once a program is loaded into memory, it will begin execution
after a **RUN** command is provided:

RUN
RUN *"program name"*
RUN *"program name"*,**R**
RUN *"volume name:program name"*
RUN *"volume name:program name"*,**R**

- **RUN** by itself causes immediate execution of the program in memory.
- **RUN** can be used like **LOAD** with an *,R* option by specifying a *program name*.
- Adding an *,R* option preserves any open data files.

Chaining

When a program loads and runs, then runs another program, the variables from the first program are lost and cannot be used by the second. However, **CHAIN** allows another program to begin execution using the variables of the program which invoked it:

CHAIN *"program name"*
CHAIN *"program name"*,*line number*
CHAIN *"program name"*,,**ALL**
CHAIN *"program name"*,*line number*,**ALL**
CHAIN MERGE *"program name"*
CHAIN MERGE *"program name"*,*line number*
CHAIN MERGE *"program name"*,,**ALL**
CHAIN MERGE *"program name"*,,,**DELETE** *firstline–lastline*
CHAIN MERGE *"program name"*,*line number*,**ALL**
CHAIN MERGE *"program name"*,*line number*,,**DELETE**
   firstline–lastline
CHAIN MERGE *"program name"*,,**ALL,DELETE** *firstline–lastline*
CHAIN MERGE *"program name"*,*line number*,**ALL,DELETE**
   firstline–lastline

- The program being chained begins execution at the first line unless an alternative line number is given. (Labels cannot serve as replacements here. If you want to use the *line number* option of **CHAIN**, you need to include line numbers in your BASIC 2.0 program.)
- When the **ALL** option is used, all variables are passed to the called program. If it's not included in the command, the only variables which remain intact are those listed in a **COMMON** statement in the called program.
- Microsoft BASIC 2.0 programs which are chained with the **MERGE** option are placed at the end of the existing program.

- Where merging a file may cause a line number conflict, a range of line numbers (delimited by *firstline* and *lastline*, and including these lines) can be removed from the host program with the **DELETE** option. *Firstline* and *lastline* can also appear as labels.
- Combining the **MERGE** and **DELETE** options when chaining programs simulates program overlays, allowing programs larger than memory to be run on the Macintosh.

MERGE

The **MERGE** command appends ASCII files to the host program in memory:

MERGE *"program name"*
MERGE *"volumename:program name"*

Programs can be appended only to the end of Microsoft BASIC 2.0 programs. **MERGE** is ideal for installing software tools or subroutines into a program.

Installing Software Tools

It's important to know how to install software tools or any other subroutine into a BASIC program. Installing such programs and routines can best be done with BASIC's separate **MERGE** command:

MERGE *filename*

MERGE merges the file specified by *filename* with the program currently in memory. In BASIC 2.0, the file is appended to the end of the current program file.

The file to be merged must be stored in ASCII format, or you'll receive a *Bad file mode* message. The *Output* window is retitled to *filename* if the merge is performed.

Program Merging

As a demonstration of the merging process, type in the following four programs. Save Program 6-1 by entering *SAVE "6-1",A* in the *Command* window. (The *,A* option indicates that the file is to be saved in ASCII format, necessary for merging.)

Program 6-1. GRAPHICS 1

```
GRAPHICS1:
  CLS
```

```
FOR I=10 TO 80 STEP 5
  CIRCLE (50+I*2,130),I
NEXT I
```

Save Program 6-2 by typing *SAVE "6-2",A* in the *Command* window.

Program 6-2. GRAPHICS 2

```
GRAPHICS2:
  CLS
  FOR I=-70 TO 120 STEP 5
    LINE (200+I,INT(80+I/4))-(160+3*I,150-I)
  NEXT I
```

Finally, save Program 6-3 by entering *SAVE "6-3",A* in the *Command* window.

Program 6-3. GRAPHICS 3

```
GRAPHICS3:
  CLS
  FOR I=.1 TO 12.56 STEP .3
    LINE (40+25*I,100+60*COS(I)) - STEP (20,20),,B
  NEXT I
```

These three graphics programs are going to be merged into Program 6-4 as subroutines. The subroutines will be called by their labels. Type in Program 6-4 and enter *SAVE "6-4",A* in the *Command* window.

Program 6-4. Main GRAPHICS Program

```
CLS
GOSUB GRAPHICS1
GOSUB GRAPHICS2
GOSUB GRAPHICS3
END
```

At this point, the main program (Program 6-4) is still in memory, and the three subroutines are saved as ASCII files on disk. The next step is to merge the subroutines into the main

program. Type the following in the *Command* window:

MERGE "6-1"
MERGE "6-2"
MERGE "6-3"

Each **MERGE** command appends the specified file at the end of the program currently in memory. After the three files have been merged with Program 6-4, you need to insert three **RETURN**s. Type in the **RETURN** commands in the *List* window at three points in the program—the first goes just before the label GRAPHICS2, the second just before the label GRAPHICS3, and the final **RETURN** at the end of the program. The program listing should now look like Program 6-5. Before running it, however, type *SAVE "6-5"* in the *Command* window.

Program 6-5. After the Merge

```
CLS
GOSUB GRAPHICS1
GOSUB GRAPHICS2
GOSUB GRAPHICS3
END

GRAPHICS1:
  CLS
  FOR I=10 TO 80 STEP 5
    CIRCLE (50+I*2,130),I
  NEXT I
  RETURN

GRAPHICS2:
  CLS
  FOR I=-70 TO 120 STEP 5
    LINE (200+I,INT(80+I/4))-(160+3*I,150-I)
  NEXT I
  RETURN

GRAPHICS3:
  CLS
  FOR I=.1 TO 12.56 STEP .3
```

```
LINE (40+25*I,100+60*COS(I)) - STEP (20,20),,B
NEXT I
RETURN
```

Merging from Within Programs

Once you set everything up, including the necessary ASCII-formatted files, merging is quite simple. But conducting a merge operation in immediate mode (in other words, by entering commands through the *Command* window) isn't the most efficient use of BASIC's abilities. You can also conduct merges and chainings in your BASIC applications, letting the program itself do the work for you.

The following hands-on example shows you how this is done. First, type in Program 6-6, using *SAVE "6-6",A* (entered in the *Command* window). Don't run this subprogram. If you want to test it yourself, type *CALL SUBCIRCLES(150,120)* in the *Command* window.

Program 6-6. **SUBCIRCLES**

```
SUBS:
SUB SUBCIRCLES(HOR,VERT) STATIC
  CLS
  FOR I=1 TO 30
    CIRCLE (HOR,VERT),I*3
  NEXT I
END SUB
SUBEND:
```

Once Program 6-6 is saved to disk, type *NEW* in the *Command* window, enter Program 6-7, save it to disk, and run it. (If you run it before saving it, a dialog box will interrupt the program, spoiling the display.)

Program 6-7. **SUBLINES**

```
HOR=150
VERT=120
CALL SUBLINES(HOR,VERT)
CHAIN MERGE "6-6",1,ALL,DELETE SUBS-SUBEND
1 CALL SUBCIRCLES(HOR,VERT)
```

```
END
SUBS:
SUB SUBLINES(HOR,VERT) STATIC
  CLS
  FOR I=1 TO 30
    LINE (HOR-I*3,VERT-I*3)-(HOR+I*3,VERT+I*3),,B
  NEXT I
END SUB
SUBEND:
```

Figure 6-1. *The initial program draws nested squares.*
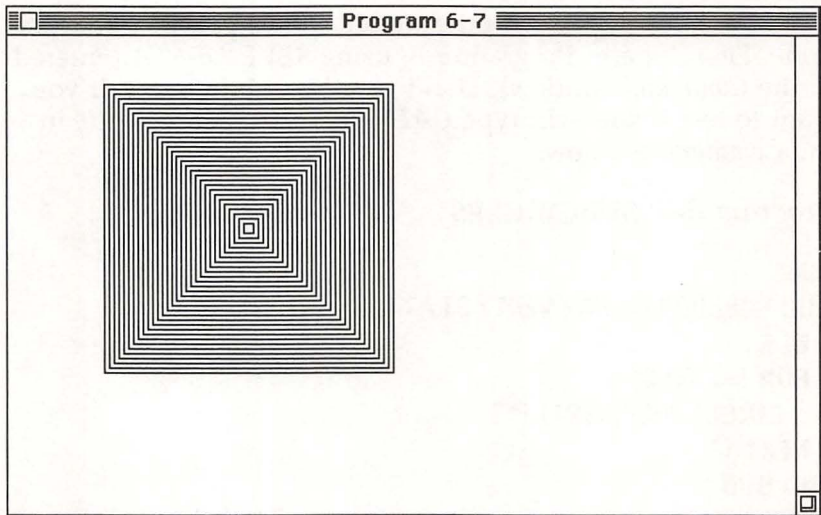
This program initializes the variables HOR and VERT, then calls the subprogram **SUBLINES** that draws some line graphics. Next, a **CHAIN MERGE** command is used to add the ASCII-formatted file 6-6 to the end of this program. Line number 1 (you can mix line numbers and labels in BASIC 2.0) is specified as the place where the program begins. The first line 1 calls the subprogram SUBCIRCLES. The **ALL** option preserves all variables of the host program. **DELETE** erases the SUBLINES subprogram in the chaining process. The labels SUBS and SUBEND are also deleted (remember that you can delete line ranges by specifying labels as well as line numbers). The chained file contains the subprogram SUBCIRCLES,

which is appended to the end of this program.

This operation simulates the overlaying of the SUBCIRCLES subprogram (this subprogram draws some circles). After it's been run, the program ends, and its resulting code is shown in Program 6-8. (You can check this by opening the *List* window.)

Program 6-8. What's Left

```
HOR=150
VERT=120
CALL SUBLINES(HOR,VERT)
CHAIN MERGE "6-6",1,ALL,DELETE SUBS-SUBEND
1 CALL SUBCIRCLES(HOR,VERT)
END
SUBS:
SUB SUBCIRCLES(HOR,VERT) STATIC
  CLS
  FOR I=1 TO 30
    CIRCLE (HOR,VERT),I*3
  NEXT I
END SUB
SUBEND:
```

Figure 6-2. *The CHAINed program draws nested circles.*

101
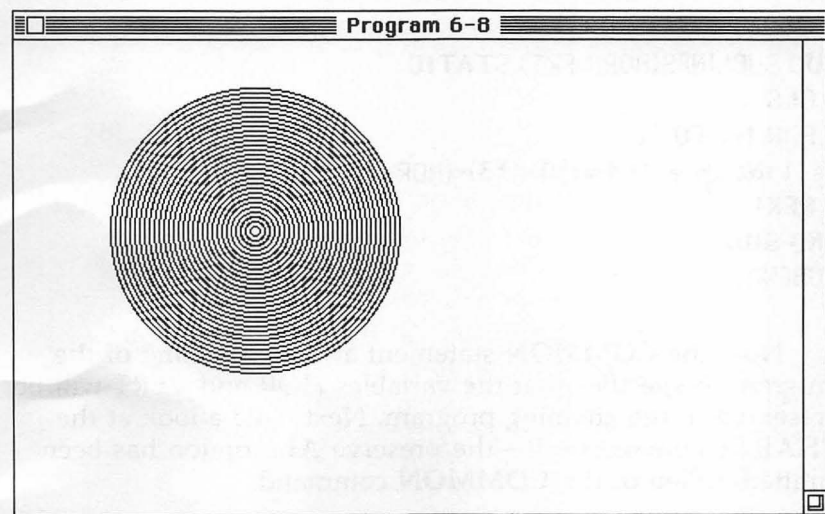
A **COMMON** statement can be used when chaining programs, in effect selectively passing on variables from one program to another in the chaining process. If you wanted to keep all the variables intact, of course, you'd use the **ALL** option instead.

COMMON *variable list*

Variable list indicates which variables' values are to be preserved during the chaining operation. Array variables are marked as such by adding parentheses after them. A command such as **COMMON** NAME$,DAY,AMOUNT,SERVICES( ), for instance, would preserve the string variable NAME$, numeric variables DAY and AMOUNT, and the array SERVICES for the next program to be chained.

Program 6-9 demonstrates overlaying subprograms, much like Program 6-7, except that the **COMMON** statement is used.

Program 6-9. **COMMON**

```
COMMON HOR,VERT
HOR=150
VERT=120
CALL SUBLINES(HOR,VERT)
CHAIN MERGE "6-6",1,DELETE SUBS-SUBEND
1 CALL SUBCIRCLES(HOR,VERT)
END
SUBS:
SUB SUBLINES(HOR,VERT) STATIC
  CLS
  FOR I=1 TO 30
    LINE (HOR-I*3,VERT-I*3)-(HOR+I*3,VERT+I*3),,B
  NEXT I
END SUB
SUBEND:
```

Note the **COMMON** statement at the beginning of the program. It specifies that the variables HOR and VERT will be preserved in the chaining program. Next, take a look at the **CHAIN** command itself—the preserve **ALL** option has been omitted in lieu of the **COMMON** command.

CHAPTER

7

Device I/O and Microsoft BASIC Techniques

7
Device I/O and Microsoft BASIC Techniques

The mouse is used in almost every Macintosh application, from software tools written in Microsoft BASIC to commercial programs written in the language C. Let's look at how the mouse is controlled in BASIC.

The Mouse Button
The command **MOUSE** accesses the status of the mouse button and the position of the mouse cursor on the screen. When checking the mouse button, **MOUSE** has this syntax:

Numeric variable = **MOUSE**(0)

- The value returned by this function is a report on what's happened with the mouse button since the start of program execution or since the last call to the **MOUSE**(0) routine. Possible values returned are −3, −2, −1, 0, 1, 2, and 3.
- The absolute value of these numbers is the number of clicks recorded.
- A negative value means that the mouse button was being pressed when the program executed the **MOUSE**(0) command. Thus, a value of 2 means that a double click has occurred and that the mouse button is not depressed, while a value of −3 indicates that a triple click has occurred and that the mouse button *is* still being pressed. Of course, a value of 0 means that nothing has happened with the button.

Program 7-1 is an example of how to determine the mouse button status.

Program 7-1. *Mouse Button*
```
START:
  CLS
  BUTSTAT = MOUSE(0)
  ON BUTSTAT+4 GOTO TCD,DCD,SCD,ZIP,SCU,DCU,TCU

TCD:
  CALL MOVETO(10,10):PRINT "Triple click, mouse button down
"
  FOR I=0 TO 40 STEP 20:CIRCLE (225,150),50+I:NEXT I:GOTO
PAUSE

DCD:
  CALL MOVETO(10,10):PRINT "Double click, mouse button dow
n"
  FOR I=0 TO 20 STEP 20:CIRCLE (225,150),50+I:NEXT I:GOTO
PAUSE

SCD:
  CALL MOVETO(10,10):PRINT "Single click, mouse button dow
n"
  CIRCLE (225,150),50:GOTO PAUSE

ZIP:
  CALL MOVETO(10,10):PRINT "No mouse activity":CALL MOVE
TO(195,150)
  PRINT "No graphics to display!":GOTO PAUSE

SCU:
  CALL MOVETO(10,10):PRINT "Single click, mouse button up"
  LINE (195,90)-(295,180),,B:GOTO PAUSE

DCU:
  CALL MOVETO(10,10):PRINT "Double click, mouse button up"
  FOR I=20 TO 40 STEP 20:LINE (215-I,110-I)-(275+I,160+I),,B
:NEXT I:GOTO PAUSE

TCU:
```

```
CALL MOVETO(10,10):PRINT "Triple click, mouse button up"
FOR I=20 TO 60 STEP 20:LINE (215-I,110-I)-(275+I,160+I),,B
:NEXT I
```

```
PAUSE: FOR I=1 TO 3000:NEXT I
GOTO START
```

The START routine clears the output window, reads the status of the mouse button with **MOUSE**(0), and saves the status in the variable BUTSTAT. The **ON-GOTO** statement determines what course of action to take. Since BUTSTAT will have a value from -3 to 3, BUTSTAT+4 must be used to transform the range into 1 to 7, required by the **ON-GOTO**.

Figure 7-1. *The* Output *window's display after triple clicking and holding down the mouse button.*
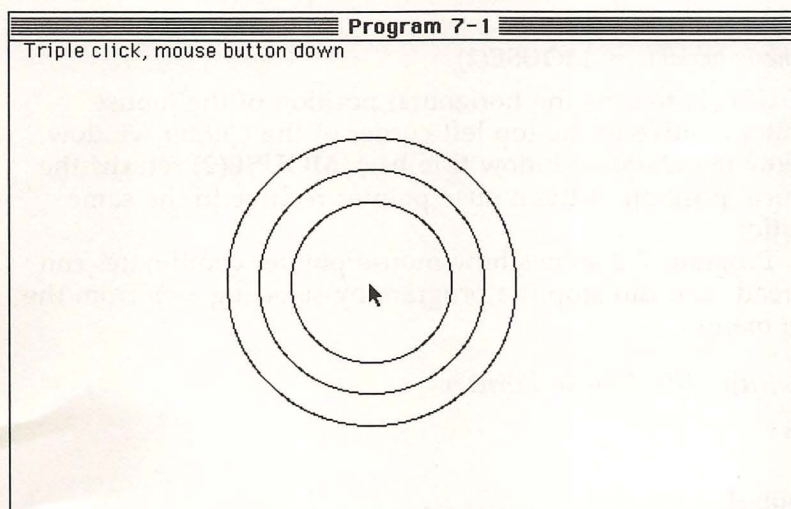
The TCD, DCD, SCD, ZIP, SCU, DCU, and TCU routines are then called at the appropriate value of **MOUSE**(0). TCD, DCD, and SCD represent triple, double, and single clicks, respectively, when the button is pressed. A message to that effect is printed at the top of the screen, and the correct number of circles is drawn.

If the button is released, SCU, DCU, or TCU is called (single, double, or triple clicks, respectively). Squares are drawn

instead of circles.

    When no mouse button activity is read—(**MOUSE**(0) is equal to 0—ZIP is called. PAUSE is referenced after each graphic is drawn.

    By experimenting with this program, you can determine relationships between timing of the mouse button presses and the events which typically follow each other. For example, events with the button still pressed indicate that either a drag operation is in progress or that the program is being run by a pensive mouse handler.

Mouse Pointer Location

The position of the mouse pointer is just as important and is also determined with the **MOUSE** command:

Horizontal position
Numeric variable = **MOUSE**(1)

Vertical position
Numeric variable = **MOUSE**(2)

MOUSE(1) returns the horizontal position of the mouse pointer relative to the top left corner of the *Output* window (below the *Output* window title bar). **MOUSE**(2) returns the vertical position of the mouse pointer relative to the same location.

    Program 7-2 shows how mouse pointer coordinates can be read. You can stop the program by selecting *Stop* from the *Run* menu.

Program 7-2. Mouse Pointer

```
CLS

READBUT:
BUTSEL = MOUSE(0)
HPOS = MOUSE(1)
VPOS = MOUSE(2)
LINE (201,131)-(299,169),30,BF
LINE (200,130)-(300,170),,B
CALL MOVETO(210,155):PRINT HPOS;VPOS;
GOTO READBUT
```

    This displays the current mouse pointer position in a box

on the screen. **MOUSE(0)** is required to set values prior to **MOUSE(1)** and **MOUSE(2)**. These values are stored in HPOS and VPOS. The display box is cleared by the first **LINE** statement and redrawn by the second. The coordinates are printed in the center of the box by the **CALL** and **PRINT** commands. **GOTO** READBUT sends the program in an endless loop.

As you move the mouse cursor around the screen, you'll soon notice that a negative vertical value is displayed if you move the pointer above the *Output* window screen. However, the lowest horizontal value is 0. Keep in mind that the values you see on the screen are not screen coordinates, but are simply values relative to the *Output* window coordinates (0,0).

Figure 7-2. *The mouse position is determined by MOUSE(1) and MOUSE(2) and is based on its position at the time MOUSE(0) is called.*
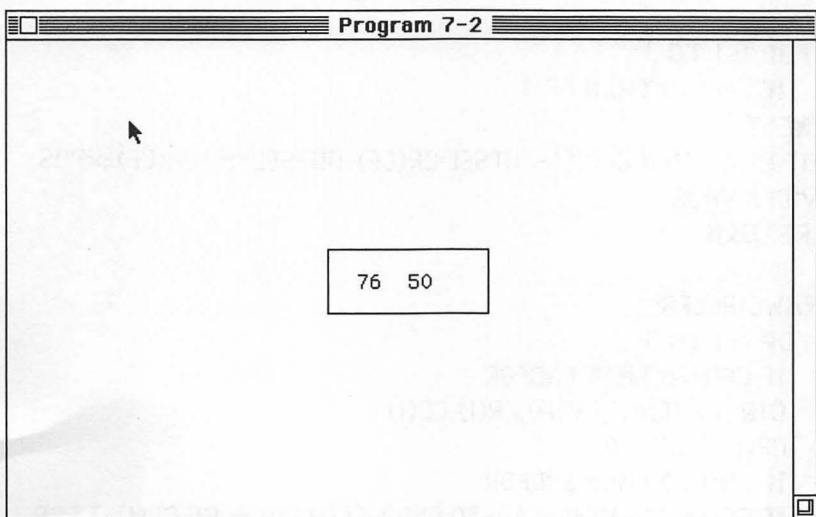
Two Together

Program 7-3 combines these commands to create a graphics demo. Click the mouse anywhere in the *Output* window. Try single, double, and triple clicks—but be quick. Choose *Stop* from the *Run* menu to halt the program.

Program 7-3. Shifting Circles

```
CLS
GOSUB INITVARS

GETBUT:
  BUTSEL = MOUSE(0)
  HPOS = MOUSE(1)
  VPOS = MOUSE(2)
  IF (VPOS>0) AND (VPOS<298) AND (HPOS>0) AND (HPOS<490)
AND (BUTSEL>0) THEN GOSUB CHKARRAYS
  GOSUB DRAWCIRCLES
  GOTO GETBUT

CHKARRAYS:
  CF=0
  FOR I=1 TO 3
    IF CF(I)=0 THEN CF=I
  NEXT I
  IF CF>0 THEN CF(CF)=BUTSEL:CR(CF)=BUTSEL*50:CH(CF)=HPOS:
CV(CF)=VPOS
  RETURN

DRAWCIRCLES:
  FOR I=1 TO 3
    IF CF(I)=0 THEN ENDFOR
    CIRCLE (CH(I),CV(I)),CR(I),CC(I)
    CR(I)=CR(I)-10
    IF CR(I)>0 THEN ENDFOR
    IF CC(I)=33 THEN CC(I)=30:CR(I)=CF(I)*50 ELSE CC(I)=33:CF
(I)=0

ENDFOR:
  NEXT I
  RETURN

INITVARS:
  DEFINT A-Z
```

```
OPTION BASE 1
DIM CF(3),CR(3),CH(3),CV(3),CC(3)
FOR I=1 TO 3:CF(I)=0:CR(I)=0:CH(I)=0:CV(I)=0:CC(I)=33:NEXT I
RETURN
```

This program draws concentric circles, centered at the current mouse pointer location, anywhere in the *Output* window. The circle sizes are dependent on the number of button clicks, with a triple click generating the biggest circle. The circles disappear after a bit—up to three can be drawn at any one time.

Figure 7-3. *Ripples are drawn in a diameter proportional to the number of mouse clicks, which in turn are read by MOUSE(0). The mouse position is read with MOUSE(1) and MOUSE(2).*
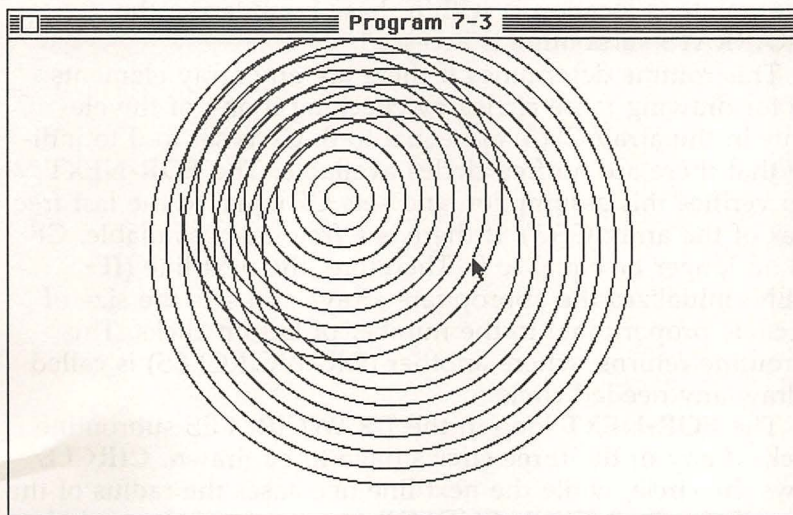
After the screen is cleared, the INITVARS subroutine is called, which initializes the variables. All variables are declared as integers (**DEFINT** A–Z) before the array base index is set to 1 instead of the default 0. The variable arrays CF(3), CR(3), CH(3), CV(3), and CC(3) are defined and have initial values stored in them. CF(3) is called the circle flag array. It has three elements which represent the status of the three possible circles. Array element CF(1) represents the status of circle 1; the other two elements represent the status of the other

circles. Numbers represent the status of a circle. Zero means that it does not exist; otherwise, the value is equal to the circle's maximum size. The array CR(3) represents the current radius of each circle. The centers of each circle are recorded with its horizontal position in CH(3) and its vertical position in CV(3). CC(3) contains the color with which each circle is currently being drawn (33, or black).

After the variables are initialized, the routine GETBUT retrieves the mouse information with **MOUSE(0)**, **MOUSE(1)**, and **MOUSE(2)**. The **IF-THEN** statement causes the program to ignore all mouse activity unless there's been a mouse click in the *Output* window (BUTSEL must be greater than 0). Something has happened within the *Output* window when the horizontal position of the mouse is between 0 and 490 and when the vertical position of the mouse is between 0 and 298. If the pointer's location is within these boundaries, the CHKARRAYS subroutine is accessed.

This routine determines if there are any array elements free for drawing more circles by checking if any of the elements in the array CF(3) are equal to 0. CF is set to 0 to indicate that there are no free circles available. The **FOR-NEXT** loop verifies this assumption and sets CF equal to the last free index of the array CF(3). If there is a free index available, CF will no longer be equal to 0. Therefore, the next line (**IF-THEN**) initializes the appropriate arrays and sets the size of the circle proportional to the number of button clicks. This subroutine returns, where another (DRAWCIRCLES) is called to draw any needed circle.

The **FOR-NEXT** loop in the DRAWCIRCLES subroutine checks if any of the three circles have to be drawn. **CIRCLE** draws the circle, while the next line decreases the radius of the circle. **IF** CR(I)>0 **THEN** ENDFOR tests to see if the circle has a radius of 0. If the radius is greater than 0, none of the circles' parameters has to be changed. The last **IF-THEN** in this routine changes the circle's color to white if it's currently black so that it can be erased. Otherwise, the corresponding element of the array CF(3) is reset to 0.

Dragging Mice

Mouse drag operations, where the mouse button is kept pressed, are also determined with the help of the **MOUSE** command:

Starting horizontal location of drag
Numeric variable = **MOUSE**(3)

Starting vertical location of drag
Numeric variable = **MOUSE**(4)

MOUSE(3) and **MOUSE**(4) return the horizontal and vertical positions of the mouse pointer at the *start* of the drag operation. Program 7-4 shows you how this is used.

Program 7-4. Mouse Drags

```
CLS

GETBUT:
  BUTSTAT=0
  WHILE BUTSTAT=0
    BUTSTAT=MOUSE(0)
  WEND
  WHILE MOUSE(0)<0
    LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),33,B
    LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),30,B
  WEND
  ON ABS(BUTSTAT) GOTO ONE,TWO,THREE

ONE:
  LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),33:GOTO E
NDLOOP

TWO:
  LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),33,B:GOTO
ENDLOOP

THREE:
  LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),33,BF

ENDLOOP:
  GOTO GETBUT
```

Click the mouse button once, drag the mouse pointer, and then let go of the mouse button—this program draws a line

from the point where the mouse button was first clicked to where it was released. The line's endpoint coordinates show the starting and ending points of the drag. If the mouse is double clicked before dragging, a box is drawn with two of its corners defined at these points. A triple click draws a filled box when the mouse button is released. The program can be stopped only by choosing *Stop* from the *Run* menu.

Figure 7-4. *To create this display, position the mouse at the top left corner of the box. Triple click and then drag the mouse to the lower right corner to make the background. Next, position the mouse at the top left corner of the background box. Double click and drag the mouse to the lower right corner of the background before releasing the mouse button.*
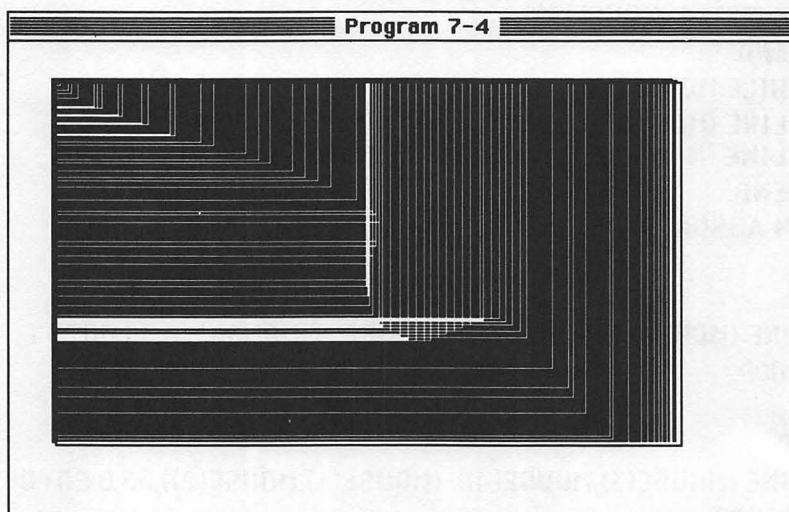
GETBUT defines the point in the program where the press of the button is monitored. First initialized at 0, the variable BUTSTAT is checked by the first **WHILE-WEND** loop until the button is pressed. As soon as the button is pressed, BUTSTAT is no longer 0, and so the program execution falls through to the next **WHILE-WEND** loop. This draws the outline between the starting and ending points (the latter is where the button is released). When the button is depressed, **MOUSE(0)** returns a negative value. The **ON-GOTO** statement determines how

many clicks were made—**ABS(BUTSTAT)**—then the appropriate subroutine (ONE, TWO, or THREE) is called to draw the line, box, and filled box, respectively. Note that **MOUSE(3)** and **MOUSE(4)** retrieve the starting coordinates of the drag.

Mouse Place

When a drag operation ends with the release of the mouse button, the exact position of the mouse pointer can be found with the **MOUSE** command when you use these formats:

Ending horizontal location of drag
Numeric variable = **MOUSE(5)**

Ending vertical location of drag
Numeric variable = **MOUSE(6)**

MOUSE(5) and **MOUSE(6)** return the horizontal and vertical positions of the mouse at the *end* of the drag operation. Program 7-5, though much like the previous example, uses these two commands to more accurately reflect the endpoints of the drag operations. **MOUSE(1)** and **MOUSE(2)**, used in Program 7-4 to serve the same purpose, have been replaced here by **MOUSE(5)** and **MOUSE(6)**.

Program 7-5. Mouse Tails

```
  CLS
GETBUT:
  BUTSTAT=0
  WHILE BUTSTAT=0
    BUTSTAT=MOUSE(0)
  WEND
  WHILE MOUSE(0)<0
    LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),33,B
    LINE (MOUSE(3),MOUSE(4))-(MOUSE(1),MOUSE(2)),30,B
  WEND
  ON ABS(BUTSTAT) GOTO ONE,TWO,THREE
ONE:
  LINE (MOUSE(3),MOUSE(4))-(MOUSE(5),MOUSE(6)),33.GOTO E
NDLOOP
TWO:
  LINE (MOUSE(3),MOUSE(4))-(MOUSE(5),MOUSE(6)),33,B:GOTO
  ENDLOOP
```

THREE:
 LINE (MOUSE(3),MOUSE(4))-(MOUSE(5),MOUSE(6)),33,BF
ENDLOOP:
 GOTO GETBUT

Figure 7-5. *MOUSE(5) and MOUSE(6) more accurately reflect the ending position of the mouse pointer when there is a time gap between calls of MOUSE(0).*
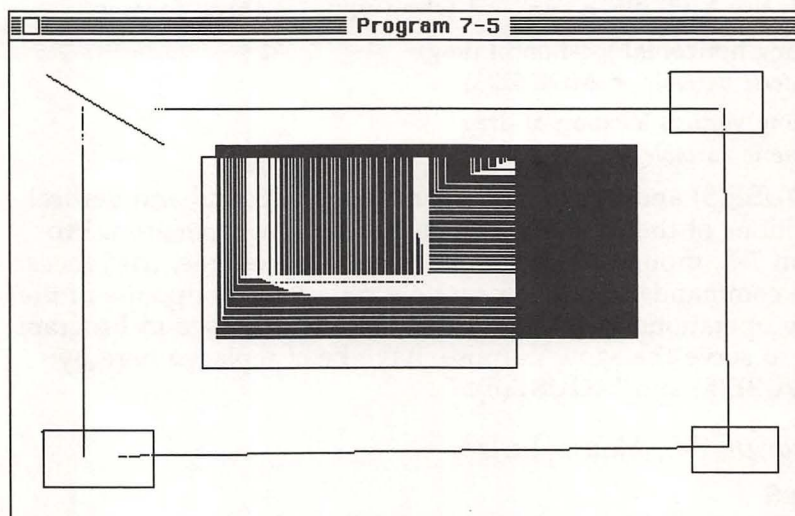
Program 7-5

Printer Commands

A printout can be extremely useful, both for debugging programs and for keeping permanent copies of information generated by applications. The command for listing programs to the printer is called **LLIST**:

LLIST
LLIST *first line number*
LLIST – *last line number*
LLIST *first line number – last line number*

This lists the program to the printer, from *first line number* to *last line number*, inclusive. If the first line number is omitted, the program's first line is chosen as the start. The last line in the program is chosen if *last line number* is not specified.

The **LLIST** command can be emulated by using the **LIST** command with the follow syntax:

116

LIST,"LPT1:"
LIST *first line number,***"LPT1:"**
LIST – *last line number,***"LPT1:"**
LIST *first line number* – *last line number,***"LPT1:"**

The parameters of these commands are the same as those in **LLIST**.

Programs can display output to the printer instead of the screen simply by using an **LPRINT** command instead of a **PRINT** command. All parameters and options remain the same.

Devices

There are five devices with which you can specify general input/output on the Macintosh. The first device is **SCRN:**, the name for the *Output* window. Since 2.0 can have more than one *Output* window, **SCRN:** specifies the current one. If you load a program into memory, for instance, and then type **LIST,"SCRN:"**, the listing will appear in the *Output* window.

The device called **KYBD:** accepts input from the keyboard.

The Clipboard file is called **CLIP:**, and can be opened as a file to read information from and save information to the Macintosh's Clipboard. Try Program 7-6. Before running it, make sure that there's something in the Clipboard file by selecting a segment of a program from the *List* window and choosing *Copy* from the *Edit* menu.

Program 7-6. CLIP:

```
CLS
OPEN "CLIP:" FOR INPUT AS #1
BYTES = LOC(1)
PRINT BYTES;"bytes in the clipboard"
CLIP$ = INPUT$(BYTES,#1)
PRINT"The clipboard contains"
PRINT CLIP$
CLOSE #1
```

The second line opens the Clipboard for input as file buffer number 1. **LOC** determines the number of bytes in the file and stores the result in the variable BYTES. **INPUT$** reads

BYTES from the Clipboard and stores them in the string variable CLIP$. This program can serve as a model for inputting information from other applications into Microsoft BASIC through the Clipboard.

Figure 7-6. *The Clipboard is read to accept data from other applications.*

```
≣□≣                              Program 7-6 ≣
  156 bytes in the clipboard
The clipboard contains
CLS
OPEN "clip:" FOR INPUT AS #1
BYTES = LOC(1)
PRINT BYTES;"bytes in the clipboard"
CLIP$ = INPUT$(BYTES,#1)
PRINT"The clipboard contains ";CLIP$
CLOSE #1
```

There are two options available when you use the device **CLIP:**. The default option is **CLIP:TEXT**, which is the same as using only **CLIP:**. The **TEXT** option specifies that the data from the Clipboard is text. The other option, **PICTURE**, is used in the format **CLIP:PICTURE** when opening a file. This option specifies that the data being transferred to or from the Clipboard is graphics.

The device **COM1:** is the Macintosh's serial port. It's typically opened for input from and output to other computers and devices connected to the serial port via a serial cable or modem. Microsoft BASIC 2.0 allows the port to be configured when a file is opened which references **COM1:**. These options are

COM1:
COM1: *baud rate,parity,data bits,stop bits*

- *Baud rate* can be any value from the set of 110, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, 19200, and 57600. If none is defined, a baud rate of 300 is used.
- *Parity* is set to *even* with an *E*, to *odd* with an *O*, or to *no parity* with an *N*. Even is selected if no parity option is used.
- The number of data bits can be set to 5, 6, 7, or 8 bits with 7 data bits the default.
- Stop bits are either 1 or 2. BASIC 2.0 uses a default number

of stop bits dependent on the baud rate—1 stop bit for 300 baud, and 2 for all other rates.

The other serial port is **LPT1:**, which can be used in these forms:

LPT1:
LPT1:DIRECT
LPT1:PROMPT

- **LPT1:** is normally called the printer port. When a printer is connected to this port, output can be sent to it. For printing graphics or enhanced text, this would be used.
- The **DIRECT** option is used when sending ASCII or unenhanced text to the printer. This is similar to the Macintosh's draft printing mode.
- If the mode must be decided at the time of printing, you can use the **PROMPT** option. This calls a short series of dialog boxes which ask for the paper size and the print quality. (Graphics cannot be printed when draft-quality print mode is selected.)

Memory Management

A 128K Macintosh has only a small fraction of its memory available for use by a BASIC program. After loading the BASIC interpreter, there are about 21,000 bytes available for a program and its variables. There are approximately 13,000 bytes available in the Macintosh *heap*, the memory work area used for the Clipboard, window management, and desk accessories.

Memory areas can be measured with the **FRE** command:

Numeric variable = **FRE**(*number or string*)

- **FRE** returns the number of bytes unused by a program and its variables when a number greater than or equal to zero is passed as a parameter. Using a string as a parameter has the same effect.
- A negative number as a parameter causes **FRE** to return the amount of free bytes left in the Macintosh heap. Typing **PRINT FRE(0),FRE(−1)** in the *Command* window after loading BASIC or after typing **NEW** will display the available bytes in both BASIC program memory and the heap.

A third area in the Macintosh memory is called the *stack*, used by BASIC to manage **FOR-NEXT** loops, **WHILE-WEND**

loops, and subroutines. These areas can be resized to allow more memory for programs and variables with a **CLEAR** command:

CLEAR
CLEAR,*size of program area*
CLEAR,,*size of stack*
CLEAR,*size of program area,size of stack*

CLEAR sets all numeric variables to zero and all strings to null. All files are closed and the program area, stack, and heap are resized as specified. When not specified, the size of the memory areas are left unchanged. The size of the program area and the stack must be at least 1024 bytes each. The heap is indirectly set by a **CLEAR** command. Its size is whatever is left over after the program area and stack size are set.

Memory can be conserved when dimensioning arrays with a **DIM** command. When 20 lists of 20 numbers are to be stored in an array, the command **DIM** NUMBS(20,20) is a possible method of allocating the array. However, these are double-precision numbers by default and occupy 8 bytes of memory each. If single-precision numbers are sufficient, the statement should be **DIM** NUMBS!(20,20) which requires only 4 bytes per number and represents a 50 percent savings of memory (1764 bytes would be saved). If the numbers to be placed in the lists are small and fall within the range of −32768 to 32767, an integer array should be declared. A statement like **DIM** NUMBS%(20,20) would suffice. These require only 2 bytes per element and represent a savings of 75 percent over the double-precision array, or about 2646 bytes. Variable defaults can be changed from double precision to single precision with a command called **DEFSNG**. Integers can be specified by **DEFINT**. There are four such commands:

Integers
DEFINT *starting letter in variable name 1*
DEFINT *starting letter in variable name 1–starting letter in variable name 2*

Single-precision numbers
DEFSNG *starting letter in variable name 1*
DEFSNG *starting letter in variable name 1–starting letter in variable name 2*

Double-precision numbers
DEFDBL *starting letter in variable name 1*

DEFDBL *starting letter in variable name 1–starting letter in variable name 2*

Strings
DEFSTR *starting letter in variable name 1*
DEFSTR *starting letter in variable name 1–starting letter in variable name 2*

* *Starting letter in variable name 1* indicates that all variables whose names begin with this character are of the specified type (integer, single-precision, double-precision, or string). When *starting letter in variable name 2* is given, all variables whose names begin with the letters defined in the range are of the specified type. For instance, the command **DEFINT** D declares that all variables starting with the letter *D* are integers. **DEFSNG** E–K declares that all variables starting with the letters *E* through *K* are single-precision numbers.

Another small waste of memory is due to the fact that array indexes begin at zero by default. For example, the array NUMBS(20,20) has enough variable space for 21 lists of 21 numbers. If you needed only 20, this array would use 328 extra bytes. It should have been dimensioned with the statement **DIM** NUMBS(19,19). However, counting from zero can lead to confusion. Fortunately, there's a command to reset the base index of an array:

OPTION BASE *number*

Number specifies the minimum value for an index to an array. The NUMBS(20,20) array is adequate if preceded by a statement like **OPTION BASE** 1.

Measuring and Setting Time

Time measurement and time stamping are two functions often required by application programs. The date can be determined in BASIC with

String variable = **DATE$**

DATE$ returns the date in month-day-year format. Two characters each represent the month and day, while four characters represent the year. All are separated with a dash. The date can be set by

DATE$ = *"mm-dd-yy"*
DATE$ = *"mm-dd-yyyy"*

DATE$ = *"mm/dd/yy"*
DATE$ = *"mm/dd/yyyy"*

• *mm* represents the month, *dd* the day, *yy* the last two digits of the year, and *yyyy* all four digits of the year.

The time can be determined with

String variable = **TIME$**

TIME$ returns the time in an hour:minute:second format (hh:mm:ss). Two characters each represent the hour, minutes, and seconds of a 24-hour clock, each separated by a colon. The time can be set by

TIME$ = *"hh"*
TIME$ = *"hh:mm"*
TIME$ = *"hh:mm:ss"*

If either seconds or minutes are unspecified, zero is assumed. Program 7-7 shows how you can both read and set the date and time.

Program 7-7. DATE$ and TIME$

```
CLS
PRINT "The date is: ";DATE$
PRINT "The time is: ";TIME$
LINE INPUT "Enter a new date if the date is wrong, else press 'R
ETURN':";D$
IF LEN(D$)>0 THEN DATE$=D$:PRINT "The new date is: ";DATE$
LINE INPUT "Enter a new time if the time is wrong, else press '
RETURN':";T$
IF LEN(T$)>0 THEN TIME$=T$:PRINT "The time is: ";TIME$
END
```

Not only does Program 7-7 display the current date and time, but it allows you to change them.

S E V E N

Figure 7-7. *The date and time are read and set with DATE$*
and TIME$.

```
▤☐▤▤▤▤▤▤▤▤▤▤▤▤▤▤▤ Program 7-7 ▤▤▤▤▤▤▤▤
The date is: 05-09-1985
The time is: 18:24:41
Enter a new date if the date is wrong, else press 'RETURN':
Enter a new time if the time is wrong, else press 'RETURN': 18:27:00
The time is: 18:27:00
```
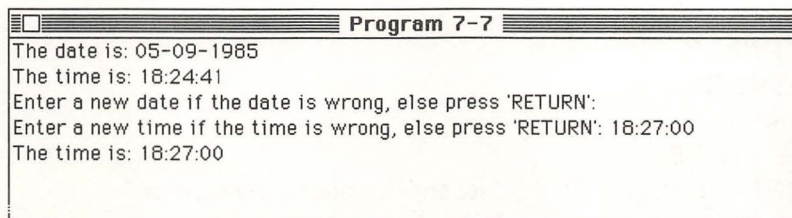
The second and third lines display the date and time. If
you respond to the message prompt displayed by **LINE IN-
PUT**, the input stored in D$ will have a length greater than
zero, and the date is set. The same procedure is repeated for
the time with T$. If either is changed, the new date or time
appears on the screen.

When you're trying to determine elapsed time, as in a
game, the timing command **TIMER** is extremely useful:

Numeric variable = **TIMER**

TIMER returns the number of seconds elapsed since midnight.
Try Program 7-8 for a little fast-paced relaxation.

Program 7-8. Timing

```
CLS
NP=0
ET=0
PRINT "This program will time you as to how fast you can quess
  a number"
PRINT "between 1 and 100."

PLAY:
NUM = INT(RND(1)*100)+1
PRINT "Go!"
TS=TIMER

GETGUESS:
LINE INPUT "Enter quess:";G$
NG=INT(VAL(G$))
IF NG=NUM THEN GETTIME
```

123

IF NG<NUM THEN PRINT "Too low." **ELSE PRINT** "Too high."
GOTO GETGUESS

GETTIME:
TF=TIMER
NP=NP+1
ET=ET+TF-TS
PRINT"It took you";TF-TS;"seconds to guess the number."
LINE INPUT "Do you wish to try again?:";YN$
IF LEFT$(YN$,1)="y" OR LEFT$(YN$,1)="Y" THEN PLAY
PRINT "You took about";ET/NP;"seconds per number to guess it."

Figure 7-8. *TIMER is used to measure the elapsed time in this game.*

```
≣□≣═══════════════════ Program 7-8 ════════════════════
This program will time you as to how fast you can guess a number
between 1 and 100.
Go!
Enter guess: 64
Too high.
Enter guess: 13
It took you  6 seconds to guess the number.
Do you wish to try again?: n
You took about  6 seconds per number to guess it.
```
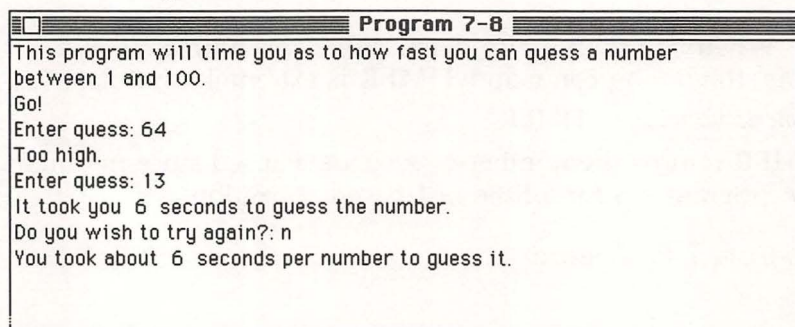
This quick-reflex game tests your ability to guess a number between 1 and 100.

Variables NP and ET represent the number of games played and the elapsed time to play those games, respectively. Next, instructions are displayed.

The PLAY subroutine randomly chooses the number to guess, prints the *Go!* message, and reads the timer and saves it into TS. This is the starting time.

The next routine, GETGUESS, inputs your guess into G$ and converts that input into a number stored in NG. If something alphabetic is entered, NG contains zero and the guess will be too low. When NG equals NUM, the GETTIME routine is called. Appropriate messages (*Too low* or *Too high*) are printed, depending on whether your number was less than or

greater than NUM. The subroutine loops endlessly until your choice matches the computer's.

GETTIME records the time of finish as TF, increments the number of games played (NP=NP+1), and then adds to the elapsed time (ET). Game time is equal to the finish time minus the starting time. After presenting you with a chance for another game, the program displays the average time it took you to guess a number. All games played up to this point are figured into this value (ET/NP).

Anything less than ten seconds is quite good.

Random Numbers

Random numbers are useful, sometimes vital, for simulations and games. As you saw in the last example, random numbers can be generated with the **RND** command, which has this form:

Numeric variable = **RND**(*number*)

RND returns a double-precision number from a sequence of random numbers between 0 and 1, when *number* is greater than 0. If *number* is equal to 0, **RND** offers the same number it returned previously. If *number* is less than 0, the random number sequence is restarted.

When a random number within a range is needed, multiply the result of a **RND** function by the range and then add the lowest number. If you want a number between 1 and 100, inclusive, for instance, you'd use (**RND**(1)*100)+1. The values will range from near 1 to almost 100. By changing the formula to **INT**(**RND**(1)*100)+1, the numbers near 1 become 1, and the numbers near 100 become 100. Changing the formula to **INT**(**RND**(1)*100) makes the lowest number possible 0, and the highest possible 99.

Type **CLS:FOR** I=1 **TO** 100:**PRINT RND**(1):**NEXT** I. Assume that three of the numbers displayed are

0.009027123451233 (a number close to zero),
0.57365065813065, and
0.99010872840882 (a number close to one).

Type **CLS:FOR** I=1 **TO** 100:**PRINT RND**(1)*100:**NEXT** I. Now the numbers have become 0.9027123451233, 57.365065813065, and 99.010872840882, respectively. Enter **CLS:FOR** I=1 **TO** 100:**PRINT INT(RND**(1)*100):**NEXT** I. These numbers have now become 0, 57, and 99. Finally, type **CLS:FOR** I=1 **TO** 100:**PRINT INT(RND**(1)*100)+1:**NEXT** I.

The numbers are now 1, 58, and 100.

Each time the list of 100 numbers was displayed, you probably noticed that **RND**(1) generated the same sequence. This sequence can be changed with

RANDOMIZE
RANDOMIZE *seed*
RANDOMIZE TIMER

The **RANDOMIZE** command, when used by itself, asks for a seed with which to generate the new sequence of numbers. At this point, you should enter a number from −32768 to 32767. The seed can be included in the **RANDOMIZE** command if you prefer. **RANDOMIZE** can also use the **TIMER** function to get a seed. Program 7-8 can be modified by adding this line as the second line of the program:

RANDOMIZE TIMER

This will prevent anyone from being able to memorize the numbers this program selects. Each time it's run, a different list of numbers is generated.

Error Processing

Even the best programs can produce errors. Some may even be intentional—one way to check for the existence of a file, for example, is to try to open it. If a *File not found* error appears, the answer is obvious.

Unfortunately, errors terminate a BASIC program unless an **ON ERROR GOTO** command is used:

ON ERROR GOTO *line number*

Line number specifies the line to go to if an error occurs. In Microsoft BASIC 2.0, you can use a label instead of a line number. Once an **ON ERROR GOTO** command has been executed, all errors will cause a jump to the specified line number. Error trapping can be turned off with the statement **ON ERROR GOTO** 0.

Once an error has been detected, **ERR** is used to determine *which* error occurred:

Numeric variable = **ERR**

ERR returns a number called an *error code*. Your Microsoft BASIC reference manual has a list of these codes and their meanings.

Another useful command for diagnosing the nature of the error is **ERL**:

Numeric variable = **ERL**

ERL returns the line number in which the error occurred. An error may have occurred in a program on a line which has no number; in that case, the last numbered line executed is returned by the **ERL** function. When there are no line numbers at all, 0 is returned.

After the appropriate action has been taken, to continue program execution, use:

RESUME
RESUME 0
RESUME NEXT
RESUME *line number*

RESUME or **RESUME 0** continues the program at the statement where the error occurred. **RESUME NEXT** continues execution at the statement after the error. Otherwise, the program resumes at the line specified by *line number*. Type in and run Program 7-9. Be ready for an error.

Program 7-9. Resuming After Error

```
CLS
ON ERROR GOTO SHOWERR
10 A$=12
PRINT "All is fine"
END

SHOWERR:
PRINT "There is an error number";ERR;"on line";ERL
RESUME MESSAGE

MESSAGE:
PRINT "Fix it and try again!"
END
```

Figure 7-9. *ERR and ERL give the error number and the line
number of the error.*

```
≣□≣══════════════ Program 7-10 ═══════════════
There is an error number  64 on line 0
Fix it and try again!
```
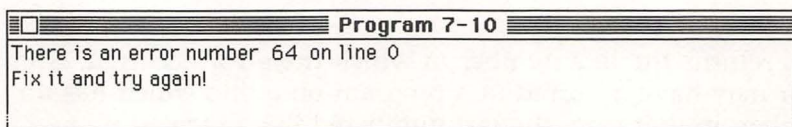
Whenever there's an error (as there will be the first time
Program 7-9 is run), the **ON ERROR GOTO** SHOWERR
sends the program to the SHOWERR routine. Line 10 (remem-
ber that BASIC 2.0 can mix line numbers and labels at will)
contains an error which must be fixed for the message *All is
fine* to appear.

Change line 10 to the following and run the program
again.

10 A$="12"

One application for this kind of error trapping is with a
subroutine that determines if a file exists. The subroutine can
be something like Program 7-10.

Program 7-10. Error Subroutine

```
ON ERROR GOTO CHKERROR
OPEN "I",#1,FILENAME$
FOR I=1 TO 10
   INPUT#1,ITEM(I)
NEXT I
CLOSE#1

CHKERROR:
IF ERR=53 THEN PRINT "File not found!":RESUME ENDOPEN ELSE
   PRINT "There is an error number";ERR;"on line";ERL:END

ENDOPEN:
RETURN
```

The first line defines subroutine CHKERROR as the place
to go if there's an error in the program. A file (whose name is
stored in FILENAME$) is opened. If the file is not present, an

error code 53 occurs, and execution continues at the CHKERROR subroutine, where a message will be displayed. Execution then resumes at ENDOPEN, which performs a RE-TURN. Since any error causes execution to continue at CHKERROR, a test for the *File not found* error code (53) must be made. If this is not the error which occurred, the actual error code and line number are shown and the program stops. If no errors occur, the file is read and closed.

This routine could be used to load data from files specified by FILENAME$. The code in the **FOR-NEXT** loop should be changed to call a routine to input the file information.

(Note that since there are no line numbers in this program, any error but error 53 will indicate line 0 as the place where the error appeared.)

CHAPTER

8
Using ROM
Routines

8
Using ROM
Routines

The Macintosh has 64K of ROM routines, some of which can be accessed with Microsoft BASIC. Many of these routines are used by the software utilities discussed in Chapter 11.

The routines accessible from BASIC deal with graphics, mouse cursors, and text display, all executed by using the BASIC **CALL** command:

CALL *routine name*
CALL *routine name (parameter list)*

CALL calls the ROM routine specified. If the routine requires parameters, they're listed as *parameter list*. Parameters are values or data required by the ROM routine. Some parameters are a memory address or a set of values. Typically, these values are stored in an array and the command **VARPTR** is used to access the variables' addresses:

Numeric variable = **VARPTR** *(variable)*

VARPTR returns the address of a variable, in other words, where it's currently found in memory. Since variables are shuffled about, this command should be used just before it's required. Program 8-1 is a short demonstration of how **VARPTR** can be correctly placed in a program.

Program 8-1. VARPTR

```
CLS
DIM X(8)
ADR1=VARPTR(X(0))
PRINT ADR1
ADR2=VARPTR(X(0))
PRINT ADR2
```

Figure 8-1. *The address of the variable X(0) changes as new variables are allocated.*

```
≣□≣═══════════════════ Program 8-1 ═══════════
44838
44848
```
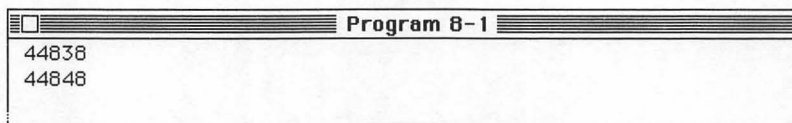
This program prints two different numbers, the addresses of the array X(8). The second line creates the array, while the third records the address of its first element in the variable ADR1. (The location of an array is found at a fixed number of bytes *before* the address of its first element. Since this offset is constant, the change in the address of an array can be determined by looking at the change in the address of its first element. In this case, the first element is X(0).) The fifth line also records the address of the first element of array X(8), but stores the address in the variable ADR2. The values of the variables are displayed on the screen. They differ by ten bytes, indicating that X(8) has moved ten bytes between the fourth and fifth lines. The assignment of a new variable (ADR2) is the cause (see Figure 8-2 for an illustration of the dynamic relocation of variables in memory).

Rectangle Limits

Some of the ROM routines require a rectangle to delimit graphics. This quadrilateral area of the screen is defined by the topmost, leftmost, bottommost, and rightmost lines, in that order. These lines are stored in an integer array with four consecutive elements defining the boundaries of the rectangle. For instance, a rectangle may be bounded by horizontal positions 20 and 120 and by vertical positions 40 and 200. The topmost line is thus 40, the leftmost 20, the bottommost 200, and the rightmost 120. The following program lines will prepare this array.

```
DIM RECT%(3)
RECT%(0)=40:RECT%(1)=20:RECT%(2)=200:RECT%(3)=120
```

The first line defines an integer array with four elements, while the second stores the appropriate values in that array. The array is referenced by **VARPTR(RECT%(0))** since ROM routines need the array's address.

The rectangle information can be located anywhere within an array. The following lines, in fact, would work just as well.

DIM RECT%(9)
RECT%(3)=40:RECT%(4)=20:RECT%(5)=200:RECT%(6)=120

The rectangle would be accessed with **VARPTR**(RECT%(3)).

Figure 8-2. *Memory maps of arrays in motion.*

Memory Map after executing DIM X(8)

Memory Map after executing ADR1=VARPTR(X(0))

Memory Map after executing ADR2=VARPTR(X(0))
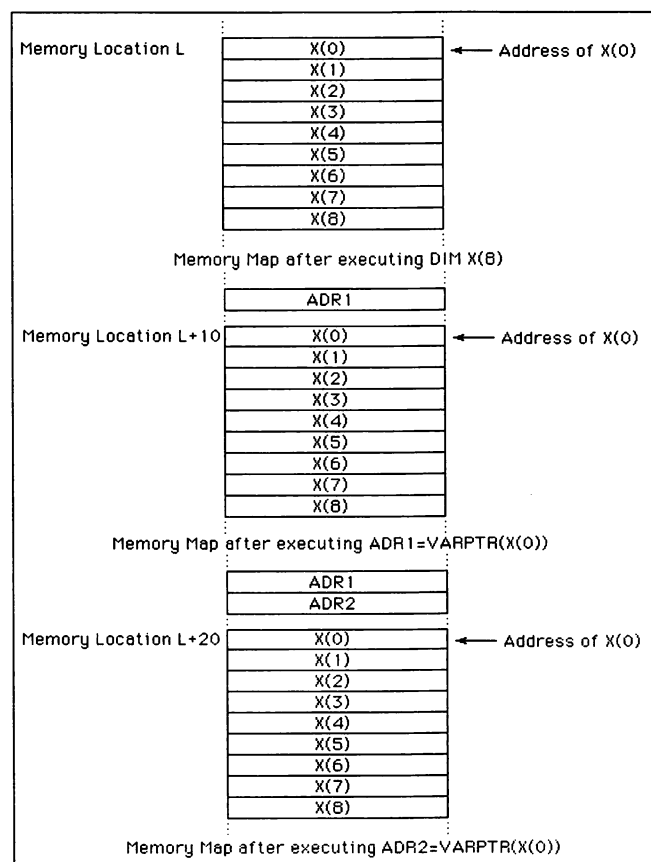
Patterns for ROM Routines

Some ROM routines require a pattern, defined by a set of four integer numbers. Figure 8-3 offers an explanation of such encoding.

Figure 8-3. *Bit image data definition.*
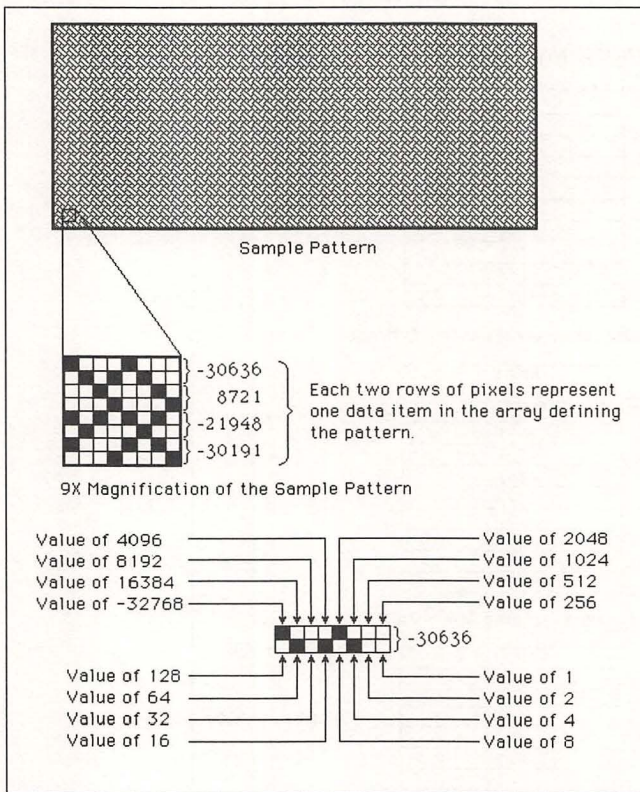
At the top, there's a sample pattern drawn within a rectangle. The high resolution of the Macintosh screen makes the pattern very fine and difficult to resolve, so an 8 × 8 pixel section of the pattern is enlarged nine times. All patterns are defined by an 8 × 8 pixel square. A closer examination of the pattern reveals that the ROM routines simply repeat this 8 × 8 pixel bitmap when drawing patterns. Portions of the bitmap are used when the area with the pattern is a size where its height or width is not a multiple of 8.

The bitmap is divided into four sections of two rows each. Each of these pairs of rows has an equivalent integer value. This way, a pattern can be defined by four numbers—in this case, -30636, 8721, -21948, and -30191. These numbers are derived in the manner shown at the bottom of the figure. Each pixel that's on (black) requires its associated value to be added. The sum of these values gives the final value for that data item. In this case, the value of the first data item representing the pattern is equal to $2048 + (-32768) + 64 + 16 + 4$, for a total of -30636.

Graphics Displays

Displaying information on the Macintosh screen is done with graphics. There's a reference point on the screen called the *pen location, pen position,* or just *pen* which indicates where the next graphics will be drawn. Many BASIC commands, such as **PRINT,** draw at this point. Some ROM routines draw at horizontal and vertical offsets from this point, while others specify the coordinates where to place the graphics.

When displaying graphics to the *Output* window, it may be desirable to temporarily turn off the display with ROM routine **HIDEPEN:**

CALL HIDEPEN

This routine has no parameters. When called, it simply turns off the output to the *Output* window. Its counterpart, **SHOWPEN**, turns the output back on:

CALL SHOWPEN

Try Program 8-2. Press the mouse button once to hide the pen and once again to show it. You can stop the program by selecting *Stop* from the *Run* menu.

Program 8-2. HIDEPEN and SHOWPEN

```
SHPEN=1

LOOP:
CLS
FOR VERT=10 TO 100 STEP 10
   FOR HOR=0 TO 400 STEP 5
      PSET (HOR,VERT)
```

```
CALL MOVETO(10,120):PRINT HOR,VERT
IF MOUSE(0)>0 THEN IF SHPEN=1 THEN CALL HIDEPEN:SH
PEN=0 ELSE CALL SHOWPEN:SHPEN=1
  NEXT HOR
NEXT VERT
GOTO LOOP
```

Program 8-2 draws ten lines of dots. After each dot is displayed, the coordinates are placed at the bottom left of the *Output* window. While drawing the dots, the program scans for button activity. When the button is pressed, all output stops, though the program continues. Dots are still being drawn, but they're invisible. When the button is pressed again, output resumes.

As soon as ten lines have been drawn, the screen is erased. However, if the output is turned off when the program clears the *Output* window, nothing appears to happen.

Figure 8-4. *HIDEPEN and SHOWPEN turn the screen off and on.*
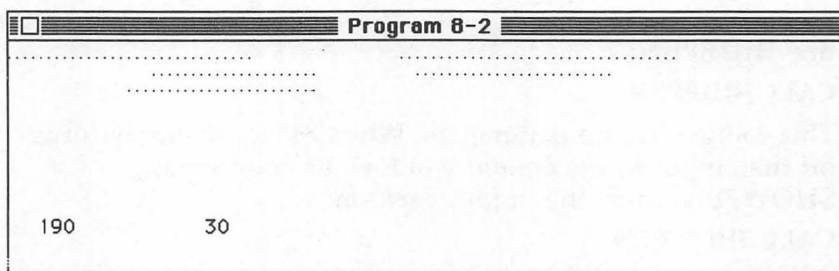
The current pen status, represented by SHPEN, is set to 1. If SHPEN is 1, the pen status is in show mode. Zero indicates that the pen is in hidden mode. The two **FOR-NEXT** loops determine the vertical and horizontal position at which to draw the dots. A dot is drawn with **PSET** and its position displayed at the bottom of the screen by the **PRINT** command.

If the pen is hidden, no dot will be drawn and the text will not be updated. **IF MOUSE(0)...** checks to see if the button has been pressed. If so, the pen status is toggled from visible to invisible with **HIDEPEN** or from invisible to visible with **SHOWPEN**. After ten rows of dots have been drawn, the program jumps back to LOOP where the dot plotting process

begins again. If the pen is currently hidden, the display window will not be cleared because **CLS** is also affected by the pen status.

PENMODE

Another ROM routine important to the way graphics are drawn on top of existing backgrounds is called **PENMODE**:

CALL PENMODE(*mode***)**

PENMODE draws subsequent graphics in a manner defined by the value of *mode*. Valid values range from 8 to 15, with the former as default. Any graphics drawn with a mode of 8 replace the background on which they're drawn. To see the effects of this and other modes, type in and run Program 8-3.

Program 8-3. PENMODE

```
CLS
DEFINT A-Z
DIM RECT(3),PAT(11)
FOR I=0 TO 11:READ PAT(I):NEXT I
CALL MOVETO(15,20)
PRINT "Mode";SPC(4);"Name";SPC(20);"Pattern"
FOR M=8 TO 15
   CALL MOVETO(20,20*(M-6)):PRINT USING "##";M
   READ NM$
   CALL MOVETO(65,20*(M-6)):PRINT NM$
   FOR J=0 TO 2
     RECT(0)=20*(M-7)+9:RECT(1)=110*J+135
     RECT(2)=RECT(0)+12
     RECT(3)=RECT(1)+90
     CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(J*4)))
     CALL FRAMERECT(VARPTR(RECT(0)))
     CALL PENMODE(M)
     CALL PENPAT(VARPTR(PAT(0)))
     CALL MOVETO(RECT(1),RECT(0)+4):CALL LINE(89,0)
     CALL PENPAT(VARPTR(PAT(8)))
     CALL MOVETO(RECT(1),RECT(0)+8):CALL LINE(89,0)
     CALL PENMODE(8)
   NEXT J
```

```
NEXT M
LINE (5,5)-(455,25),,B
LINE (5,5)-(60,190),,B
LINE (60,5)-(125,190),,B
LINE (125,5)-(455,190),,B
CALL MOVETO(120,220)
PRINT "Press the mouse button to continue"

READBUTTON:
IF MOUSE(0)=0 THEN READBUTTON
END
DATA 0,0,0,0:'white pat
DATA 21930,21930,21930,21930:'gray pat
DATA -1,-1,-1,-1:'black pat
DATA "Copy","OR","XOR","BIC","Not Copy","Not OR","Not XOR","Not
BIC"
```
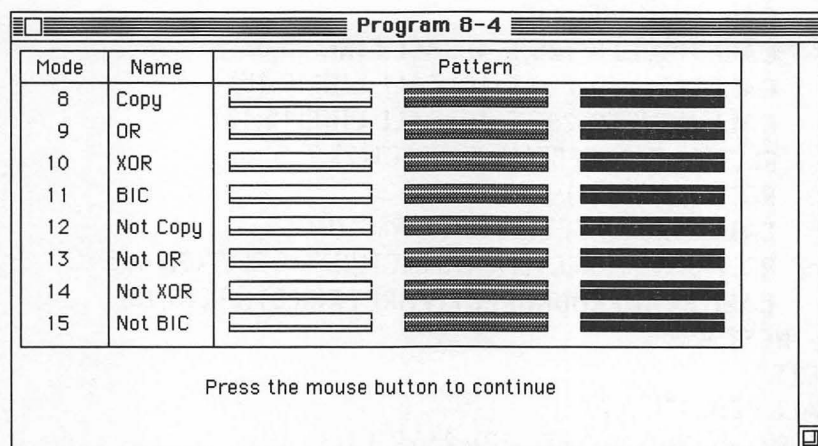
Figure 8-5. *The effect of using different pen modes when draw-*
ing lines through various backgrounds is illustrated
in the form of a table. The lines are drawn with
ROM routines instead of the BASIC LINE command.

Mode	Name	Pattern		
8	Copy			
9	OR			
10	XOR			
11	BIC			
12	Not Copy			
13	Not OR			
14	Not XOR			
15	Not BIC			

Press the mouse button to continue

This table illustrates the effects of different pen modes
when drawing lines atop three different backgrounds. The
backgrounds are placed in boxes and are tested against white

and black lines as pen patterns. The effects are demonstrated with patterns of white, gray, and black.

RECT(3) is a rectangle array used to define the boxes which contain the test patterns. PAT(11) is a pattern array representing the values of the three test patterns. After reading the data values for the patterns, the column titles are positioned with **MOVETO** and then displayed by **PRINT**.

The main part of the program starts with a **FOR-NEXT** loop that uses the counter M to represent the value of the pen mode tested. The first column of the table is the value of the pen mode, and the second column lists the names given to these modes. The **FOR J=0 TO** 2 loop draws a set of three test boxes, fills each with a test pattern using **FILLRECT**, and frames them with **FRAMERECT**. The pen mode is set with **PENMODE (CALL PENMODE(M))**. A white line is drawn through the set of boxes by the next line, and the following line draws a black line. **PENPAT** sets the color of the lines. **CALL PENMODE(8)** resets the pen mode back to default.

The first pen mode has a value of 8, called the *Copy* mode. When using this mode, the color of the pen will be displayed regardless of the background—white is drawn over black and vice versa. The *OR* mode gets its name from its similarity to the Boolean **OR** operation—the only drawing that registers with this mode results from drawing black over white; otherwise, the background remains unchanged. In the third mode, *XOR*, whenever white is drawn, nothing shows and the background remains unchanged. When black is drawn, the background is inverted wherever the black appears. (This is a popular mode because by drawing an object twice in this mode and then repositioning the object, you can simulate animation without destroying the background.) The final mode, Black Is Changed, or *BIC*, is similar to drawing in the *OR* mode except that black is changed to white. Like the *OR* mode, drawing in white is ignored. There are four more *Not* modes, identical to their counterparts except that all drawing in black is treated as white and white is treated as black.

PENMODE works very well when drawing with other ROM routines. However, there are problems when using it with the BASIC **LINE** command. Program 8-4 replaces the ROM routines with BASIC commands to draw lines.

Note: The only changes you must make from Program 8-3 are to delete the four lines between **CALL PENMODE(M)** and

E I G H T

CALL PENMODE(8) and insert the two LINE statements you
see in the program listing below.

Program 8-4. Using LINE
```
CLS
DEFINT A-Z
DIM RECT(3),PAT(11)
FOR I=0 TO 11:READ PAT(I):NEXT I
CALL MOVETO(15,20)
PRINT "Mode";SPC(4);"Name";SPC(20);"Pattern"
FOR M=8 TO 15
   CALL MOVETO(20,20*(M-6)):PRINT USING "##";M
   READ NM$
   CALL MOVETO(65,20*(M-6)):PRINT NM$
   FOR J=0 TO 2
      RECT(0)=20*(M-7)+9:RECT(1)=110*J+135
      RECT(2)=RECT(0)+12
      RECT(3)=RECT(1)+90
      CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(J*4)))
      CALL FRAMERECT(VARPTR(RECT(0)))
      CALL PENMODE(M)
      LINE(RECT(1),RECT(0)+4) - STEP (89,0),30
      LINE(RECT(1),RECT(0)+8) - STEP (89,0),33
      CALL PENMODE(8)
   NEXT J
NEXT M
LINE (5,5)-(455,25),,B
LINE (5,5)-(60,190),,B
LINE (60,5)-(125,190),,B
LINE (125,5)-(455,190),,B
CALL MOVETO(120,220)
PRINT "Press the mouse button to continue"

READBUTTON:
IF MOUSE(0)=0 THEN READBUTTON
END
DATA 0,0,0,0:'white pat
DATA 21930,21930,21930,21930:'gray pat
```

DATA -1,-1,-1,-1:'black pat
DATA "Copy","OR","XOR","BIC","Not Copy","Not OR","Not XOR","Not BIC"

The results are as though the pen mode stayed in *Copy* mode. ROM routines should be used with **PENMODE** as much as possible to get consistent results.

Figure 8-6. *PENMODE affects only graphics drawn by ROM routines.*

Mode	Name	Pattern		
8	Copy			
9	OR			
10	XOR			
11	BIC			
12	Not Copy			
13	Not OR			
14	Not XOR			
15	Not BIC			

Program 8-4

Press the mouse button to continue

PENSIZE

The pen also has an attribute called *size*. By plotting a point with the default pen, it produces a dot one pixel high and one pixel wide. This size can be altered with

CALL PENSIZE(*pen width,pen height*)

PENSIZE redefines the size of the pen to the width and height you specify. In Microsoft BASIC 2.0, only ROM routines are affected by pen size. This differs considerably from earlier versions of BASIC on the Macintosh.

Program 8-5. PENSIZE

```
CLS
DEFINT A-Z
DIM RECT(3)
```

```
CALL MOVETO(35,20):PRINT "Pensize";SPC(23);"Shapes"
CALL MOVETO(15,40)
PRINT "Width  Height  Hor. Line  Vert. Line  Diag. Line  Fram
e  Round Rect."
FOR H=1 TO 3
  FOR W=1 TO 3
    T=(H-1)*60+(W+2)*20
    CALL MOVETO(22,T):PRINT W
    CALL MOVETO(65,T):PRINT H
    CALL PENSIZE(W,H)
    CALL MOVETO(125,T-4):CALL LINE(25,0)
    CALL MOVETO(220,T-10):CALL LINE(0,15)
    CALL MOVETO(285,T-10):CALL LINE(15,15)
    RECT(0)=T-10:RECT(1)=340:RECT(2)=T+5
    RECT(3)=RECT(1)+50
    CALL FRAMERECT(VARPTR(RECT(0)))
    RECT(0)=T-10:RECT(1)=415:RECT(2)=T+5:RECT(3)=465
    CALL FRAMEROUNDRECT(VARPTR(RECT(0)),15,15)
  NEXT W
NEXT H
CALL PENSIZE(2,2)
RECT(0)=5:RECT(1)=5:RECT(2)=236:RECT(3)=481
CALL FRAMERECT(VARPTR(RECT(0)))
CALL PENSIZE(1,1)
LINE (5,25)-(480,45),,B
LINE (105,5)-(105,235)
LINE (56,25)-(56,235)
LINE (180,25)-(180,235)
LINE (257,25)-(257,235)
LINE (332,25)-(332,235)
LINE (395,25)-(395,235)
CALL MOVETO(120,260)
PRINT "Press the mouse button to continue.";
WHILE MOUSE(0)=0:WEND
```

Figure 8-7. *This chart displays the effects that occur to lines and shapes when altering the pen size with PENSIZE.*

Program 8-5

Pensize		Shapes				
Width	Height	Hor. Line	Vert. Line	Diag. Line	Frame	Round Rect.
1	1	—				
2	1	—				
3	1	—				
1	2	—				
2	2	—				
3	2	—				
1	3	—				
2	3	—				
3	3	—				

Press the mouse button to continue.
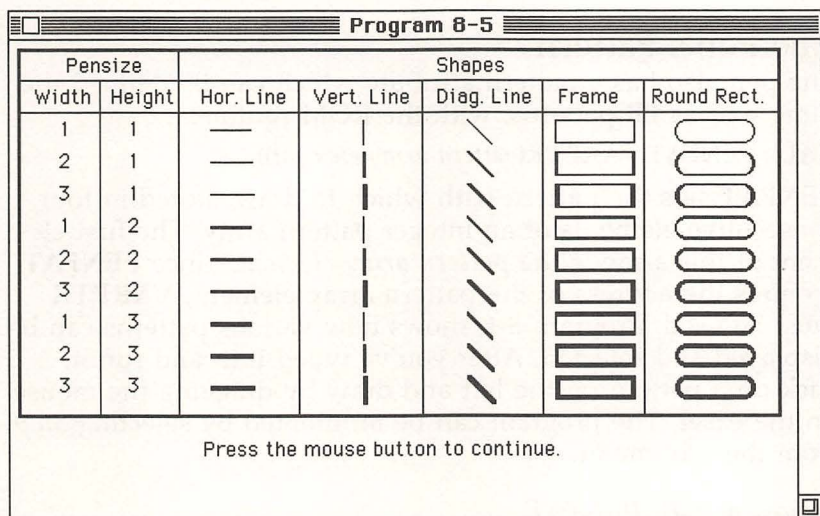
Pen width and *pen height* are varied with all nine permutations of combining widths and heights ranging from one to three. Lines are drawn horizontally, vertically, and diagonally to show the component effects of using various values. Also, a rectangle and a rounded-corner rectangle are manipulated. After the chart is drawn, the program waits for the button to be pressed.

A rectangle array, RECT(3), is dimensioned to draw the rectangles and rounded-corner rectangles. The main loop of the program lies within the **FOR** H=1 **TO** 3 loop. Here, H represents pen height and W, pen width. A value for T is calculated as a reference point for the vertical location of each line of the chart. The pen dimensions are set with **CALL PENSIZE**. The next three lines, each containing a **CALL MOVETO** and **CALL LINE** command, draw the horizontal, vertical, and diagonal lines. The rectangle array is redefined and drawn with **FRAMERECT**. Similarly, the rounded-corner rectangle is defined and drawn with **FRAMEROUNDRECT**.

Though the chart contents are completed, the various borders and column dividers have yet to be drawn. The thicker outer border of the chart is created by using **CALL**

145

PENSIZE(2,2), then using another **CALL FRAMERECT** command. The pen size is reset to (1,1) before the rest of the line work is performed by several **LINE** statements.

The Pen's Pattern

The pen also has a pattern attribute which can be defined the same way as fill patterns, with the ROM routine:

CALL PENPAT(VARPTR(*pattern array element***))**

PENPAT sets the pattern with which to draw, stored in four consecutive elements of an integer pattern array. The first element of this array is the *pattern array element*. Since **PENPAT** requires the address of the pattern array element, **VARPTR** must be used. Program 8-6 shows how various patterns can be displayed and selected. After you've typed it in and run it, click on a pattern on the left and draw by dragging the mouse on the easel. The program can be terminated by selecting *Stop* from the *Run* menu.

Program 8-6. PENPAT

```
CLS
DEFINT A-Z
DIM PAT(47),BUTTN(3,11),EASEL(3)
EASEL(0)=10:EASEL(1)=90:EASEL(2)=260:EASEL(3)=470
CALL FRAMERECT(VARPTR(EASEL(0)))
EASEL(2)=252:EASEL(3)=462
FOR P=0 TO 47:READ PAT(P):NEXT P
FOR P=0 TO 11
   BUTTN(0,P)=10+(P\2)*40:BUTTN(1,P)=(P AND 1)*40+10
   BUTTN(2,P)=BUTTN(0,P)+30:BUTTN(3,P)=BUTTN(1,P)+30
   CALL FILLRECT(VARPTR(BUTTN(0,P)),VARPTR(PAT(P*4)))
   CALL FRAMERECT(VARPTR(BUTTN(0,P)))
NEXT P
CALL PENSIZE(8,8):CALL PENPAT(VARPTR(PAT(0)))

LOOP:
WHILE MOUSE(0)>-1:WEND
X=MOUSE(1):Y=MOUSE(2)
IF X>EASEL(1) AND X<EASEL(3) AND Y>EASEL(0) AND Y<EASEL(2)
 THEN CALL MOVETO(X,Y):CALL LINE(0,0):GOTO LOOP
```

```
BUTSEL=-1:P=0
WHILE (BUTSEL=-1) AND (P<12)
IF X>BUTTN(1,P) AND X<BUTTN(3,P) AND Y>BUTTN(0,P) AND Y<BU
TTN(2,P) THEN BUTSEL=P
P=P+1:WEND
IF BUTSEL>-1 THEN CALL PENPAT(VARPTR(PAT(BUTSEL*4))):B
UTSEL=-1
GOTO LOOP
END
DATA -1,-1,-1,-1
DATA 0,0,0,0
DATA 21930,21930,21930,21930
DATA 4420,4420,4420,4420
DATA -30720,8704,-30720,8704
DATA -30601,0,-30601,0
DATA 16546,1296,16546,1296
DATA 612,-26606,-17326,4120
DATA -30652,8721,-30652,8721
DATA 4386,17544,4386,17544
DATA 4369,4369,4369,4369
DATA -256,0,-256,0
```

At the start, the drawing pattern is black, but it can be
changed by moving the mouse pointer to a new pattern box
and clicking the button. To draw, simply move the pointer to
within the easel boundaries and hold the button down while
dragging the mouse. The program continues to run until you
choose *Stop* from the *Run* menu.

Figure 8-8. *PENPAT controls the pattern used by graphic ROM routines.*
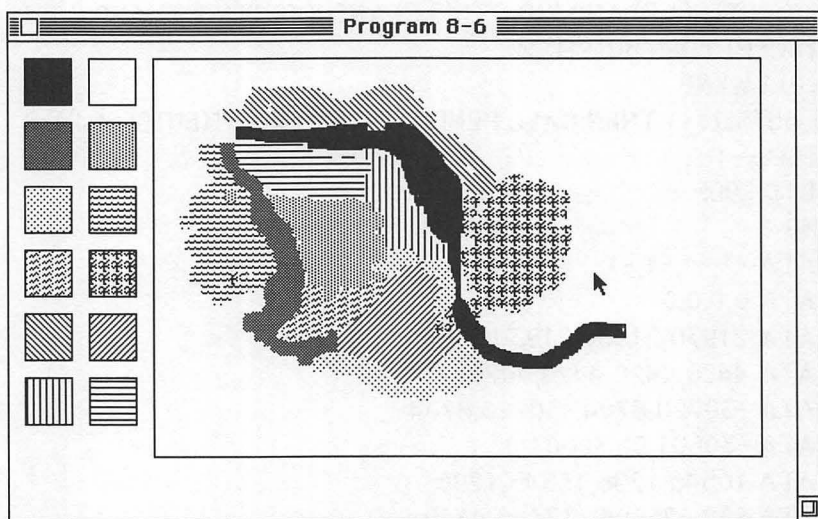
The program uses three arrays—PAT(47), BUTTN(3,11), and EASEL(3). PAT(47) is a pattern array containing data defining 12 patterns. Since each pattern is defined by four integers, the pattern array element for pattern P can be found at PAT(P*4), with the first pattern being pattern 0. BUTTN(3,11) is a set of 12 rectangle arrays for the boxes holding the 12 drawing patterns. The rectangle array element of box x is equal to BUTTN(0,x), which contains the value of the location of the top of box x. Similarly, the values of the location of the left, bottom, and right sides of box x are stored in BUTTN(1,x), BUTTN(2,x), and BUTTN(3,x). There are 12 boxes, numbered 0 to 11. The last rectangle array is EASEL(3). It defines the drawing bounds of the easel drawn with **FRAMERECT**. The drawing boundary is later modified to account for the size of the pen.

The **FOR** P=0 **TO** 11 loop draws the pattern boxes to the left of the easel. The boxes' positions are calculated, filled with a pattern with **FILLRECT**, and framed with **FRAMERECT**.

Pen size is set at eight pixels wide by eight pixels high (**PENSIZE(8,8)**). (You can alter this size if you want a wider or narrower brush stroke.) The pattern is set to black with **PENPAT**.

The **WHILE-WEND** statement holds the program until the mouse button is pressed (**MOUSE**(0)> −1). **MOUSE**(1) and **MOUSE**(2) record the horizontal and vertical positions of the pointer in X and Y. If the pointer is within the easel boundary (as checked for by the **IF-THEN** statement), the current pen pattern is plotted at the pointer position with **CALL MOVETO** and **CALL LINE**. If the pointer is *not* on the easel, the program determines if the pointer was located in any of the 12 pattern boxes. BUTSEL is set to −1 to indicate that none of the pattern boxes has been selected. If the value changes, it contains the box number selected—the first box to be checked is number 0. P is used for the box number being checked. The next **IF-THEN** compares the bounds of each of the pattern boxes with the coordinates of the pointer. The value of BUTSEL will indicate which, if any, of the 12 pattern boxes have been chosen. If one has, the pen pattern is set with **PENPAT**.

To restore the pen back to normal, or to its default attributes (resetting the pen mode, pen size, and pen pattern), you'll use the **PENNORMAL** ROM routine. Its syntax is simple:

CALL PENNORMAL

CHAPTER

9
ROM
Routines—
Graphics

9
ROM Routines— Graphics

Two ROM routines change the pen location. You've already seen one demonstrated in the last chapter—**MOVETO**:

CALL MOVETO(*horizontal position,vertical position*)

This moves the pen location to the coordinates specified by *horizontal* and *vertical positions*, the location being relative to the *Output* window. This is identical to similar commands for BASIC graphics. The topmost point is below the title bar of the *Output* window. This routine is useful for direct cursor placement when printing messages.

Take a look at Program 9-1, which demonstrates **MOVETO** more specifically.

Program 9-1. MOVETO

```
CLS
FOR I=0 TO 6.2 STEP .25
   CALL MOVETO(220+200*COS(I),100+80*SIN(I))
   PRINT CHR$(217)
NEXT I
CALL MOVETO(180,95)
PRINT"The Rabbit Center"
```

Program 9-1 draws a group of figures in a circle.

The **FOR-NEXT** loop draws the figures. The **CALL MOVETO** positions the pen location. The position is calculated with a little trigonometry using the built-in BASIC functions **COS** and **SIN**. The figures are draw with **CHR$**(217). A second **MOVETO** repositions the pen to the center of the circle before the message is displayed.

Figure 9-1. *The* Output *window's display.*
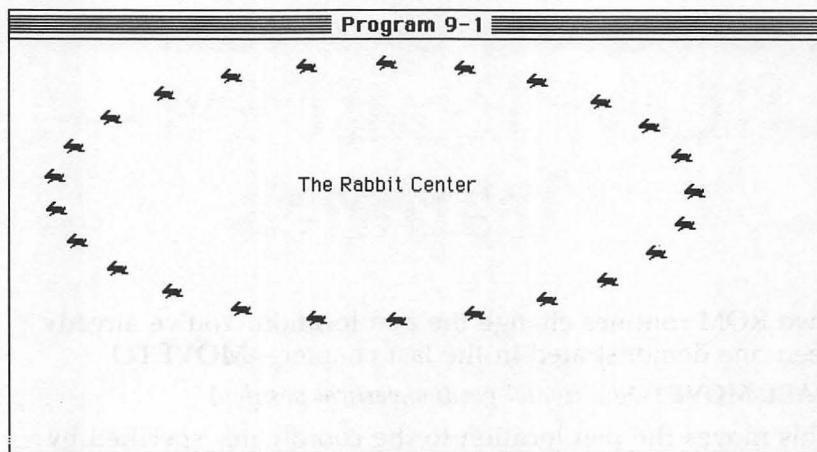
The pen location can also be moved to a position *relative* to its current position with the ROM routine MOVE:

CALL MOVE(*horizontal offset,vertical offset*)

This displaces the pen position by the *horizontal* and *vertical offsets*. A positive horizontal offset moves the pen position right, while a negative value moves it left. A positive vertical offset moves the pen position down, and a negative offset moves the pen position up. Program 9-2 is an example of the MOVE ROM routine. When the prompt appears, type your name and press Return. Pressing Return without any input stops the program.

Program 9-2. MOVE

```
START:
CLS
LINE INPUT "Enter your name: ";NM$
IF LEN(NM$)=0 THEN END
CALL MOVETO(10,30)
FOR I=LEN(NM$) TO 1 STEP -1
   PRINT MID$(NM$,I,1);
   CALL MOVE(10,10)
NEXT I
CALL MOVETO(10,180)
```

```
PRINT"Press a key ";

GETKEY:
IF INKEY$="" THEN GETKEY
GOTO START
```

After you enter a name, the program prints it both backward and diagonally.

Figure 9-2. *MOVETO and MOVE are used to position the characters in the* Output *window.*
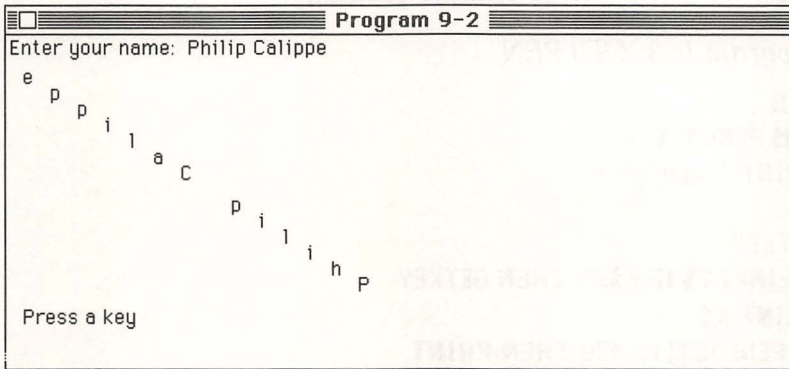
LINE INPUT places your name into the variable NM$. If only Return is pressed, NM$'s length is zero and the program ends. Otherwise, the pen position is set to coordinates (10,30) with **CALL MOVETO**. The **FOR-NEXT** loop prints the characters of NM$ in reverse order, diagonally toward the lower right of the screen. After each character appears, the pen position is shifted ten pixels right and ten down—**CALL MOVETO**(10,10). Inverse printing is done by letting the index into the string, I, point to the last character of the string (which is also the length of the string). I is decremented (by **STEP** −1) until it equals one and points to the first character of the string. The *Ith* character is printed with **MID$** using I as an index into NM$ and extracting a string with a length equal to one. Finally, the pen position is relocated near the bottom of the screen—**CALL MOVETO**(10,180)—so that the message appears there.

155

NINE

Control Complements

Gaining control over pen positioning is aided by using a ROM routine called **GETPEN**, which returns the current pen location:

CALL GETPEN(VARPTR(*pen location array element***))**

GETPEN determines the current vertical and horizontal pen locations in an integer array called *pen location array*. The vertical position is stored in the *pen location array element*, while the horizontal location is stored in the element which follows. Type in Program 9-3 and enter a few lines of text. Select *Stop* from the *Run* menu to end the program.

Program 9-3. GETPEN

```
CLS
DIM PENLOC%(1)
PRINT "Type:";

GETKEY:
K$=INKEY$:IF K$="" THEN GETKEY
PRINT K$;
IF PENLOC%(1)>479 THEN PRINT
CALL GETPEN(VARPTR(PENLOC%(0)))
CALL MOVETO(10,270):PRINT "Hor:";PENLOC%(1);" Vert:";PENLO
C%(0);
CALL MOVETO(PENLOC%(1),PENLOC%(0))
GOTO GETKEY
```

Figure 9-3. *The position of the current character is displayed as each one is typed.*

```
≣☐≣══════════════ Program 9-3 ≣═════════════
Type:this is a test!



 Hor: 120   Vert: 12
```
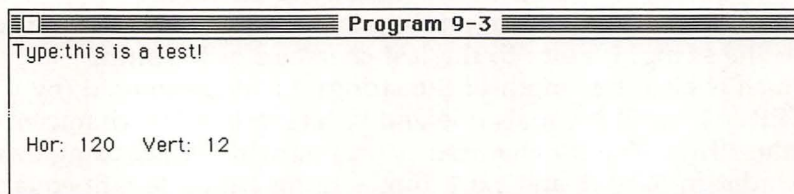
After each character is typed, the program checks to see if the next character entered will be outside the boundaries of

156

the *Output* window. If this is the case, the pen location is moved to the next line. Before going back to get another keystroke, the program displays the current pen location. All this is done with **PENLOC.**

After the screen is cleared, the program allocates a pen location array called PENLOC%(1). INKEY$ is used to determine if a key has been pressed. When one is pressed, it will be stored in K$. This character is displayed by **PRINT** K$ before the **IF-THEN** statement determines if the next character will be outside the *Output* window. The pen location is moved to the beginning of the next line with a **PRINT** command when the test is true. **GETPEN** finds the new pen location, which is then displayed at the bottom of the window. **MOVETO** must reposition the pen to its prior location.

Lines

Since the ROM routines are the basis of all Macintosh graphics, it's not surprising to find that there are ROM routines that duplicate some of the Microsoft BASIC graphic commands. Among these routines is one which draws lines:

CALL LINETO(*horizontal position,vertical position*)

LINETO draws a line from the current pen position to the coordinates specified by the *horizontal* and *vertical positions*. This routine is used when the destination point of the line is known.

If this location is better expressed as a point *relative* to the current pen position, then use the **LINE** ROM routine instead:

CALL LINE(*horizontal offset,vertical offset*)

LINE draws a line from the current pen position to a coordinate removed from that position horizontally and vertically by the values noted. Program 9-4 illustrates both **CALL LINE** and **CALL LINETO** in action.

Program 9-4. LINETO and LINE

```
CLS
CALL MOVETO(124,146)
PRINT "YES"
CALL MOVETO(95,45)
CALL LINETO(295,45)
CALL LINETO(295,175)
```

```
CALL LINETO(95,175)
CALL LINETO(95,45)
CALL MOVETO(119,131)
CALL LINE(32,0)
CALL LINE(4,4)
CALL LINE(0,12)
CALL LINE(-4,4)
CALL LINE(-32,0)
CALL LINE(-4,-4)
CALL LINE(0,-12)
CALL LINE(4,-4)
CALL MOVETO(105,65)
PRINT "If you see this Alert Box,"
CALL MOVE(105,0)
PRINT "Press the YES button."
CALL MOVETO(190,120)
PRINT "Alert Box"

READBUT:
WHILE MOUSE(0)<1:WEND
IF (MOUSE(1)<115 ) OR (MOUSE(1)>155) OR (MOUSE(2)<135) OR
(MOUSE(2)>155) THEN READBUT
```

You'll see an alert box, which waits for you to select the
YES button with the mouse.

Before printing the button title, the pen position is established with a call to **MOVETO**. Once *YES* is displayed, the
pen is repositioned and four **LINETO** calls are made to draw
the alert box. The pen is again "picked up" with **MOVETO**,
and the button inside the box is created with eight **LINE** calls.

The first part of the alert message (*If you see this Alert
Box*) is printed once the pen is in position. Note that **MOVE**,
not **MOVETO**, is used to put the pen in the proper place for
the second half of the message (*Press the YES button*). **MOVE**
is necessary since the vertical location of the pen is unknown
after the **PRINT** statement. However, it *is* known that the pen
location is at an appropriate vertical position for the next
string and that the horizontal location is zero. Therefore, it has
to be moved to the right 105 pixels to be lined up with the

Figure 9-4. *The alert box is drawn with a set of LINEs and LINETOs.*

```
═══════════════════ Program 9-4 ═══════════════════

    ┌─────────────────────────────┐
    │ If you see this Alert Box,   │
    │ Press the YES button.        │
    │                             │
    │              Alert Box       │
    │  ┌─────┐                     │
    │  │ YES │                     │
    │  └─────┘  ▶                  │
    └─────────────────────────────┘



                                              ▭
```
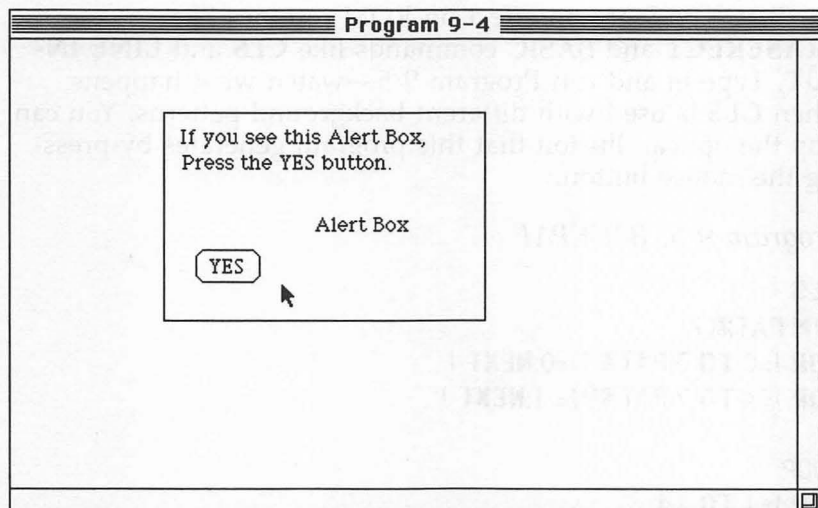
first string. The third part of the alert box message is positioned and displayed.

The READBUT routine waits until the mouse button is pressed before allowing program execution to continue. If the mouse's coordinates when its button was pressed were not within the boundary of the YES button, execution returns to the start of the routine. Otherwise, the program ends.

Background Patterns

Some ROM routines display graphics which affect an area of the screen. The area contents usually change as one pattern replaces another. When the resulting pattern is the same as the pattern that clears the screen, it's called the *background pattern* (white by default.) Using the BASIC command **CLS**, for instance, clears the *Output* window to white. You can change the background pattern with the ROM routine **BACKPAT**:

CALL BACKPAT(VARPTR(*pattern array element***))**

BACKPAT sets the background pattern to the pattern defined by the contents of the integer pattern array. The first element of the array, the *pattern array element*, defines the first two rows of the eight-row bit pattern. Since **BACKPAT** requires

159

the address of the pattern array element, **VARPTR** is mandatory.

BACKPAT has an effect on ROM routines like **ERASERECT** and BASIC commands like **CLS** and **LINE IN-PUT**. Type in and run Program 9-5—watch what happens when **CLS** is used with different background patterns. You can stop the optical illusion that this program generates by pressing the mouse button.

Program 9-5. BACKPAT

```
CLS
DIM PAT%(7)
FOR I=0 TO 3:PAT%(I)=0:NEXT I
FOR I=4 TO 7:PAT%(I)= 1:NEXT I

LOOP:
FOR I=1 TO 14
  CALL BACKPAT(VARPTR(PAT%(4)))
  CLS
  FOR J=4 TO 7:PAT%(J)=PAT%(J)*2:NEXT J
NEXT I
FOR I=1 TO 14
  CALL BACKPAT(VARPTR(PAT%(4)))
  CLS
  FOR J=4 TO 7:PAT%(J)=PAT%(J)/2:NEXT J
NEXT I
IF MOUSE(0)<1 THEN LOOP
CALL BACKPAT(VARPTR(PAT%(0)))
CLS
```

Program 9-5 cycles through a sequence of background patterns to make the *Output* window appear to shimmer.

As with the other programs in this book, the first line clears the *Output* window to white—the default background pattern. A pattern array, PAT%(7), is then dimensioned. The array will contain two patterns—the first pattern is white and has a value of zero for each of its data elements, used to reset the background pattern to white. The second pattern is initialized to a set of dots by the line **FOR** I=4 **TO** 7:PAT%(I)=1 :**NEXT** I; each of that pattern's data elements is one. The loop

Figure 9-5. *The different backgrounds are set with BACKPAT.*
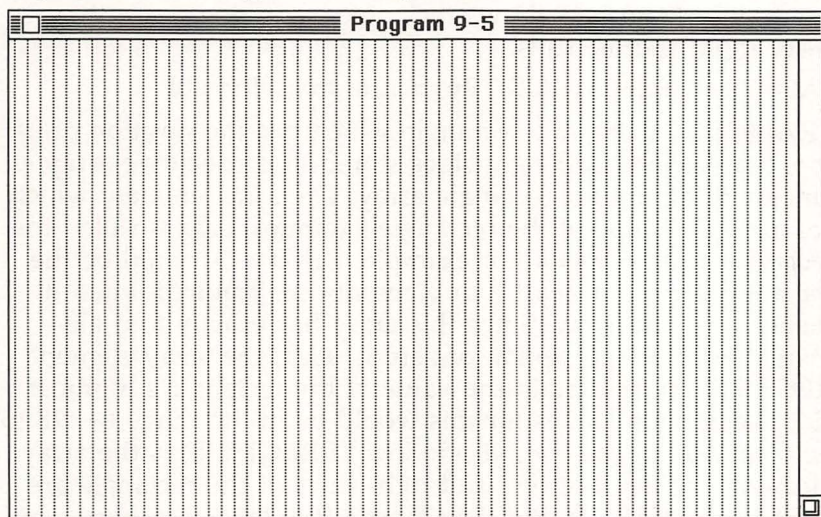
beginning with **FOR** I=1 **TO** 14 draws the background pattern and shifts it to the left by one pixel; this is done 14 times for the first half of the shimmer. **BACKPAT** is called to establish a new background pattern. **CLS** draws the background pattern, and the next line recalculates the values of the pattern array.

The second major **FOR-NEXT** loop draws the background pattern, iterating 14 times, while shifting the background pattern to the right by one pixel. Again **BACKPAT** is called to establish a new background pattern, and **CLS** draws the background pattern before the values for the pattern array are recalculated. This cycle is repeated until the program detects the press of the mouse button.

Changing the background pattern can be useful with input boxes and software utilities. The background pattern should equal the pattern on which the input is being made so that typing errors can be deleted with the Backspace key, yet leave the correct pattern. If the patterns don't match, you'll end up with unsightly gaps in the input box.

Rectangles

Drawing boxes with the **LINE** and **LINETO** ROM routines is one way to create the necessary shapes on your Macintosh

screen, but there's another approach that makes the job much easier. **FRAMERECT** draws rectangles, and does so with fewer ROM routine calls:

CALL FRAMERECT(VARPTR(*rectangle array element***))**

FRAMERECT draws a rectangle which has the position of its top, left, bottom, and right sides stored in consecutive array elements of an integer rectangle array. The array element which contains the value of the rectangle's topmost side is called the *rectangle array element*. For example, RECT%(3) could be the rectangle array—at least four elements are required when dimensioning the array. Since this array has exactly four elements when a base option of zero is used, the array element RECT%(0) must contain the value of the rectangle's topmost side. Legitimate values for this would be in the range 0–298 to be displayed in the *Output* window. If RECT%(0) is the array element, the left, bottom, and right sides of the rectangle are stored in RECT%(1), RECT%(2), and RECT%(3), respectively. After these values have been stored in the rectangle array, the ROM routine would be called with **CALL FRAMERECT (VARPTR(RECT%(0)))**. Remember that the parentheses must match. **FRAMERECT** requires the address of the rectangle array element; thus, VARPTR is used.

Take a look at Figure 9-6, which shows how to define not only rectangles, but round rectangles, ovals, and arcs. (The last three shapes have their own ROM calls, which are explored in more detail later in this chapter.)

Program 9-6 draws two rectangles on the screen. Type it in and run it. After you've seen what the program does, try experimenting a bit. Change the values of the RECT% array elements to increase or decrease the size of the two shapes.

Program 9-6. FRAMERECT

```
CLS
DIM RECT%(3)
CALL MOVETO(40,40)
PRINT "Type your name in the input box within"
CALL MOVE(40,0)
PRINT "this dialog box. Press 'RETURN'"
CALL MOVETO(40,120)
PRINT "Name"
```

```
RECT%(0)=25:RECT%(1)=35:RECT%(2)=150:RECT%(3)=300
CALL FRAMERECT(VARPTR(RECT%(0)))
RECT%(0)=105:RECT%(1)=80:RECT%(2)=125:RECT%(3)=280
CALL FRAMERECT(VARPTR(RECT%(0)))
CALL MOVETO (85,119)
LINE INPUT "";NM$
CALL MOVETO(10,18)
PRINT "Thank you!"
```

Figure 9-6. *Definitions of rectangles, round rectangles, ovals, and arcs.*
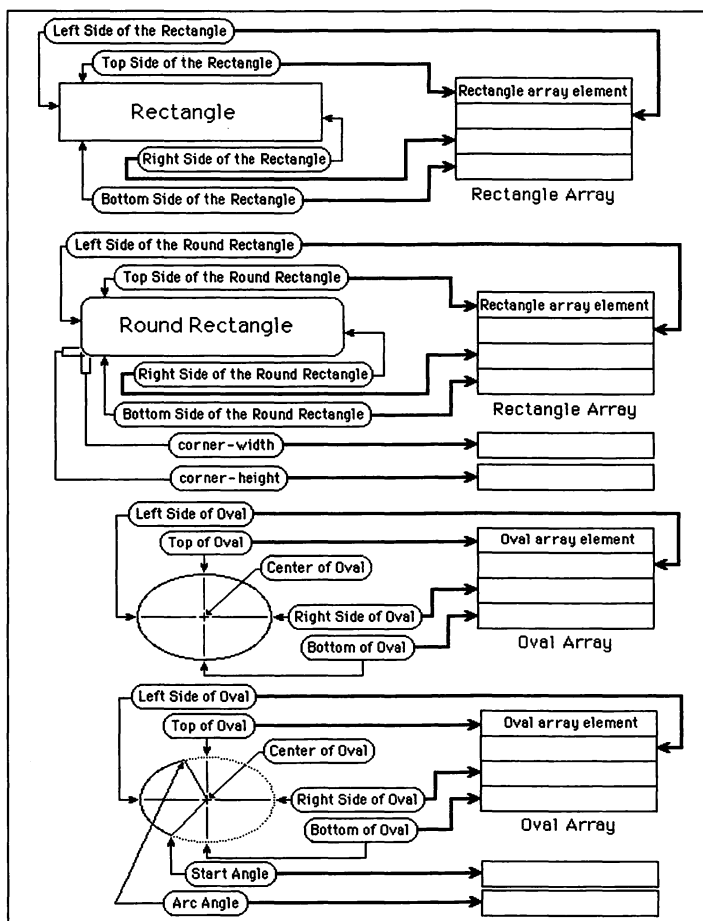
Figure 9-7. *The dialog box displayed with FRAMERECT.*

```
================ Program 9-6 ================

┌─────────────────────────────────────────┐
│  Type your name in the input box within  │
│  this dialog box. Press 'RETURN'         │
│                                          │
│                                          │
│  Name │Phil                          │   │
│       └──────────────────────────────┘   │
└─────────────────────────────────────────┘
```
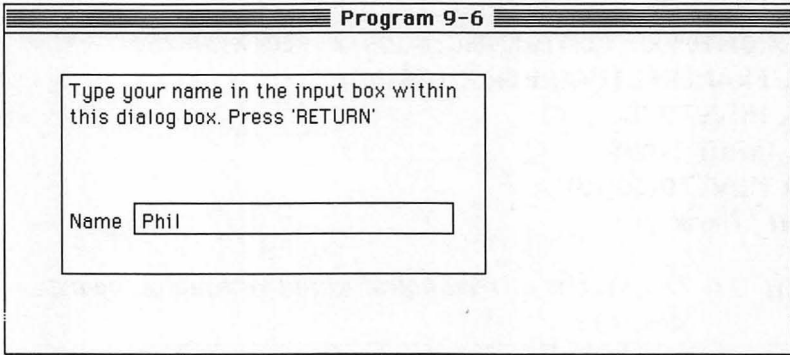
The rectangle array is dimensioned in the second line as RECT%(3). The message inside the box is positioned with **MOVETO** and **MOVE** before being displayed by **PRINT**. Similarly, the input box title is positioned and displayed by another **MOVETO** and **PRINT** combination. The top, left, bottom, and right sides of the dialog box are stored in RECT%(3); then it's drawn with **FRAMERECT** by the next line. The input box is defined and displayed in the same manner. The pen position is set to the inside left of the input box with **MOVETO** in preparation for **LINE INPUT**. When the Return key is pressed, your name is stored in NM$.

Filling Rectangles

Rectangles can be created and filled with patterns using a ROM routine called **FILLRECT**:

CALL FILLRECT(VARPTR(*rectangle array element***),VARPTR(***pattern array element***))**

FILLRECT draws a rectangle filled with a pattern. The rectangle array element is defined as when using **FRAMERECT**. The pattern used is defined by the contents of the integer pattern array—the first element of the array is the *pattern array element*, which defines the first two rows of the eight-row bit pattern. Look at Program 9-7 to see how to use this routine. Press the mouse button in any of the four boxes drawn on the left side of the *Output* window. Press a key, then the mouse button, to stop the program.

Program 9-7. FILLRECT

```
CLS
DIM RECT%(3),PAT%(15),BUTTN%(3,3),TRECT%(3)
FOR I=0 TO 15:READ PAT%(I):NEXT I
FOR I=0 TO 3:BUTTN%(I,0)=40*I+10:BUTTN%(I,1)=10:BUTTN%(I,2)
=40*I+40:BUTTN%(I,3)=40:NEXT I
FOR I=0 TO 3:FOR J=0 TO 3:TRECT%(J)=BUTTN%(I,J):NEXT J:CAL
L FILLRECT(VARPTR(TRECT%(0)),VARPTR(PAT%(I*4))):CALL F
RAMERECT(VARPTR(TRECT%(0))):NEXT I
RECT%(0)=10:RECT%(1)=50:RECT%(2)=160:RECT%(3)=350
CALL FRAMERECT(VARPTR(RECT%(0)))
BUTSEL= -1
WHILE INKEY$=""
  WHILE MOUSE(0)<1:WEND
  FOR I=0 TO 3
    IF MOUSE(2)>BUTTN%(I,0) AND MOUSE(1)>BUTTN%(I,1) AND
  MOUSE(2)<BUTTN%(I,2) AND MOUSE(1)<BUTTN%(I,3) THEN BUTS
EL=I
  NEXT I
    IF BUTSEL> -1 THEN CALL FILLRECT(VARPTR(RECT%(0)),VA
RPTR(PAT%(BUTSEL*4))):CALL FRAMERECT(VARPTR(RECT%(0)
)):BUTSEL = -1
WEND
END
DATA -1,-1,-1,-1:'pattern 1
DATA 21930,21930,21930,21930:'pattern 2
DATA 4420,4420,4420,4420:'pattern 3
DATA 0,0,0,0:'pattern 4
```

You'll see four rectangular buttons on the left side of the
Output window. Each is filled with one of four patterns—
black, gray, light gray, and white. A larger rectangle acts as an
easel. Whenever one of the four pattern buttons is selected
with the mouse, the easel fills with the chosen pattern.

The four arrays used by Program 9-7 are dimensioned in
the second line. RECT%(3) defines the easel's borders, the pat-
tern array PAT%(15) contains the four patterns, BUTTN%(3,3)
stores the locations of the four buttons on the *Output* window,

Figure 9-8. *The patterns are drawn with FILLRECT by selecting the appropriate box.*
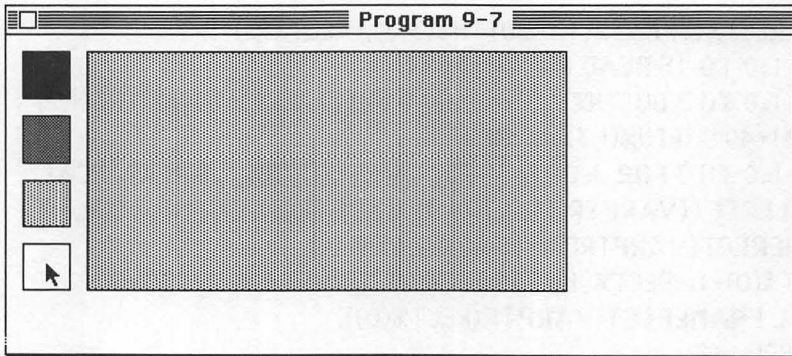
and TRECT%(3) is a temporary rectangle array used when drawing the buttons.

The element BUTTN%(x,0) contains the location of the top side of button x, while BUTTN%(x,1), BUTTN%(x,2), and BUTTN%(x,3) represent the locations of the left, bottom, and right sides of button x. Since x can be any integer value from 0 to 3, four buttons are defined.

The pattern definitions are read from **DATA** statements at the end of the program. (Two sets of pattern data which are easy to remember are those of black and white. Black is represented by four negative ones and white by four zeros.)

The location parameters of each of the four buttons are calculated and stored in BUTTN%(3,3). The following line draws the four buttons on the screen. First, the temporary rectangle array TRECT%(3) is assigned the values of the location of a button. Then the button's fill pattern is drawn with **FILLRECT**. Since **FILLRECT** does not draw a frame around the rectangle, **FRAMERECT** is also used. The variable *I* represents the button number being drawn. Since patterns are defined as a set of four integers, the pattern array element to be used by a particular button is located within PAT%(15) with an index of I*4.

After each of the four buttons is drawn, the location of the easel is defined and drawn. **FRAMERECT** is used again for this task.

BUTSEL contains the number of the button selected with the mouse. It will have a value from 0 to 3 if one of the but-

tons has been selected. Otherwise, its value is −1. Initially, it's assumed that no button has been selected and the appropriate value is assigned to **BUTSEL**.

The main loop of the program is a **WHILE-WEND** loop. The program remains in the loop until a key is pressed. When this happens, **INKEY$** will not return a null string and execution will resume, ending the program. To stop the program, a key must be pressed. The program may still require a mouse button click to exit the main loop if it's executing the **WHILE** statement at the time of the keypress.

The **FOR-NEXT** loop determines if any button has been pressed. If the coordinates of the mouse click are within one of the rectangle boundaries, the first **IF** statement notes it, and the number of the button is stored in BUTSEL. BUTSEL will no longer be equal to −1. The second **IF-THEN** tests for this condition and fills the easel with the appropriate pattern, done with **FILLRECT** and **FRAMERECT**. After updating the easel, BUTSEL is reset to −1.

Erasing Rectangles

Erasing rectangles is easy when performed with the ROM routine **ERASERECT**:

CALL ERASERECT(VARPTR(rectangle array element**))**

ERASERECT is the counterpart to **FILLRECT**—it erases the rectangle defined by the *array element*, replacing it with the background pattern defined by **BACKPAT** (white by default). Program 9-8 provides an example of **ERASERECT** in use.

Program 9-8. ERASERECT

```
DEFINT A-Z
DIM PAT(7),RECT(3)
FOR I=0 TO 7:READ PAT(I):NEXT I
CALL BACKPAT(VARPTR(PAT(4)))
CLS
FOR I=0 TO 3:READ RECT(I):NEXT I
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(RECT(0)))
CALL MOVETO(120,50):PRINT"Click inside this box.";

GETBUTTON:
```

```
WHILE MOUSE(0)<1:WEND
IF MOUSE(2)<RECT(0) OR MOUSE(1)<RECT(1) OR MOUSE(2)>RECT
(2) OR MOUSE(1)>RECT(3) THEN GETBUTTON
CALL ERASERECT(VARPTR(RECT(0)))
CALL BACKPAT(VARPTR(PAT(0)))
END
DATA 0,0,0,0
DATA 4420,4420,4420,4420
DATA 30,100,190,400
```

Program 9-8 generates a light gray background and a dialog box. Clicking within the box causes it to go away, leaving the background intact.

Figure 9-9. *When the dialog box is erased with ERASERECT, the background is restored.*
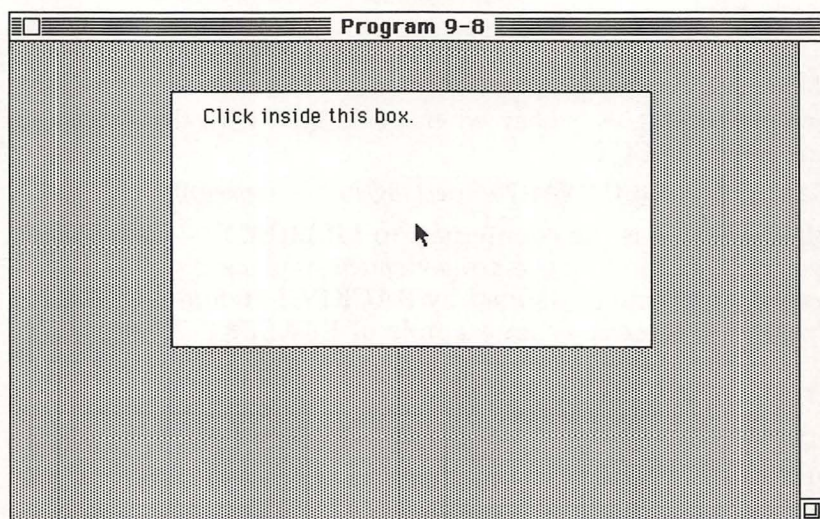
First of all, PAT(7) and RECT(3), the program's two arrays, are dimensioned. PAT(7) is a pattern array containing two patterns to change the background from white to light gray and back again. RECT(3) is a rectangle array to define the dialog box. The patterns are read from **DATA** statements at the end of the program. The background pattern is set to light gray with **BACKPAT**. **CLS** clears the *Output* window to the

background pattern. The parameters of the rectangle array are read from the last **DATA** statement in the program.

The dialog box is drawn in three parts—first, a rectangle is filled with white by **FILLRECT**. Then, the rectangle is framed by **FRAMERECT**. Finally, a message is placed in the box with **MOVETO** and **PRINT**.

The GETBUTTON routine waits for some mouse button activity. **MOUSE(1)** and **MOUSE(2)** check if the mouse pointer is within the box. If it's not, execution goes back to the beginning of the routine. Otherwise, the box is erased with **ERASERECT**. Before the program finishes, a final **CALL BACKPAT** resets the background pattern to white.

Inverting Patterns

Included with the ROM routines that erase various shapes is another set of routines which invert shapes, a process where all the white pixels within a shape become black and all the black pixels become white. One such routine is **INVERTRECT**:

CALL INVERTRECT(VARPTR(*rectangle array element***))**

INVERTRECT inverts the defined rectangle. Program 9-9 is an example of one way **INVERTRECT** can be used. Once you've typed in the program, click inside the box with the word *Menu*. Select some of the items. The program stops when you choose *Quit*.

Program 9-9. INVERTRECT

```
CLS
DEFINT A-Z
DIM RECT(3),TRECT(3),MEN$(5),EASELOR(3)
FOR I=0 TO 5:READ MEN$(I):NEXT I
CALL MOVETO(35,22):PRINT"Menu"
RECT(0)=10:RECT(1)=30:RECT(2)=26:RECT(3)=75
CALL FRAMERECT(VARPTR(RECT(0)))
GETBUTTON:
WHILE MOUSE(0)=0:WEND
X=MOUSE(1):Y=MOUSE(2)
IF Y<RECT(0) OR X<RECT(1) OR Y>RECT(2) OR X>RECT(3) THEN GE
TBUTTON
CALL INVERTRECT(VARPTR(RECT(0)))
```

```
FOR I=0 TO 5
  CALL MOVETO(40,38+16*I):PRINT MEN$(I);
NEXT I
RECT(0)=25:RECT(1)=31:RECT(2)=123:RECT(3)=120
CALL FRAMERECT(VARPTR(RECT(0)))
ITEM= -1:TRECT(1)=32:TRECT(3)=119
GETMOUSE:
WHILE MOUSE(0)<0
  X=MOUSE(1):Y=MOUSE(2)
  IF Y<RECT(0)+1 OR X<RECT(1) OR Y>RECT(2)-2 OR X>RECT(3) TH
EN UNSELECT
  NEWITEM=INT((Y-26)/16)
  IF ITEM=NEWITEM THEN GETMOUSE
  IF ITEM<> -1 THEN TRECT(0)=ITEM*16+26:TRECT(2)=TRECT(0)+
16:CALL INVERTRECT(VARPTR(TRECT(0)))
  ITEM=NEWITEM:TRECT(0)=ITEM*16+26:TRECT(2)=TRECT(0)+16:C
ALL INVERTRECT(VARPTR(TRECT(0)))
  GOTO GETMOUSE
UNSELECT:
  IF ITEM<> -1 THEN TRECT(0)=ITEM*16+26:TRECT(2)=TRECT(0)+
16:CALL INVERTRECT(VARPTR(TRECT(0))):ITEM= -1
WEND
CALL ERASERECT(VARPTR(RECT(0)))
RECT(0)=10:RECT(1)=30:RECT(2)=26:RECT(3)=75
CALL INVERTRECT(VARPTR(RECT(0)))
IF ITEM>=0 THEN ON ITEM+1 GOSUB CLREASEL,DWBOX,DWCIRCLE
,DWELLIPSE,DWROUNDRECT,QUIT
GOTO GETBUTTON
CLREASEL:
  EASELOR(0)=30:EASELOR(1)=200:EASELOR(2)=230:EASELOR(3)=
400
  CALL ERASERECT(VARPTR(EASELOR(0)))
  RETURN
DWBOX:
  EASELOR(0)=35:EASELOR(1)=290:EASELOR(2)=55:EASELOR(3)=3
10
  CALL FRAMERECT(VARPTR(EASELOR(0)))
  RETURN
```

170

```
DWCIRCLE:
   EASELOR(0)=60:EASELOR(1)=290:EASELOR(2)=80:EASELOR(3)=3
   10
   CALL FRAMEOVAL(VARPTR(EASELOR(0)))
   RETURN
DWELLIPSE:
   EASELOR(0)=85:EASELOR(1)=285:EASELOR(2)=105:EASELOR(3)=
   315
   CALL FRAMEOVAL(VARPTR(EASELOR(0)))
   RETURN
DWROUNDRECT:
   EASELOR(0)=110:EASELOR(1)=285:EASELOR(2)=130:EASELOR(3)
   =315
   CALL FRAMEROUNDRECT(VARPTR(EASELOR(0)),15,15)
   RETURN
QUIT:
   END
   RETURN
DATA "Clear","Box","Circle","Ellipse","Round Rect","Quit"
```

When you select an item from this menu, a corresponding
shape is drawn. You can clear all the shapes by choosing
Clear; Quit exits the program.

Figure 9-10. *The menu is functional, but located in the* Output
window. It is used like a menu at the top of the
screen. Select the shapes from the list of items in
the menu, and they will be drawn on the left side
of the screen. Select Clear *to erase the shapes.*
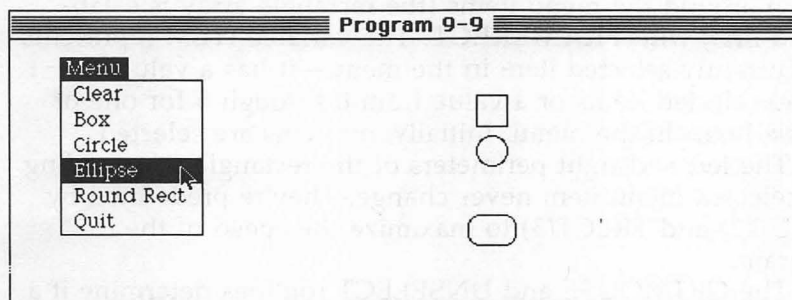
Program 9-9 is divided into two major areas. The main part of the program consists of the routines GETBUTTON, GETMOUSE, and UNSELECT, as well as the few lines at the program beginning. These routines and lines generate the menu and act in accordance with the menu selections. The second section contains a number of subroutines which are called when selections are made.

Four arrays are used: RECT(3), a rectangle array that draws the menu; TRECT(3), a rectangle array used when inverting the selected items in the menu; string array MEN$(5) containing the menu item names; and EASELOR(3), a general-purpose rectangle array or oval array required when drawing shapes. These arrays are allocated in the third line. The following line reads the menu item names from the **DATA** statement at the program's end. The remainder of the lines before the GETBUTTON subroutine create the menu title bar. Note that **FRAMERECT** draws the rectangle.

At this point, the menu has been defined and displayed. The next step is to wait until it's selected, accomplished by the **WHILE-WEND** loop in the GETBUTTON routine. As long as the mouse button has not been pressed, **MOUSE(0)** returns zero. When button activity occurs, program execution exits this loop and proceeds to the next line where the coordinates of the mouse pointer are recorded into X and Y. After testing to see if the coordinates in X and Y are outside the perimeter of the menu bar (if so, the action is ignored), program execution proceeds to the **INVERTRECT** call, which inverts the menu title.

The **FOR-NEXT** loop displays the item names of the menu; each name is positioned with **MOVETO** before it's printed. There are six items, numbered 0 through 5. A frame is drawn around the menu items (the rectangle array is established first) with **FRAMERECT**. The variable ITEM represents the currently selected item in the menu—it has a value of -1 for no selected items or a value from 0 through 5 for one of the six items in the menu. Initially, no items are selected.

The left and right perimeters of the rectangle surrounding the selected menu item never change. They're predefined by TRECT(1) and TRECT(3) to maximize the speed of the program.

The GETMOUSE and UNSELECT routines determine if a menu item is selected and, if so, which one. **WHILE**

MOUSE(0)<0 forces an exit from the loop when the button is let go. Thus, the menu remains showing as long as the mouse button remains pressed. Otherwise, action is taken according to which item was selected, if any, at the time the mouse button was released. Again, the mouse pointer location is monitored and recorded in X and Y. The location must be compared to the rectangle's area—if the pointer is not within the menu, the program must unselect any selected menu items. (If this is true, the program jumps to the UNSELECT routine.) Otherwise, NEWITEM contains the item number of the selection the pointer is on. If the new item selected equals the last item selected, no action is required. The new menu item must be highlighted and if another menu item is already highlighted, it must be "unhighlighted" by **INVERTRECT** before the new menu item is inverted.

When the mouse button is released, **ERASERECT** erases the menu listing from the screen. A rectangle array is defined, then used by **INVERTRECT** to return the inverted menu title back to normal. The **IF** ITEM>=0 statement determines if any items were selected by examining ITEM. If its value is not −1, then it contains the number of the item selected from the menu. An **ON-GOSUB** is performed (note that a value of 1 is added to the item number because the **ON-GOSUB** command cannot accept 0 as an index to calculate which subroutine to execute).

The various subroutines perform the tasks indicated by the items in the menu. Each subroutine first defines an array (the type depends on the shape to be drawn) which is then used by **ERASERECT, FRAMERECT, FRAMEOVAL,** or **FRAMEROUNDRECT** to draw the desired shape.

Painting

Shapes can be drawn with different patterns by using ROM routines prefixed with **PAINT**. One such ROM routine is **PAINTRECT**:

CALL PAINTRECT(VARPTR(*rectangle array element*)**)**

PAINTRECT fills a defined rectangle with the current pen pattern defined by **PENPAT**, black by default. Try Program 9-10. Press and hold the mouse button to fire the missile, which detonates when the button is released. The objective is to fill the target with black.

Program 9-10. PAINTRECT

```
DEFINT A-Z
DIM TRECT(3),PAT(3),MISSLE(2)
FOR I=0 TO 3:READ PAT(I):NEXT I
FOR I=0 TO 2:READ MISSLE(I):NEXT I
K$="Y"
WHILE K$="y" OR K$="Y"
  CLS
  TRECT(0)=129:TRECT(1)=300:TRECT(2)=151:TRECT(3)=332
  CALL FRAMERECT(VARPTR(TRECT(0)))
  TRECT(0)=130:TRECT(1)=0:TRECT(2)=150:TRECT(3)=10
  CALL FILLRECT(VARPTR(TRECT(0)),VARPTR(PAT(0)))
  SHELLS=5
  WHILE SHELLS>0
    LINE (0,140) - STEP (19,0)
    HX=4
    WHILE MOUSE(0)<>0:WEND
    WHILE MOUSE(0)=0:WEND
    WHILE (MOUSE(0)<0 AND HX<400)
      HX=HX+8:PUT (HX,140),MISSLE
      FOR I=0 TO 80:NEXT I
    WEND
    SHELLS=SHELLS-1
    PUT (HX+8,140)-(HX+15,140),MISSLE
    IF HX>323 OR HX<292 THEN EXPLODE
    TRECT(1)=HX+8:TRECT(3)=HX+16
    CALL PAINTRECT(VARPTR(TRECT(0)))

EXPLODE:
  WEND
  CALL MOVETO(20,20):PRINT "Score:";POINT(305,140)+POINT
(313,140)+POINT(321,140)+POINT(329,140)-120
  CALL MOVETO(20,40):PRINT "Play again?:";
  K$="":WHILE K$="":K$=INKEY$:WEND
WEND
END
DATA -1,-1,-1,-1
DATA 16,1,-1
```

A missile launcher is drawn on the left side of the window and a target on the right. By pressing and holding the mouse button, you launch the missile toward the target. It explodes when the mouse button is released. The objective of this game is to use five missiles or fewer to fill the target. Your shooting accuracy is scored and displayed at the end of each round. If you don't answer Y to the *Play again?* prompt, the program terminates.

Figure 9-11. *A nonflickering animation technique is used to draw the missile. The target is filled with PAINTRECT.*
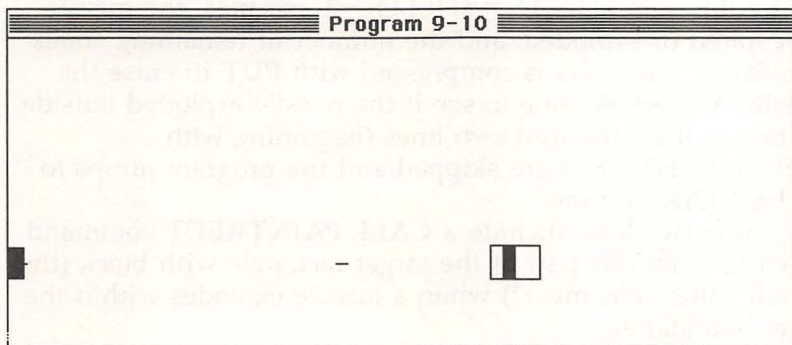
The second line dimensions three arrays—TRECT(3), a rectangle array that draws the missile launcher and target; PAT(3), a pattern array containing the fill pattern for the missile launcher; and MISSLE(2), the bit image of the missile. The data for PAT(3) and MISSLE(2) is read from DATA statements at the end of the program. K$ contains the response to the *Play again?* prompt. It's initially set to Y so that the first game can be played.

After initializing the game variables and preparing the screen, a **WHILE-WEND** loop executes once for each shell fired. (This loop starts with the statement **WHILE** SHELLS>0.) The **LINE** statement draws the missile in the launcher. Its horizontal position, HX, is set to 4. Several **MOUSE(0)** commands are used to make the program wait for the button to be inactive before starting the firing sequence and idling the program until the button is pressed. At that point, the missile fires at the target. HX, the missile's horizontal position, is repeatedly increased by eight pixels. The missile remains in flight until

175

either the mouse button is released or the missile goes past the target. The missile is redrawn with the **PUT** command.

This technique for animating the missile does not involve erasing the missile from its old position and then redrawing it at its new position. Instead, a mask is drawn on top of the missile which erases its trailing edge, while at the same time draws its leading edge. This mask is larger than the bit image of the missile, and the animation is fast and smooth. As long as the mouse button is not pressed, the missile keeps moving. (Note that a **FOR-NEXT** loop is used to *slow* the animation. If the loop is removed, the missile moves too fast to control. Try it out for yourself.)

By the time SHELLS=SHELLS−1 executes, the missile has expired or exploded, and the number of remaining shells is updated. The mask is compressed with **PUT** to erase the missile. A check is done to see if the missile exploded outside the target. If so, the next two lines (beginning with TRECT(1)=HX+8...) are skipped and the program jumps to the EXPLODE routine.

These two lines include a **CALL PAINTRECT** command which actually fills part of the target rectangle with black (the default color, remember?) when a missile explodes within the target boundaries.

Rounded Rectangles

Another type of rectangle can be drawn with the ROM routine called **FRAMEROUNDRECT**:

CALL FRAMEROUNDRECT(VARPTR(*rectangle array element***),*corner-width,corner-height***)**

FRAMEROUNDRECT draws a rectangle which has the position of its top, left, bottom, and right sides stored in this order in consecutive array elements of an integer rectangle array. The element containing the value of the rectangle's topmost side is the *rectangle array element*. Unlike **FRAMERECT**, **FRAMEROUNDRECT** draws a rectangle with rounded corners. The rounded corners are equal to a quarter of an oval whose width is equal to the value of *corner-width* and whose height is equal to the value of *corner-height*. Otherwise, **FRAMEROUNDRECT** is used similarly to **FRAMERECT** with respect to the way the rectangle array is defined. Program 9-11 uses **FRAMEROUNDRECT** to draw two answer buttons on the screen. Type it in and try out the quiz.

Program 9-11. FRAMEROUNDRECT

```
DIM RECT%(11)
READ NQ
RECT%(0)=20:RECT%(1)=20:RECT%(2)=180:RECT%(3)=360
RECT%(4)=150:RECT%(5)=40:RECT%(6)=170:RECT%(7)=120
RECT%(8)=150:RECT%(9)=260:RECT%(10)=170:RECT%(11)=340
C=0
FOR I=1 TO NQ
  CLS
  READ Q$
  CALL MOVETO(25,40)
  PRINT Q$
  CALL MOVETO(RECT%(5)+20,RECT%(6)-6)
  PRINT "True"
  CALL MOVETO(RECT%(9)+20,RECT%(10)-6)
  PRINT "False"
  CALL MOVETO(65,90)
  PRINT "True or False?"
  CALL FRAMERECT(VARPTR(RECT%(0)))
  CALL FRAMEROUNDRECT(VARPTR(RECT%(4)),15,15)
  CALL FRAMEROUNDRECT(VARPTR(RECT%(8)),15,15)
  WHILE MOUSE(0)<1:WEND
  IF MOUSE(1)>RECT%(5) AND MOUSE(1)<RECT%(7) AND MOUSE
(2)>RECT%(4) AND MOUSE(2)<RECT%(6) AND INSTR(Q$,"is")>0 T
HEN C=C+1
  IF MOUSE(1)>RECT%(9) AND MOUSE(1)<RECT%(11) AND MOUS
E(2)>RECT%(8) AND MOUSE(2)<RECT%(10) AND INSTR(Q$,"is")=0
 THEN C=C+1
NEXT I
CALL MOVETO(10,15)
PRINT "Number of questions correct was";C
END
DATA 10
DATA "The formula for sulfuric acid is H2SO4."
DATA "The capital of Saskatchewan was Saskatoon."
DATA "Polar bears can stay submerged for 30 minutes."
DATA "Osmium is denser than uranium."
```

DATA "Fluorine is an elemental gas."
DATA "ROM memory can generally be read faster than RAM."
DATA "COBOL is older than BASIC."
DATA "Bubble sorts are the fastest sorts."
DATA "Shell sorts are used for arranging gambling data."
DATA "This is the last question."

Program 9-11 draws a series of boxes containing a true or false question. The questions are answered by clicking in either the *True* or *False* button. The program also keeps track of how many correct responses are entered.

Figure 9-12. *True and false questions displayed in a dialog box are answered by clicking the mouse in the appropriate button.*
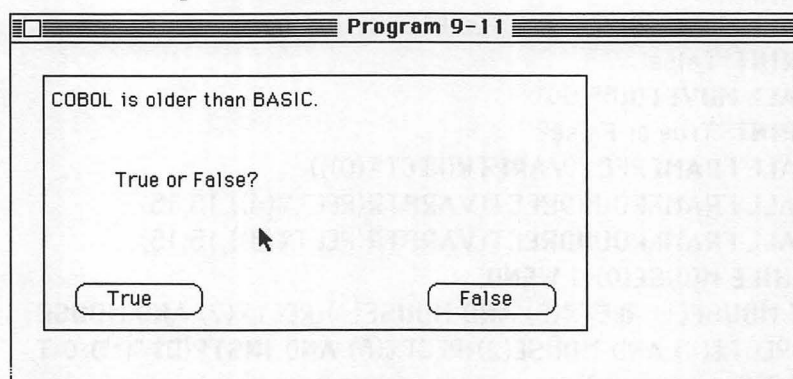
The box is drawn with a rectangle as are the two buttons. Since each rectangle requires four parameters (the top, left, bottom, and right sides of the rectangle), an array of 12 elements for three rectangles is defined in the first line by DIM RECT%(11). The option base of this program is 0, so, the parameters of the first rectangle are located in RECT%(0) to RECT%(3). Similarly, the parameters of the second rectangle, the *True* button, are located in RECT%(4) to RECT%(7), and the parameters of the third rectangle, the *False* button, are located in RECT%(8) to RECT%(11). NQ, the number of questions asked, is read from the first DATA statement in the program. NQ is thus set to 10. Parameters of the three rectangles are assigned values. C represents the number of correct

responses to the true and false questions. Initially, of course, the value of C is set to 0.

The main portion of the program lies with the **FOR-NEXT** loop. The loop counter is the variable I, which equals the question number being asked. It starts at 1 for question number 1, and its ending value is NQ since there are NQ questions. The *Output* window must be cleared before each dialog box is created, and before the current question is read and stored in Q$. The question is then positioned and displayed on the screen by using **MOVETO** and **PRINT**.

The labels for the *True* and *False* buttons are drawn in the same way. However, their positions are 20 pixels to the right of the left edge of the button and 6 pixels above the bottom edge of the button.

The dialog box is drawn by **FRAMERECT**, but the two buttons are created with **FRAMEROUNDRECT**. The corner-width and corner-height are both 15 pixels. After the dialog box is completed, the program waits for the mouse to be clicked in one of the two buttons. When this happens, **MOUSE(0)** returns a value greater than zero at the **WHILE** statement. **MOUSE(1)** and **MOUSE(2)** return the horizontal and vertical positions of the mouse pointer location to determine which of the two buttons (if any) was clicked on. If it was the *True* button, and if the answer was true, the value of C is incremented by one. The second **IF-THEN** statement checks to see if the *False* button was clicked and if the answer was false. Again, the value of C is incremented if this is the case. All other possibilities are considered to be a wrong response. After NQ questions have been answered, execution exits the **FOR-NEXT** loop to the **CALL MOVETO** statement where the pen position is moved to the top of the screen.

The program can be modified to ask different questions by changing the **DATA** statements at the end of the program. The number of **DATA** statements with questions must be represented by the value of the first piece of data. If there are 16 questions, for instance, you'll have to change the first **DATA** statement to **DATA 16**.

This program determines if the question is true or false by looking for the word *is* within the question stored in the string variable Q$. This is done with the **INSTR** command in the two **IF-THEN** statements near the end of the **FOR-NEXT** loop. If the word *is* can be found, the question is considered

true. This technique makes it hard for someone to determine the answers by looking at the data at the end of the program. However, the question must be worded carefully.

Filling Rounded Rectangles

Round-cornered rectangles can be filled with patterns by using the ROM routine **FILLROUNDRECT**:

CALL FILLROUNDRECT(VARPTR(*rectangle array element***),***corner-width,corner-height***,VARPTR(***pattern array element***))**

FILLROUNDRECT draws a round-cornered rectangle filled with a pattern. The round-cornered rectangle is defined just as if you were using **FRAMEROUNDRECT**. The only difference is that a pattern must be specified, as you would with **FILLRECT**. Program 9-12 is a good demonstration of using **FILLROUNDRECT** in BASIC.

Program 9-12. FILLROUNDRECT

```
CLS
DIM RECT%(3),PAT%(7)
FOR I=0 TO 7:READ PAT%(I):NEXT I
FOR I=0 TO 3
   RECT%(0)=I*50+20:RECT%(1)=140:RECT%(2)=RECT%(0)+20:RECT
%(3)=RECT%(1)+100
   CALL FILLROUNDRECT(VARPTR(RECT%(0)),15,15,VARPTR(P
AT%(4)))
   RECT%(0)=RECT%(0)-3:RECT%(1)=RECT%(1)-3:RECT%(2)=RECT%(
0)+20:RECT%(3)=RECT%(1)+100
   CALL FILLROUNDRECT(VARPTR(RECT%(0)),15,15,VARPTR(P
AT%(0)))
   CALL FRAMEROUNDRECT(VARPTR(RECT%(0)),15,15)
   READ MESG$
   CALL MOVETO(RECT%(1)+10,RECT%(2)-6):PRINT MESG$;
NEXT I
END
DATA 0,0,0,0:'white pattern
DATA -1,-1,-1,-1:'black pattern
DATA "These","are","3-D","buttons"
```

Figure 9-13. *These buttons were drawn with FILLROUNDRECT.*
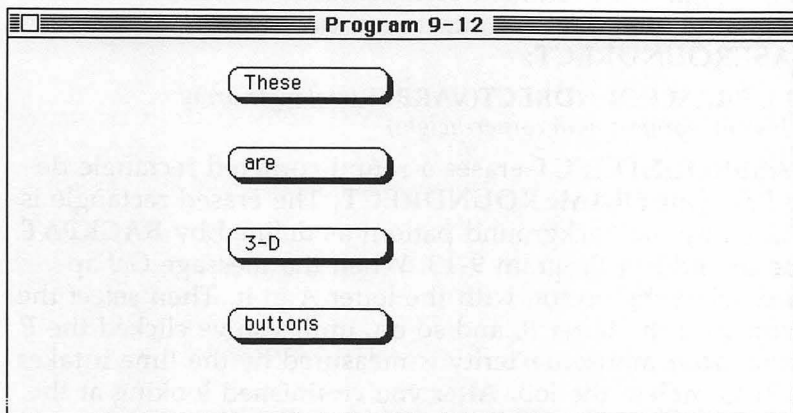
Program 9-12 draws and labels four three-dimensional buttons. The program reads the pattern elements from **DATA** statements, after clearing the *Output* window to white and dimensioning the data arrays (RECT%(3), a rectangle array defining the four borders of a button, and the pattern array PAT%(7), which has eight elements to contain the two patterns. The main part of the program is a **FOR-NEXT** loop. The loop iterates four times to draw the four three-dimensional buttons. The location of the buttons' shadows is calculated in the fifth line and stored in RECT%(3). The first shadow is drawn 20 pixels from the top of the *Output* window and 140 pixels from the left side of the window. Each button is 20 pixels high, 100 pixels wide, and 50 pixels below the previous button. The black shadow of the button is drawn in the next line with **FILLROUNDRECT**. The button itself has its location calculated as 3 pixels above and to the left of its shadow, done in the seventh line. **FILLROUNDRECT** draws the white inside of the button, and the button is completed by having its frame drawn with a call to **FRAMEROUNDRECT**. A label for each button is read from the **DATA** statement and stored in MESG$. **CALL MOVETO** positions the pen location inside the button and prints the label. After four buttons are drawn, program execution ends.

Erasing While Preserving

If a rectangle with rounded corners has to be erased, but the background must be kept intact, you can use ERASEROUNDRECT:

CALL ERASEROUNDRECT(VARPTR(*rectangle array element*),*corner-width,corner-height*)

ERASEROUNDRECT erases a round-cornered rectangle defined as with FRAMEROUNDRECT. The erased rectangle is replaced by the background pattern as defined by BACKPAT. Type in and run Program 9-13. When the message *Go!* appears, select the button with the letter *A* in it. Then select the button with the letter *B*, and so on, until you've clicked the *P* button. Your mouse dexterity is measured by the time it takes you to complete the job. After you've finished looking at the results, press the mouse button and the program ends.

Program 9-13. ERASEROUNDRECT

```
RANDOMIZE TIMER
DEFINT A-Z
DIM RECT(3),PAT(7),LAB$(15)
FOR I=0 TO 7:READ PAT(I):NEXT I
CALL BACKPAT(VARPTR(PAT(4)))
CLS
RECT(0)=10:RECT(1)=110:RECT(2)=RECT(0)+20:RECT(3)=RECT(1)+
270
CALL FILLROUNDRECT(VARPTR(RECT(0)),15,15,VARPTR(PAT(
0)))
CALL FRAMEROUNDRECT(VARPTR(RECT(0)),15,15)
CALL MOVETO(RECT(1)+80,RECT(2)-6):PRINT "Mouse Dexterity"
;
FOR I=0 TO 14

GETJ:
    J=INT(RND(1)*16)
    IF LAB$(J)<>"" THEN GETJ
    LAB$(J)=CHR$(65+I)
NEXT I
I=0:WHILE LAB$(I)<>"":I=I+1:WEND
LAB$(I)="P"
```

```
FOR I=0 TO 3
  FOR J=0 TO 3
    RECT(0)=40*I+40:RECT(1)=90*J+70:RECT(2)=RECT(0)+20:REC
T(3)=RECT(1)+80
    CALL FILLROUNDRECT(VARPTR(RECT(0)),15,15,VARPTR(P
AT(0)))
    CALL FRAMEROUNDRECT(VARPTR(RECT(0)),15,15)
    CALL MOVETO(RECT(1)+30,RECT(2)-6):PRINT LAB$(I*4+J);
  NEXT J
NEXT I
RECT(0)=200:RECT(1)=110:RECT(2)=RECT(0)+20:RECT(3)=RECT(1)
+270
CALL FILLROUNDRECT(VARPTR(RECT(0)),15,15,VARPTR(PAT(
0)))
CALL FRAMEROUNDRECT(VARPTR(RECT(0)),15,15)
CALL MOVETO(RECT(1)+100,RECT(2)-6):PRINT "Go!";
GOTIME!=TIMER
LETTER=65
WHILE LETTER<81

GETBUTTON:
  WHILE MOUSE(0)<1:WEND
  X=MOUSE(1):Y=MOUSE(2)
  IF (X<70) OR (X>150 AND X<160) OR (X>240 AND X<250) OR (X>
330 AND X<340) OR (X>420) THEN GETBUTTON
  IF (Y<40) OR (Y>60 AND Y<80) OR (Y>100 AND Y<120) OR (Y>14
0 AND Y<160) OR (Y>180) THEN GETBUTTON
  J=INT((X-70) / 90):I=INT((Y-40) /40):BUT =I*4+J
  IF LAB$(BUT)<>CHR$(LETTER) THEN GETBUTTON
  RECT(0)=40*I+40:RECT(1)=90*J+70:RECT(2)=RECT(0)+20:RECT(
3)=RECT(1)+80
  CALL ERASEROUNDRECT(VARPTR(RECT(0)),15,15)
  LETTER=LETTER+1
WEND
TTLTIME!=TIMER-GOTIME!
CALL MOVETO(120,214):PRINT "Elapsed time was";TTLTIME!;"se
conds."
WHILE MOUSE(0)<1:WEND
```

```
CALL BACKPAT(VARPTR(PAT(0)))
END
DATA 0,0,0,0
DATA 4420,4420,4420,4420
```

Program 9-13 draws 16 round-cornered rectangles, each labeled with a letter from *A* to *P*. The letters are randomly assigned to the buttons each time the program is run. The program times your response in selecting the buttons in alphabetical order. Before the program ends, it displays your reaction time.

Figure 9-14. *The Output window display during program execution. The object of the game is to press the buttons in alphabetical order in the quickest time possible. The elapsed time will be displayed when the last button is pressed.*
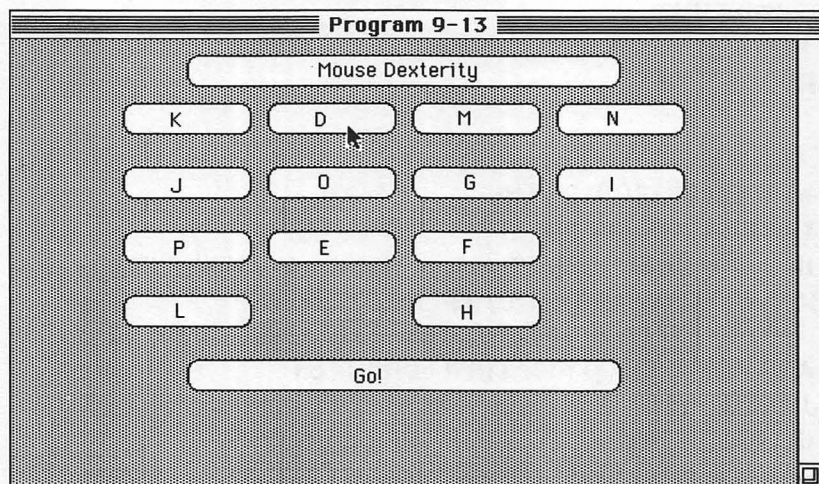
The first line seeds the random number sequence generator used when assigning the labels to the buttons. All variables are designated as integers by **DEFINT** A–Z. The three arrays of this program are RECT(3), the rectangle array used to create the buttons; PAT(7), the pattern array for setting the background of the *Output* window and for drawing the buttons; and LAB$(15), a string array for the 16 button labels.

Data for the pattern array is read from **DATA** statements at the very end of the program, after which **BACKPAT** sets the background pattern to light gray. (When using **ERASE-ROUNDRECT**, **BACKPAT** should first be called to define the background pattern.)

The background actually changes after the execution of the **CLS** command. A title box in the form of a round-cornered rectangle is defined and drawn by several lines which establish the rectangle array, clear the inside of the box with **FILL-ROUNDRECT**, and frame it with **FRAMEROUNDRECT**.

The **FOR-NEXT** loop immediately before the GETJ subroutine generates the labels for 15 random buttons. The value of the loop counter, I, is relative to the value of the ASCII equivalent of the label being assigned. The labels A, B, C, and so on, are assigned sequentially to random buttons. INT(RND(1)*16) picks a random button and stores its number in J. A number of checks are conducted, including determining if button *J* already has a label. If the label for button *J* is a null string, then it's given the current label. Otherwise, the program loops back to GETJ where another button can be picked. The label is assigned LAB$(J)=CHR$(65+I). CHR$ yields the character *A* for the label of button *J* when I is zero (the sixty-fifth character of the ASCII set is the letter *A*). Each of the 25 subsequent uppercase letters has an ASCII value one greater than the previous letter.

The nested **FOR-NEXT** loops draw the buttons. The loop variable, I, represents the row number of the button, while J represents the column number. (Rows are numbered 0 through 3, from top to bottom; columns are labeled 0 through 3, left to right.) Since there are four buttons to a row, the button number is calculated as four times the row number of the button plus the column number of the button. The array elements are defined based on the button's row and column numbers. Finally, **FILLROUNDRECT** fills the inside of the button with white, and **FRAMEROUNDRECT** frames it.

A number of other commands are placed next, including reading the time of the start of the test by **TIMER** and recording it in the single-precision variable GOTIME!. The next button to be selected must have the label which corresponds to an ACSII code equal to the value of the variable LETTER, called the target label. The initial value of LETTER is set to 65.

The main section of the program is a **WHILE-WEND** loop

which repeats until all 16 buttons have been correctly chosen. The first step is to monitor the mouse and wait until its button has been pressed, performed by **WHILE MOUSE(0)<1:WEND**. The position of the mouse pointer is recorded in the variables X and Y; then those locations are verified to insure that the pointer is within a button's boundaries. If not, the mouse click is ignored and program execution goes back to GETBUTTON. Otherwise, a calculation is made to see which button was selected—that value is stored in BUT. If the label of that button (LAB$(BUT)) does not equal the ASCII value of LETTER (the target label), the selection is ignored and the program again retreats to GETBUTTON. On the other hand, if it does match, the rectangle array of the button is calculated and the button is erased with **ERASEROUNDRECT**. LETTER is incremented by one. As long as LETTER doesn't exceed 80 (the ASCII value of the letter *P*), this continues. When all 16 buttons have been selected, the elapsed time is found and stored in TTLTIME!.

Round-Corner Inversion

The ROM routine to invert round-cornered rectangles is called **INVERTROUNDRECT**:

CALL INVERTROUNDRECT(VARPTR(*rectangle array element***),***corner-width,corner-height***)**

INVERTROUNDRECT inverts a defined round-cornered rectangle. Try Program 9-14. Press the buttons displayed at the bottom of the *Output* window to control the rabbit. When you do, the button (a round-cornered rectangle) is inverted. What was black is now white; what was white is now black.

Holding down the mouse button while selecting a button temporarily stops the bunny. Select the *End* button to quit the program.

Program 9-14. INVERTROUNDRECT

```
DEFINT A-Z
DIM RECT(3),PAT(11),BUTTN(3,4),BUNNY(16)
FOR I=0 TO 11:READ PAT(I):NEXT I
CALL BACKPAT(VARPTR(PAT(4)))
CLS
RECT(0)=212:RECT(1)=20:RECT(2)=252:RECT(3)=460
```

```
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(8)))
RECT(0)=210:RECT(1)=18:RECT(2)=250:RECT(3)=458
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(RECT(0)))
RECT(0)=10:RECT(1)=20:RECT(2)=200:RECT(3)=460
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(8)))
RECT(0)=8:RECT(1)=18:RECT(2)=198:RECT(3)=458
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(RECT(0)))
FOR I=0 TO 4
   BUTTN(0,I)=220:BUTTN(1,I)=85*I+28:BUTTN(2,I)=240:BUTTN(3,
I)=BUTTN(1,I)+80
   READ LAB$:CALL MOVETO(BUTTN(1,I)+20,BUTTN(0,I)+14):PRI
NT LAB$;
   CALL FRAMEROUNDRECT(VARPTR(BUTTN(0,I)),15,15)
NEXT I
CALL MOVETO(230,111):PRINT CHR$(217)
GET (230,100)-(244,114),BUNNY
RX=230:RY=100:DX=0:DY=0
GETBUT:
WHILE MOUSE(0)=0:GOSUB MOVEBUNNY:WEND
X=MOUSE(1):Y=MOUSE(2)
BUT= -1:I=0
WHILE (BUT= -1) AND (I<5)
   IF Y>BUTTN(0,I) AND X>BUTTN(1,I) AND Y<BUTTN(2,I) AND X<BU
TTN(3,I) THEN BUT=I
   I=I+1
WEND
IF BUT= -1 THEN GETBUT
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUT)),15,15)
WHILE MOUSE(0)<0:WEND
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUT)),15,15)
ON BUT+1 GOSUB MOVEUP,MOVELEFT,QUIT,MOVERIGHT,MOVEDOW
N
GOTO GETBUT

MOVEUP:
   DY=DY-1:IF DY< -1 THEN DY= -1
```

```
RETURN

MOVELEFT:
 DX=DX-1:IF DX< -1 THEN DX= -1
 RETURN

QUIT:
 CALL BACKPAT(VARPTR(PAT(0)))
 END

MOVERIGHT:
 DX=DX+1:IF DX>1 THEN DX=1
 RETURN

MOVEDOWN:
 DY=DY+1:IF DY>1 THEN DY=1
 RETURN

MOVEBUNNY:
 NX=RX+DX:IF NX<RECT(1)+10 OR NX>RECT(3)-10 THEN DX=0:NX=
RX
 NY=RY+DY:IF NY<RECT(0)+10 OR NY>RECT(2)-10 THEN DY=0:NY=
RY
 PUT (RX,RY),BUNNY:PUT (NX,NY),BUNNY:RX=NX:RY=NY
 RETURN
DATA 0,0,0,0
DATA 21930,21930,21930,21930
DATA -1,-1,-1,-1
DATA "Up","Left","End","Right","Down"
```

Program 9-14 creates a rabbit testing field and a control panel with which to issue instructions. The commands are made by pressing the appropriate buttons. Choosing a sequence of commands makes the rabbit move diagonally. For example, pressing the button labeled *Right*, then the button labeled *Up* causes a stationary rabbit to move up and to the right. The buttons on the left side of the control panel undo the instructions given by the buttons on the right and vice versa. If the rabbit is moving left, for instance, it can be

stopped by pressing the *Right* button. Rabbit testing is terminated by pressing the *End* button.

When the pointer is placed on one of the control buttons and the button is pressed, the control button is inverted as long as the mouse button remains down. Simultaneously, the rabbit stops to listen for its new instructions. When the mouse button is released, the selected control button returns to normal and the rabbit performs its new instructions. The rabbit ignores redundant commands. If it's already moving up, for example, it ignores all subsequent commands to go up.

Figure 9-15. *The rabbit is controlled from the panel by selecting the appropriate direction button. Diagonal motion can be created by selecting a combination of buttons.*
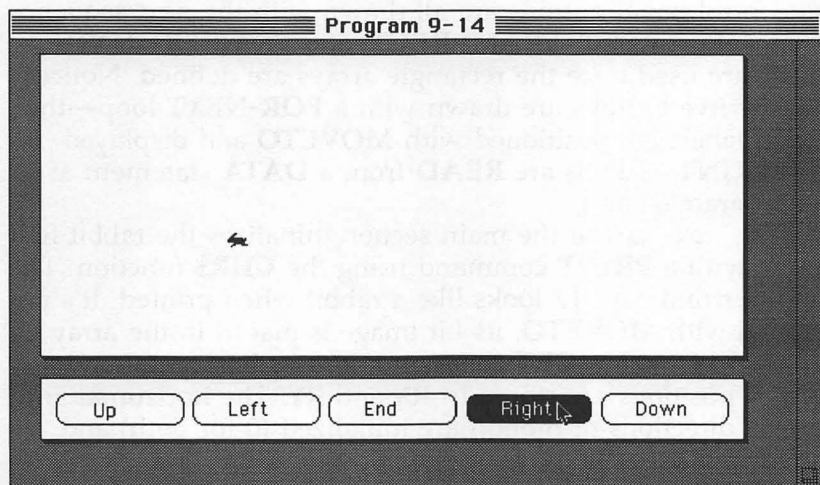
The program is divided into three sections: the code before the MOVEUP routine comprises the main part of the program. This prepares the *Output* window display and monitors user activity. The second major section consists of five subroutines (MOVEUP, MOVELEFT, QUIT, MOVERIGHT, and MOVEDOWN) which execute when a control button is pressed. These routines are executed on an event-driven basis—they're executed only when a control button selection event has occurred. The third section, the MOVEBUNNY subroutine, controls the rabbit's movement.

NINE

Most of the program structure, and in fact many individual lines, should already be easily identifiable. Four arrays are dimensioned: RECT(3), PAT(11), BUTTN(3,4), and BUNNY(16). PAT(11) contains 12 elements to store three patterns. PAT(0) is white, PAT(4) gray, and PAT(8) is black. BUTTN(3,4) is a set of five rectangle arrays used to draw and record the locations of the five control panel buttons. Array element BUTTN(0,*x*) contains the location of the topmost point of button *x*. Similarly, the array elements BUTTN(1,*x*), BUTTN(2,*x*), and BUTTN(3,*x*) represent the left, bottom, and rightmost points of button *x*. BUNNY(16) is a bit image array used to record the bit pattern to draw the rabbit. This pattern is read from **DATA** statements at the end of the program. Note that the background pattern is set to gray by **BACKPAT**.

The control panel (and its shadow), the testing field, and the control panel buttons are all drawn with the next sequence of lines. **FILLRECT**, **FRAMERECT**, and **FRAMEROUND-RECT** are used once the rectangle arrays are defined. Notice that the five buttons are drawn with a **FOR-NEXT** loop—the button labels are positioned with **MOVETO** and displayed with **PRINT**. (Labels are **READ** from a **DATA** statement at the program's end.)

The next part of the main section initializes the rabbit first drawn with a **PRINT** command using the **CHR$** function. The character number 217 looks like a rabbit when printed. It's positioned with **MOVETO**, its bit image is placed in the array BUNNY(16) with a **GET** command, and its horizontal and vertical positions are stored in RX and RY. The horizontal and vertical directions of motion are initialized (0 for both) and stored in DX and DY.

The main loop of the program runs from GETBUT to the first movement subroutine. As long as there's no mouse button activity, the GETBUNNY routine is repeatedly called. This updates the rabbit's position on the screen. After the button is pressed, the pointer's position is recorded in X and Y. Control buttons are checked to see if one was pressed; if BUT is −1, none has been selected. It's initially assumed that no button has been pressed, so I equals zero. The mouse pointer location is compared to the buttons' perimeters, and BUT is updated if necessary.

When a button is pressed, **INVERTROUNDRECT** is called, inverting the button, which stays inverted until the but-

ton is released. As soon as that happens, **INVERT-ROUNDRECT** is called again to turn the button back to normal. Based on the value B + 1, the appropriate subroutine is accessed.

When any of the movement subroutines is called, DY or DX is increased or decreased before the **RETURN** is executed. Values are not allowed to go beyond −1 or 1. The QUIT subroutine simply ends the program.

MOVEBUNNY updates the rabbit's position, calculating the new horizontal position stored in NX and the new vertical position stored in NY. If this new position is going to be outside the testing field perimeter, it's set to the current horizontal or vertical position and motion stops. **PUT** erases the rabbit and draws it at its new position. (Note the default XOR mode of drawing block graphics.)

PAINTROUNDRECT

A similar ROM routine is **PAINTROUNDRECT**:

CALL PAINTROUNDRECT(VARPTR(*rectangle array element***),***corner-width,corner-height***)**

PAINTROUNDRECT draws a round-cornered rectangle filled with the current pen pattern. Program 9-15 is a dramatic example of **PAINTROUNDRECT** in use. It ends as soon as you press the mouse button.

Program 9-15. PAINTROUNDRECT

```
CLS:DEFINT A-Z
DIM TRECT(3),PAT(3),AL(20)
FOR I=0 TO 3:READ PAT(I):NEXT I
TRECT(0)=28:TRECT(1)=78:TRECT(2)=272:TRECT(3)=382
CALL FRAMERECT(VARPTR(TRECT(0)))
O=20
FOR I=32 TO 252 STEP 20
  FOR J=62+O TO 362 STEP 40
    TRECT(0)=I:TRECT(1)=J:TRECT(2)=I+16:TRECT(3)=TRECT(1)+3
6
    IF TRECT(1)<82 THEN TRECT(1)=82
    IF TRECT(3)>378 THEN TRECT(3)=378
    CALL PAINTROUNDRECT(VARPTR(TRECT(0)),5,5)
  NEXT J
```

```
IF O=20 THEN O=0 ELSE O=20
NEXT I
TRECT(0)=70:TRECT(1)=120:TRECT(2)=150:TRECT(3)=180
CALL FILLRECT(VARPTR(TRECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(TRECT(0)))
TRECT(0)=73:TRECT(1)=123:TRECT(2)=147:TRECT(3)=177
CALL PAINTRECT(VARPTR(TRECT(0)))
FOR I=1 TO 1000:NEXT I
FOR I=0 TO 20:READ AL(I):NEXT I
FOR I=1 TO 30
   TRECT(0)=146-I*2:TRECT(1)=125:TRECT(2)=TRECT(0)+2:TRECT
(3)=175
   CALL INVERTRECT(VARPTR(TRECT(0)))
NEXT I
FOR I=1 TO 19
   AL(1)=I
   PUT (140,146-I),AL,PSET
NEXT I
WHILE MOUSE(0)<1:WEND
END
DATA 0,0,0,0
DATA 16,16,-16381,-16381,8196,4104,2448,18018,26644,123
00,5736,4104,4104,2064,1440,1056,2640,4488,24582,-32767,
-32767
```

Three arrays are dimensioned—TRECT(3), a rectangle array for the house; PAT(3), a pattern array to draw the window; and AL(20), which contains the bit image of the alien. The data for PAT(3) is read from a **DATA** statement at the program's end.

A number of the following lines, including the nested **FOR-NEXT** loops, draw the house. The house's frame is created with **FRAMERECT**, while the first brick's offset is set to zero before the statements beginning at **FOR** I=32 **TO** 252 **STEP** 20 draw the bricks. The first **FOR-NEXT** loop records the brick row in I, while the second loop records the brick position for the current row in J. Each row of bricks is offset by 20 pixels horizontally, and each brick is offset by 40 pixels. The array TRECT(3) specifies the size of an entire brick, and the two **IF-THEN** statements chop the brick if it doesn't fit

Figure 9-16. *After the brickwork is drawn with PAINTROUNDRECT, the alien opens the window and rises from his hiding spot.*
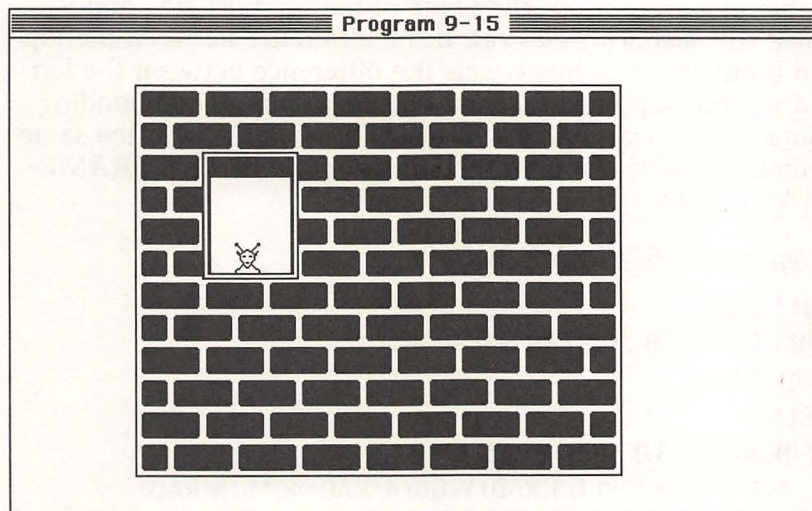
within the frame. Bricks are actually drawn with **PAINT-ROUNDRECT**. Each row of bricks alternates its brick offset between 0 and 20 pixels so that the bricks are staggered.

A hole in the wall for the window is drawn by **FILLRECT** and framed with **FRAMERECT**. **PAINTRECT** creates the window shade. The alien's bit image is read from a **DATA** statement before the **FOR** I=1 **TO** 30 loop animates the opening of the shade. The shades are opened in 30 increments controlled by I. The shade rises by erasing a portion of it from the bottom up with **INVERTRECT**.

Once the shade is up, the alien rises into view. Another **FOR-NEXT** loop (**FOR** I=1 **TO** 19) controls the height of the bit image stored in the second element of the bit image array. The image is drawn starting from one pixel above the window sill; another row of pixels appears with each iteration of the loop. **PUT** with the **PSET** option is used to draw the alien.

Circles and Ellipes

Circles and ellipses can be drawn with the **FRAMEOVAL** ROM routine:

CALL FRAMEOVAL(VARPTR(*oval array element***))**

FRAMEOVAL draws an oval or ellipse which has the position of its top, left, bottom, and rightmost points stored in that order in consecutive integer array elements. The first element of the oval array holds the value of the topmost side and is called the *oval array element*. If the difference between the top and bottommost points equals the difference between the left and rightmost points, the oval will be a circle. The bounding points of the oval will just fit inside a rectangle with the same bounding points. Program 9-16 is a sample of what **FRAME-OVAL** can do.

Program 9-16. FRAMEOVAL

```
DIM OVAL%(3)
FOR HT = 10 TO 20 STEP 5
  WD=30-HT
  CLS
  FOR RAD=0 TO 30 STEP .5
    X=16*RAD+40*COS(RAD):Y=60+4*RAD+40*SIN(RAD)
    OVAL%(0)=Y-HT:OVAL%(1)=X-WD:OVAL%(2)=Y+HT:OVAL%(3)=X+WD
    CALL FRAMEOVAL(VARPTR(OVAL%(0)))
  NEXT RAD
  FOR I=1 TO 1000:NEXT I
NEXT HT
```

You'll see three corkscrewlike shapes drawn with circles and ovals.

First of all, the oval array OVAL%(3) is dimensioned. Note that it's an integer array. The first **FOR-NEXT** loop includes the variable HT, which represents the height of the oval above its center. WD, the width of the oval from its center, is calculated in the following line. The first time the oval is drawn, then, HT will equal 10, and WD will equal 20. The second time both equal 15. The third time the shape is drawn, HT will be 20, while WD will be 10. Notice that when HT and WD are identical, the shape is a circle.

The **FOR RAD=0 TO 30 STEP .5** statement defines the boundary of the inner loop of the program. This controls the number of ovals drawn per corkscrew display. In this case, there are 62 ovals drawn at angles separated by 0.5 radians. Radians are used because the Microsoft BASIC intrinsics **SIN**

Figure 9-17. *The circles are drawn with FRAMEOVAL.*
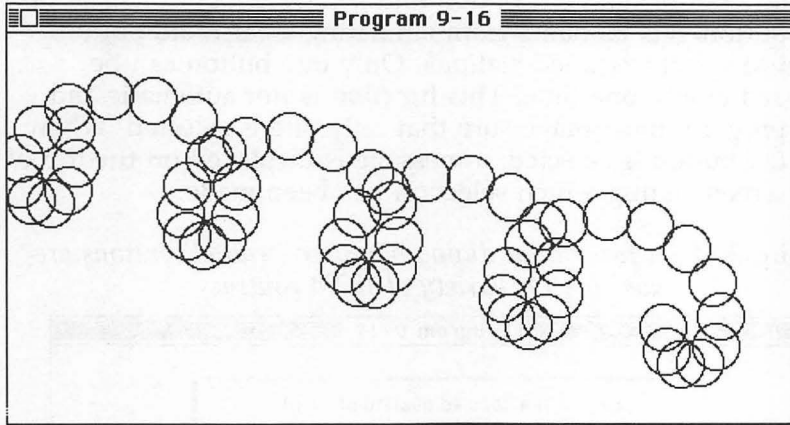
and **COS** interpret their parameters as an angular measure in radians. The center of the next oval is calculated and stored in X (horizontal) and Y (vertical).

The next line is important. It calculates the top, left, bottom, and rightmost points of the oval and stores these values in the respective oval array elements. The topmost point is equal to the vertical position of the oval center minus the height above the center. The bottommost point is equal to the vertical position of the center plus the height above the center.

The respective formulas for the topmost and bottommost points are thus Y−HT and Y+HT. The leftmost point is equal to the horizontal position of the oval minus the width of the oval from the center, while the rightmost point is equal to the horizontal position of the center plus the width from the center. The formulas then for the left and rightmost points are X−WD and X+WD. The ovals are finally drawn with **FRAMEOVAL.**

Filling Ovals

The Macintosh also has a routine for filling in circles and ovals:

CALL FILLOVAL(VARPTR(*oval array element***),VARPTR(***pattern array element***))**

FILLOVAL draws an oval filled with a pattern. The oval array element is defined as when using **FRAMEOVAL**. The pattern within the oval is defined as with **FILLRECT.**

Program 9-17 creates a functional dialog box with three "radio" buttons, an *OK* button and a *CANCEL* button. The radio button gets its name from emulating a car radio button—used to select pretuned stations. Only one button can be pressed at any one time. This function is not automatic and the program must make sure that only one is selected. When the *OK* button is selected, a message is displayed on the top of the screen stating which selection has been made.

Figure 9-18. *A functional dialog box with "radio" buttons created with a variety of ROM routines.*
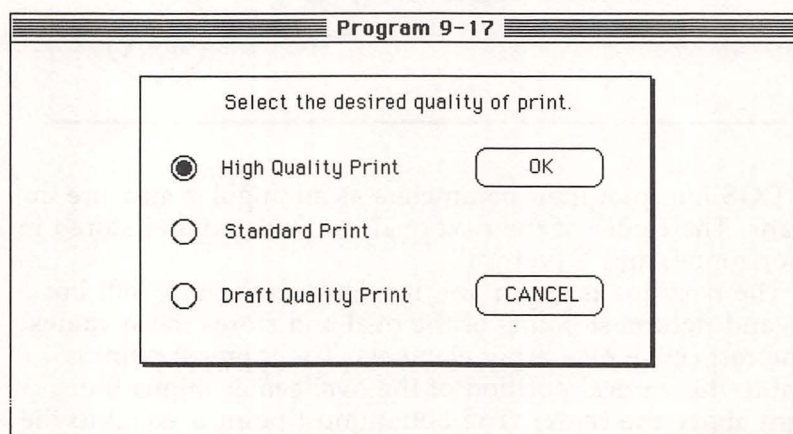
Program 9-17 creates a dialog-style box with several ovals, each representing a button to click to set a desired print quality. Click on the *OK* or *CANCEL* button to exit the program.

Program 9-17. FILLOVAL

```
CLS
DEFINT A-Z
BUTSEL=0
DIM RECT(3),BUTTN(3,2),OK(3),CANCEL(3),LAB$(2),PAT(7),BTNON
(2),OVAL(3)
FOR I=0 TO 7:READ PAT(I):NEXT I
FOR I=0 TO 3:READ RECT(I):NEXT I
FOR I=0 TO 3:RECT(I)=RECT(I)+2:NEXT I
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(4)))
```

```
FOR I=0 TO 3:RECT(I)=RECT(I)-2:NEXT I
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(RECT(0)))
CALL MOVETO(RECT(1)+50,RECT(0)+20):PRINT"Select the desir
ed quality of print.";
FOR I=0 TO 2
    BUTTN(0,I)=40*I+70:BUTTN(1,I)=100:BUTTN(2,I)=BUTTN(0,I)+16
:BUTTN(3,I)=BUTTN(1,I)+16
    READ LAB$(I)
    CALL MOVETO(BUTTN(1,I)+30,BUTTN(2,I)-4):PRINT LAB$(I):
NEXT I
FOR I=0 TO 2:BTNON(I)=0:NEXT I:BTNON(BUTSEL)=1
GOSUB SETBUTTONS
FOR I=0 TO 3:READ OK(I):NEXT I
CALL MOVETO(OK(1)+30,OK(2)-9):PRINT"OK";
CALL FRAMEROUNDRECT(VARPTR(OK(0)),15,15)
FOR I=0 TO 3:READ CANCEL(I):NEXT I
CALL MOVETO(OK(1)+15,CANCEL(2)-9):PRINT"CANCEL";
CALL FRAMEROUNDRECT(VARPTR(CANCEL(0)),15,15)

READBUTTON:
WHILE MOUSE(0)<1:WEND
X=MOUSE(1):Y=MOUSE(2)
IF Y<RECT(0) OR X<RECT(1) OR Y>RECT(2) OR X> RECT(3) THEN B
EEP:GOTO 260
IF X>OK(1) AND X<OK(3) AND Y>OK(0) AND Y<OK(2) THEN CALL M
OVETO(10,10):PRINT LAB$(BUTSEL);" selected.";:GOTO ENDPROG
IF X>CANCEL(1) AND X<CANCEL(3) AND Y>CANCEL(0) AND Y<CANC
EL(2) THEN CALL MOVETO(10,10):PRINT"Printing Canceled.";:G
OTO ENDPROG
FOR I=0 TO 2
    IF X>BUTTN(1,I) AND X<BUTTN(3,I) AND Y>BUTTN(0,I) AND Y<BU
TTN(2,I) THEN TMP=I
NEXT I
IF BUTSEL<>TMP THEN BUTSEL=TMP:GOSUB SETBUTTONS:GOTO R
EADBUTTON
GOTO READBUTTON
```

```
ENDPROG:
END

SETBUTTONS:
FOR I=0 TO 2
   CALL FILLOVAL(VARPTR(BUTTN(0,I)),VARPTR(PAT(0)))
   CALL FRAMEOVAL(VARPTR(BUTTN(0,I)))
NEXT I
OVAL(0)=BUTTN(0,BUTSEL)+3:OVAL(1)=BUTTN(1,BUTSEL)+3:OVAL
(2)=BUTTN(2,BUTSEL)-3:OVAL(3)=BUTTN(3,BUTSEL)-3
CALL FILLOVAL(VARPTR(OVAL(0)),VARPTR(PAT(4)))
RETURN
DATA 0,0,0,0:'white
DATA -1,-1,-1,-1:'black
DATA 20,80,200,400
DATA "High Quality Print","Standard Print","Draft Quality Print"
DATA 65,290,90,370
DATA 145,290,170,370
```

Several arrays are dimensioned at the start of the program, but the two most important are BUTTN(3,2) and PAT(7). BUTTN(3,2) contains the locations of the three radio buttons. PAT(7) holds the two patterns, black and white. Other arrays take care of the box outline rectangle, the two round-cornered rectangles for the *OK* and *CANCEL* buttons, and the radio button labels. You've already seen a number of examples of these kinds of arrays.

The element BUTTN(0,x) contains the topmost point of button x. BUTTN(1,x), BUTTN(2,x), and BUTTN(3,x) represent the locations of the left, bottom, and rightmost points of button x, respectively. The array BTNON(2) records the status of the radio buttons, and an oval array OVAL(3) is defined for filling the appropriate radio button.

After creating the dialog box and its shadow with **FILL-RECT** and **FRAMERECT**, and printing a message in the top part of the box, the program defines the radio buttons and their labels. **FOR** I=0 **TO** 2 begins this loop. The elements of BUTTN are specified, and labels are read into LAB$(2). BTNON(2) is initialized. Each element is set to zero except for the corresponding selected radio button. **GOSUB** SETBUTTONS calls

a subroutine which updates the radio buttons by redrawing each and filling it with a black dot. The **FOR-NEXT** loop within this subroutine draws each of the three radio buttons with **FILLOVAL** and **FRAMEOVAL**. First, the button is erased and then redrawn. Note that the elements of OVAL(3) are all defined as six pixels smaller in diameter than a radio button (BUTTN(3)). The black center of the selected radio button is drawn with **FILLOVAL** before the subroutine returns to the main section of the program.

At that point the program draws and labels the *OK* and *CANCEL* buttons with **FRAMEROUNDRECT**.

The main loop of the program begins with the **WHILE MOUSE**(0)<1:**WEND** statement. This line executes repeatedly until the mouse button is pressed. X and Y hold the pointer location. A series of location checks are conducted to insure that the pointer is within the box and, if so, within the *OK* or *CANCEL* button. If the pointer is outside the dialog box boundaries, **BEEP** sounds an audio complaint. Failing these tests indicates that the pointer was somewhere else inside the dialog box.

FOR I=0 **TO** 2 and the **IF-THEN** statement immediately following check to see if the pointer location coincides with any of the radio buttons. If radio button *I* was selected, I is stored in TMP. TMP is compared with BUTSEL—if they differ, a new button was chosen and SETBUTTONS is again called to redraw the radio buttons.

Erasing Circles

The routine to erase circles and ellipses is called **ERASEOVAL**:

CALL ERASEOVAL(VARPTR(*oval array element***))**

ERASEOVAL erases a circle or an ellipse, shapes generally called ovals. The oval array element is defined as when using **FRAMEOVAL**. The erased oval is replaced by the background pattern, as defined by **BACKPAT**. Program 9-18 presents another hand and eye coordination game, this time using **ERASEOVAL**. When the message *Go!* appears, select the circle with the smallest diameter. It then disappears. Continue to select the circle with the next smallest diameter until you've eliminated all the circles. Your mouse dexterity time is measured and displayed. After you've had a chance to look at the results, press the mouse button again to end the program.

Program 9-18. ERASEOVAL

```
RANDOMIZE TIMER
DEFINT A-Z
DIM RECT(3),OVAL(3,8),PAT(7),RD(8)
FOR I=0 TO 7:READ PAT(I):NEXT I
CALL BACKPAT(VARPTR(PAT(4)))
CLS
RECT(0)=10:RECT(1)=110:RECT(2)=RECT(0)+20:RECT(3)=RECT(1)+
270
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(RECT(0)))
CALL MOVETO(RECT(1)+60,RECT(2)-6):PRINT"Mouse - Eye Dexte
rity";
FOR I=0 TO 7

GETJ:
  J=INT(RND(1)*9)
  IF RD(J)<>0 THEN GETJ
  RD(J)=2*I+10
NEXT I
I=0:WHILE RD(I)>0:I=I+1:WEND
RD(I)=26
FOR I=0 TO 2
  FOR J=0 TO 2
    OVAL(0,I*3+J)=60*I+70-RD(I*3+J):OVAL(1,I*3+J)=100*J+14
0-RD(I*3+J):OVAL(2,I*3+J)=60*I+70+RD(I*3+J):OVAL(3,I*3+J)=1
00*J+140+RD(I*3+J)
    CALL FILLOVAL(VARPTR(OVAL(0,I*3+J)),VARPTR(PAT(0))
)
    CALL FRAMEOVAL(VARPTR(OVAL(0,I*3+J)))
  NEXT J
NEXT I
RECT(0)=230:RECT(1)=110:RECT(2)=RECT(0)+20:RECT(3)=RECT(1)
+270
CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(0)))
CALL FRAMERECT(VARPTR(RECT(0)))
CALL MOVETO(RECT(1)+100,RECT(2)-6):PRINT"Go!";
GOTIME!=TIMER
```

```
RADIUS=10
WHILE RADIUS<27

GETBUTTON:
  WHILE MOUSE(0)<1:WEND
  X=MOUSE(1):Y=MOUSE(2)
  OV=-1:I=0
  WHILE OV<0 AND I<9
    IF Y>OVAL(0,I) AND X>OVAL(1,I) AND Y<OVAL(2,I) AND X<OVA
L(3,I) THEN OV=I
    I=I+1
  WEND
  IF OV <0 THEN GETBUTTON
  IF RD(OV)<>RADIUS THEN GETBUTTON
  CALL ERASEOVAL(VARPTR(OVAL(0,OV)))
  RADIUS=RADIUS+2
WEND
TTLTIME!=TIMER-GOTIME!
CALL MOVETO(120,244):PRINT "Elapsed time was";TTLTIME!;"se
conds."
WHILE MOUSE(0)<1:WEND
CALL BACKPAT(VARPTR(PAT(0)))
END
DATA 0,0,0,0
DATA 4420,4420,4420,4420
```

Program 9-18 draws a set of nine circles; each is randomly assigned different diameters each time the program is run. The program times your response in selecting the circles in order of increasing size.

The first line seeds the random number sequence generator used when assigning the diameters to the circles. Of the four arrays, OVAL(3,8) is the most important. It's a set of nine oval arrays used to draw and record the locations of the nine circles. The oval array element OVAL(0,x) contains the location of the topmost point of the circle x. Similarly, the array elements OVAL(1,x), OVAL(2,x), and OVAL(3,x) represent the left, bottom, and rightmost points of the circle x. Circle diameters, however, are stored in RD(8). Note also the PAT(7) array,

Figure 9-19. *This game tests your mouse-eye coordination as it calculates the time it takes you to click in each circle in order of increasing size. The circles are erased with ERASEOVAL.*
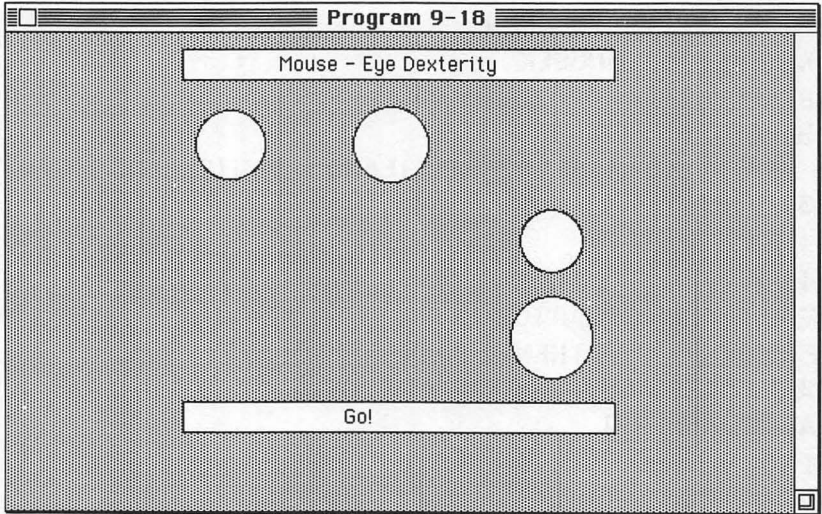
which contains two patterns.

A rectangular background box is defined and then drawn by **FILLRECT** and **FRAMERECT**, with the background set by **BACKPAT**.

Starting at the GETJ label, the program generates the eight random circle diameters. After picking a random circle (stored in J) and determining whether circle J has already been given a diameter, the statement RD(J)=2*I+10 assigns diameters to the various circles. Since I is incremented through a **FOR-NEXT** loop, the circles are of increasing diameter. The ninth circle is given its diameter by the line I=0:**WHILE** RD(I)>0:I=I+1:**WEND**, which keeps incrementing I until the circle with a diameter of zero is found. RD(I)=26 gives this circle its diameter.

The nested **FOR-NEXT** loops draw the circles. I represents the circle row number, while J represents the column number. (Row 0 is on top, column 0 is on the left.) Oval arrays are defined in a series of four lines, filled with white (so that they show against the gray background) with **FILLOVAL** and framed with **FRAMEOVAL**. A sequence of lines create

and label the message box at the bottom of the screen.

TIMER reads the time of the start of the test and records it in the single-precision variable GOTIME!. The next circle to be selected *must* have a diameter which corresponds to the value of RADIUS, the *target diameter*. The initial value of RADIUS is set to 10, the smallest circle.

The main section of the program is a **WHILE-WEND** loop which repeats until all nine circles have been correctly chosen. Its first step is to monitor the mouse and wait until the button is pressed. As in other mouse-driven programs, the horizontal and vertical positions are recorded in X and Y. The nested **WHILE-WEND** loop verifies the location of the pointer and calculates which circle, if any, has been chosen. OV contains two possible values, -1 or the selected circle number. Initially, OV is -1, which indicates that no circle has been chosen. X and Y are compared with the oval array elements of each circle until either the last circle is checked or the circle encompassing the coordinates (X,Y) is found. If the pointer is not within the perimeter of any circle, the click is ignored. If a match between RADIUS and the target button is found, **ERASEOVAL** erases the circle. The target diameter value is incremented by RADIUS=RADIUS+2.

Once all nine circles have been selected and erased, the time is displayed at the bottom of the screen.

Inverted Circles

The ROM routine to invert circles and ellipses is **INVERTOVAL**:

CALL INVERTOVAL(VARPTR(*oval array element*))

INVERTOVAL inverts a circle or an ellipse defined in the same way as ovals are with **FRAMEOVAL**. Program 9-19 shows you a simple application of **INVERTOVAL**. Move the mouse pointer to any position in the *Output* window and click. Overlap patterns to form new patterns. When you wish to quit, select *Stop* from the *Run* menu.

Program 9-19. INVERTOVAL

```
CLS
DEFINT A-Z
```

```
DIM OVAL(3)

LOOP:
WHILE MOUSE(0)<1:WEND
OVAL(0)=MOUSE(2)-53:OVAL(1)=MOUSE(1)-53:OVAL(2)=MOUSE(
2)+53:OVAL(3)=MOUSE(1)+53
FOR I=1 TO 10
  CALL INVERTOVAL(VARPTR(OVAL(0)))
  OVAL(0)=OVAL(0)+5:OVAL(1)=OVAL(1)+5:OVAL(2)=OVAL(2)-5:O
VAL(3)=OVAL(3)-5
NEXT I
GOTO LOOP
```

Program 9-19 draws a set of inverted circles with different diameters, all centered at the pointer's position when the button is clicked.

Figure 9-20. *INVERTOVAL draws the graphics in this interactive demo.*
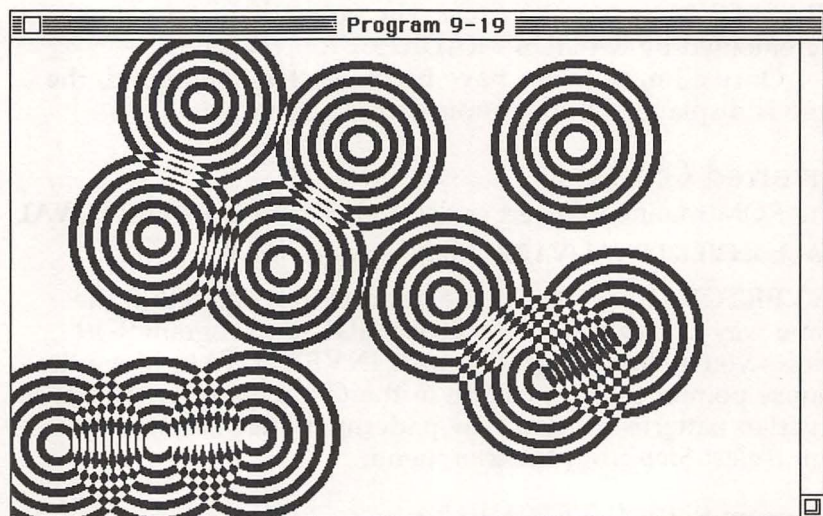
LOOP, the main part of the program, waits for the mouse button to be pressed, then stores the locations (modified somewhat) of the pointer's horizontal and vertical locations in the array OVAL(3). After that, a simple **FOR-NEXT** loop creates

an inverted circle with **INVERTOVAL**. Ten concentric circles, each ten pixels smaller in diameter than the last, are drawn, one atop another. All use the same center point. Black and white alternate, since inverting white gives black and vice versa. This creates the appearance of a bull's-eye, even though it was built by ten solidly filled ovals.

Changing the values +5 and −5 in the last two OVAL(0) statements can alter the appearance of the final product. Decreasing them, for instance, makes the circles' edges narrower.

Painting Ovals and Circles

Ovals are painted with the ROM routine **PAINTOVAL**:

CALL PAINTOVAL(VARPTR(*oval array element*))

PAINTOVAL draws a defined oval, but fills it with the current pen pattern. Program 9-20 is a good example.

Program 9-20. PAINTOVAL

```
CLS:DEFINT A-Z
DIM TRECT(3),CAN(69),CANMSK(69),PLUG(23),PLUGMSK(19),PAT(
7),LIQ(61)
FOR I=0 TO 5:READ CAN(I):NEXT I:FOR I=6 TO 65 STEP 2:CAN(I)
= -16384:CAN(I+1)=768:NEXT I:FOR I=66 TO 69:READ CAN(I):NE
XT I
FOR I=0 TO 5:READ CANMSK(I):NEXT I:FOR I=6 TO 65 STEP 2:CA
NMSK(I)= -4096:CANMSK(I+1)=960:NEXT I:FOR I=66 TO 69:READ
CANMSK(I):NEXT I
FOR I=0 TO 23:READ PLUG(I):NEXT I
FOR I=0 TO 19:READ PLUGMSK(I):NEXT I
FOR I=0 TO 7:READ PAT(I):NEXT I
LIQ(0)=20:LIQ(1)=30:FOR I=2 TO 58 STEP 4:LIQ(I)=-30584:LIQ(I+
1)=-32768!:LIQ(I+2)=8738:LIQ(I+3)=8192:NEXT I
TRECT(0)=150:TRECT(1)=0:TRECT(2)=153:TRECT(3)=489
CALL PAINTRECT(VARPTR(TRECT(0)))
TRECT(0)=56:TRECT(1)=0:TRECT(2)=60:TRECT(3)=489
CALL PAINTRECT(VARPTR(TRECT(0)))
TRECT(0)=60:TRECT(1)=10:TRECT(2)=149:TRECT(3)=34
CALL PAINTRECT(VARPTR(TRECT(0)))
PUT (134,60),PLUG:PUT (284,60),PLUG
FOR I=0 TO 1000:NEXT I
```

```
FOR I=10 TO 488 STEP 20
  TRECT(0)=154:TRECT(1)=I:TRECT(2)=159:TRECT(3)=I+5
  CALL PAINTOVAL(VARPTR(TRECT(0)))
NEXT I
SOUND 120,8,255
FOR I=1 TO 34
  CAN(1)=I
  PUT (10,149-I),CAN,PSET
NEXT I
FOR I=114 TO 100 STEP -1:LINE (10,I)- STEP (24,0),30:NEXT I
FOR I=0 TO 1000:NEXT I
FOR I=10 TO 456 STEP 2
  PUT (I,115),CANMSK
  IF I=136 THEN GOSUB FILLCAN
  IF I=286 THEN GOSUB PLUGCAN
  FOR J=0 TO 8:NEXT J
NEXT I
WHILE MOUSE(0)<1:WEND
END


FILLCAN:
SOUND 80,3,255
FOR K=63 TO 108:PUT (134,K),PLUGMSK:NEXT K
FOR T=0 TO 1000:NEXT T
CALL PENPAT(VARPTR(PAT(4)))
FOR K=1 TO 10
  TRECT(0)=117:TRECT(1)=150-K:TRECT(2)=117+3*K:TRECT(3)=1
50+K
  CALL PAINTOVAL(VARPTR(TRECT(0)))
NEXT K
CALL PENPAT(VARPTR(PAT(4)))
FOR K=500 TO 720 STEP 20:SOUND k,1,255:NEXT k
FOR K=1 TO 30:LIQ(1)=K:PUT (140,147-K),LIQ,PSET:NEXT K
FOR T=0 TO 1000:NEXT T
SOUND 70,6,255
FOR K=108 TO 63 STEP -1:PUT (134,K),PLUGMSK:NEXT K
FOR K=6 TO 62 STEP 4:CANMSK(K)=-9558:CANMSK(K+1)=-21568
:CANMSK(K+2)=-1366: CANMSK(K+3)=-22080:NEXT K
```

```
CALL PENPAT(VARPTR(PAT(0)))
RETURN

PLUGCAN:
SOUND 80,8,255
SOUND 1000,1,255
SOUND 70,6,255
FOR K=63 TO 108:PUT (284,K),PLUGMSK:NEXT K
FOR T=0 TO 1000:NEXT T
FOR K=108 TO 104 STEP -1:PUT (284,K),PLUGMSK:NEXT K
TRECT(0)=115:TRECT(1)=296:TRECT(2)=117:TRECT(3)=306
CALL PAINTRECT(VARPTR(TRECT(0)))
CANMSK(2)=-16384:CANMSK(3)=192:CANMSK(4)=-16384:CANMSK
(5)=192
FOR K=103 TO 63 STEP -1:PUT (284,K),PLUGMSK:NEXT K
RETURN


DATA 24,34,-61,-256,-61,-256,-1,-256,-1,-256
DATA 26,34,-16333,192,-16333,192,-16384,192,-16384,192
DATA 32,11,-1,-1,-1,-1,-1,-1,4095,-16,255,-256,15,-4096,7,-
8192,3,-16384,3,-16384,3,-16384,3,-16384
DATA 32,9,-4096,15,3840,240,240,3840,8,4096,4,8192,0,0,0,0
,0,0,3,-16384
DATA -1,-1,-1,-1,4420,4420,4420,4420
```

Program 9-20 demonstrates three animation techniques with a three-stage production line. The first, located at the left edge of the screen, produces a container from a dispenser which rises up out of the can's path. The container moves along a conveyor belt until it reaches the second station where it's filled from an overhead injector. The container moves to the third station where an overhead capper lowers a plug. After the capper moves out of the way, the container slides to the right.

Seven arrays are dimensioned for this program, ranging from TRECT(3) for the rectangular array (conveyor belt and container dispenser) to PAT(7), which is the pattern array for patterns to fill the container. Most of the arrays' names are self-explanatory.

Figure 9-21. *The container is filled with PAINTOVAL.*
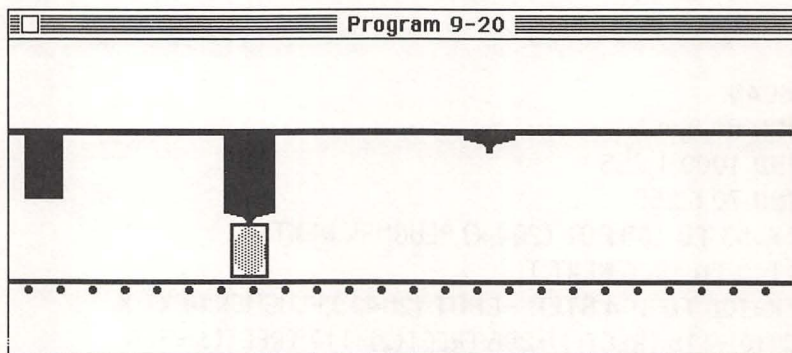
The data for these arrays is initialized, calculated, or read from **DATA** statements at the end of the program. All this is taken care of by a collection of **FOR-NEXT** statements. **PAINTRECT** draws the conveyor belt, overhead injector support, and the container dispenser. (These shapes are first defined with the TRECT array, then drawn.) The two injectors are drawn by **PUT**. After a pause, another **FOR-NEXT** loop creates the 24 rollers beneath the conveyor with **PAINTOVAL**.

The first station's animation begins with the **FOR I=1 TO** 34 statement. Using **PUT** with the **PSET** option, the bit image of the can dispenser is drawn starting 2 pixels above the conveyer belt, then drawn 1 pixel higher each time through the loop. Another **FOR-NEXT** loop raises the dispenser 14 more pixels by erasing it upward from the bottom.

The container's movement is done in the loop which begins with **FOR I=10 TO 456 STEP** 2. The container is animated by **PUT**, which draws a bit image mask over its current position, simultaneously drawing its leading edge and erasing its trailing edge. Checks are conducted to see if the container is at station 2 or 3 (and the appropriate subroutine called).

When station 1 is reached, the FILLCAN subroutine executes. The liquid injector is lowered by drawing its leading edge, stored in PLUGMSK(19), with **PUT** until it mates with the container. The pen pattern for the liquid is made with **PENPAT**, before a growing blob in the container is drawn with **PAINTOVAL**. When the blob hits the bottom of the container, the second stage of animating the filling process is performed by several **FOR-NEXT** loops which raise the injector

with **PUT**, using a mask that erases its trailing edge. The bit image mask of the container is changed to account for the liquid and the pen pattern is reset before the subroutine returns execution to the main section of the program.

The PLUGCAN subroutine animates the capper. In many ways, it's similar to the FILLCAN routine. A bit mask is used again as **PUT** and **PAINTRECT** are called.

Arcs

When you need to draw just part of an oval or a circle, in other words, an arc, you can use the ROM routine **FRAMEARC**:

CALL FRAMEARC(VARPTR(*oval array element***),***start angle,arc angle***)**

FRAMEARC draws an arc from an oval or ellipse. The oval has the position of its top, left, bottom, and rightmost points stored in this order in consecutive elements of an integer array called the oval array. The first element, which contains the value of the location of the topmost point of the oval, is the *oval array element*. The oval will look like a circle when the difference between the top and bottommost points equals the difference between the left and rightmost points.

Since an arc is only part of an oval, the *start angle* and *arc angle* are required to specify which part of the oval to draw. These are angles measured in degrees, with 0 degrees as the topmost point, or the twelve o'clock position. Angles increase in size clockwise. The *start angle* defines the starting angle at which to draw the arc, while the *arc angle* determines how many degrees wide to make the arc. A complete oval has 360 degrees of arc. Program 9-21 uses **FRAMEARC**, but also demonstrates some animation techniques. Press keys 1–9 to control the speed of the graphics. Click the mouse button to end the demo.

Program 9-21. FRAMEARC

```
CLS
DIM OVAL%(3),RECT%(3),PIC1%(310),PIC2%(310),PIC3%(310)
RECT%(0)=20:RECT%(1)=0:RECT%(2)=32:RECT%(3)=200
CALL FRAMERECT(VARPTR(RECT%(0)))
RECT%(0)=16:RECT%(1)=178:RECT%(2)=21:RECT%(3)=190
CALL FRAMERECT(VARPTR(RECT%(0)))
```

```
RECT%(1)=181:RECT%(3)=187
CALL FRAMERECT(VARPTR(RECT%(0)))
RECT%(0)=12:RECT%(1)=181:RECT%(2)=17:RECT%(3)=187
CALL FRAMERECT(VARPTR(RECT%(0)))
CALL MOVETO(186,12):CALL LINETO(181,15)
CALL MOVETO(181,32):CALL LINE(0,15)
CALL MOVETO(186,32):CALL LINE(0,11)
OVAL%(0)=47:OVAL%(1)=175:OVAL%(2)=59:OVAL%(3)=187
CALL FRAMEARC(VARPTR(OVAL%(0)),0,270)
OVAL%(0)=41:OVAL%(1)=169:OVAL%(2)=65:OVAL%(3)=193
CALL FRAMEARC(VARPTR(OVAL%(0)),30,240)
CALL MOVETO(169,53):CALL LINE(6,0)
CALL MOVETO(181,57):CALL LINE(0,20)
D=1:GOSUB DRAWSPRING:GET (169,77)-(193,230),PIC1%:PUT (1
69,77),PIC1%
D=2:GOSUB DRAWSPRING:GET (169,77)-(193,230),PIC2%:PUT (1
69,77),PIC2%
D=3:GOSUB DRAWSPRING:GET (169,77)-(193,230),PIC3%:PUT (1
69,77),PIC3%
P=70

DRAWPIC:
PUT (169,77),PIC1%:FOR I=1 TO P:NEXT I:PUT (169,77),PIC1%
PUT (169,77),PIC2%:FOR I=1 TO P:NEXT I:PUT (169,77),PIC2%
PUT (169,77),PIC3%:FOR I=1 TO P:NEXT I:PUT (169,77),PIC3%
K$=INKEY$:IF K$<>"" THEN IF (K$>="1") AND (K$<="9") THEN P=V
AL(K$)*10
IF MOUSE(0)>0 THEN ENDPROG
GOTO DRAWPIC

ENDPROG:
PUT (169,77),PIC3%
END

DRAWSPRING:
CALL MOVETO(181,77):CALL LINE(6,D)
FOR I=1 TO 10
   CALL LINE(-12,2*D):CALL LINE(12,2*D)
```

```
NEXT I
CALL LINE(-6,D):CALL LINE(0,6)
OVAL%(0)=83+42*D:OVAL%(1)=172:OVAL%(2)=OVAL%(0)+16:OVAL
%(3)=190
CALL FRAMEARC(VARPTR(OVAL%(0)),270,180)
CALL MOVETO(OVAL%(1),OVAL%(0)+8)
CALL LINE(8,12):CALL LINE(8,-12)
RETURN
```

Program 9-21 goes one step beyond being a simple spirographic demo using **FRAMEARC**. A plumb bob is suspended from a spring, which in turn is fastened to a hook bolted into a plank lying across the top of the screen. Once the screen is drawn, the plumb bob begins to oscillate up and down by using some straightforward block graphic animation techniques. To add some user input, the speed of the oscillation can be controlled with the keyboard. Press the 1 key for the fastest oscillation rate and the 9 key for the slowest rate. The program monitors mouse activity so that when the mouse button is clicked, the program stops.

Figure 9-22. *The oscillating spring is drawn with FRAMEARC.*
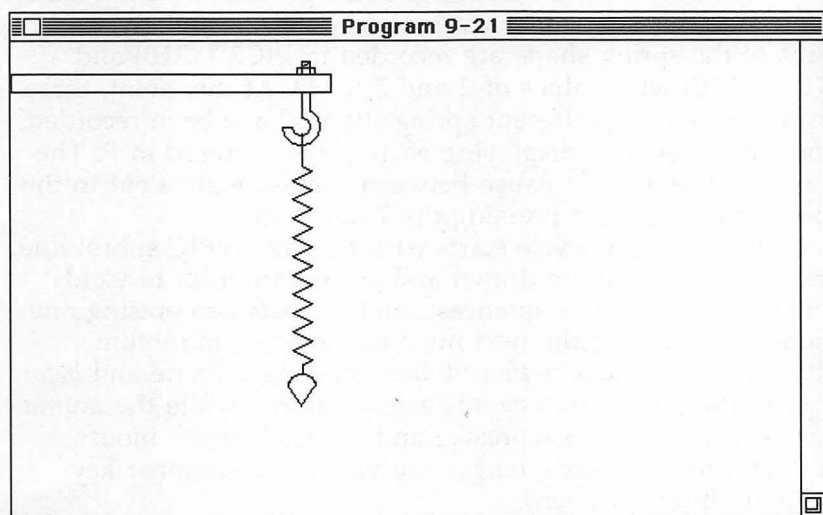
The necessary arrays are declared in the second line. OVAL%(3) stores the ovals used when drawing arcs with

FRAMEARC. RECT%(3) stores rectangles used when drawing with **FRAMERECT**. Three pictures of the plumb bob in different positions are recorded and played back in an endless sequence to create the effect of animation. Each of these pictures requires an array—PIC1%(310), PIC2%(310), and PIC3%(310). Since each picture requires about 620 bytes of storage (see **GET**), the integer arrays are dimensioned to contain 310 elements. Remember, integer arrays provide 2 bytes of storage per array element.

FRAMERECT draws a number of the initial graphics, including the plank, the nut, and the hook shaft. The hook itself is defined and drawn with **FRAMEARC** as a pair of parellel arcs. The innermost arc extends 270 degrees from the 0-degree mark, while the outermost arc extends 240 degrees from the 30-degree mark.

The spring is created in the DRAWSPRING subroutine through a series of **LINE** calls. Each coil is stretched a distance stored in D. The top of the plumb bob is drawn by **FRAME-ARC**, with start and arc angles of 270 and 180, respectively. The rest of the plumb bob is completed by two **LINE** calls.

Once the spring and plumb bob are drawn, the routine returns to the main program where the initial shape is recorded in PIC1%(310) with a **GET** command and erased with a **PUT**. (Note that this shape has a value of 1 for D.) Similarly, pictures of the spring shape are recorded in PIC2%(310) and PIC3%(310) with values of 2 and 3 for D. At this point, three frames containing different spring shapes have been recorded. The pause between displaying each frame is stored in P. The starting value for the pause between frames, equivalent to the speed resulting from pressing the 7 key, is set.

The animation cycle starts with the DRAWPIC subroutine. Frames 1, 2, and 3 are drawn and erased. In order to yield smoother animation sequences, the time between erasing one frame and drawing the next must be kept to a minimum. Therefore, the pause is placed *after* drawing a frame and *before* erasing the frame. Two events are monitored while the animation is in progress—keypresses and the click of the mouse. Hitting a number key changes the value of P (number key value multiplied by ten).

Because this program monitors events and acts only when one occurs, it's called an *event-driven* program. This is the type of program which must be used for arcade games and other realtime programs.

N I N E

Filling Arcs

The Macintosh also has a ROM routine which fills in sections of a circle or ellipse with a user-definable pattern:

CALL FILLARC(VARPTR(*oval array element***),start angle,arc angle,VARPTR(***pattern array element***))**

FILLARC draws an arc from an oval or ellipse and fills it with a pattern. The oval array element, start angle, and arc angle are defined as with **FRAMEARC**, while the pattern used within the arc is defined as with **FILLRECT**. Program 9-22 is a quick sample.

Program 9-22. FILLARC

```
CLS
DEFINT A-Z
DIM OVAL(3),PAT(15)
FOR I=0 TO 15:READ PAT(I):NEXT I
OVAL(0)=30:OVAL(1)=140:OVAL(2)=230:OVAL(3)=340
FOR I=0 TO 3
  READ STARTANGLE,ARCANGLE
  CALL FILLARC(VARPTR(OVAL(0)),STARTANGLE,ARCANGLE,VA
RPTR(PAT(I*4)))
NEXT I
CALL FRAMEOVAL(VARPTR(OVAL(0)))
CALL MOVETO(200,10):PRINT"A Pie Chart";
END
DATA -1,-1,-1,-1
DATA 4420,4420,4420,4420
DATA 21930,21930,21930,21930
DATA 888,888,888,888
DATA 0,140
DATA 140,100
DATA 240,80
DATA 320,40
```

Program 9-22 uses two arrays—OVAL(3), an oval array for the pie chart sections of the pie chart, and PAT(15), a pattern array for filling in the pie sections. The patterns are read from **DATA** statements at the program's end. The entire pie must be predefined as an oval array so that sections of it can

Figure 9-23. *This pie chart is drawn with FILLARC.*
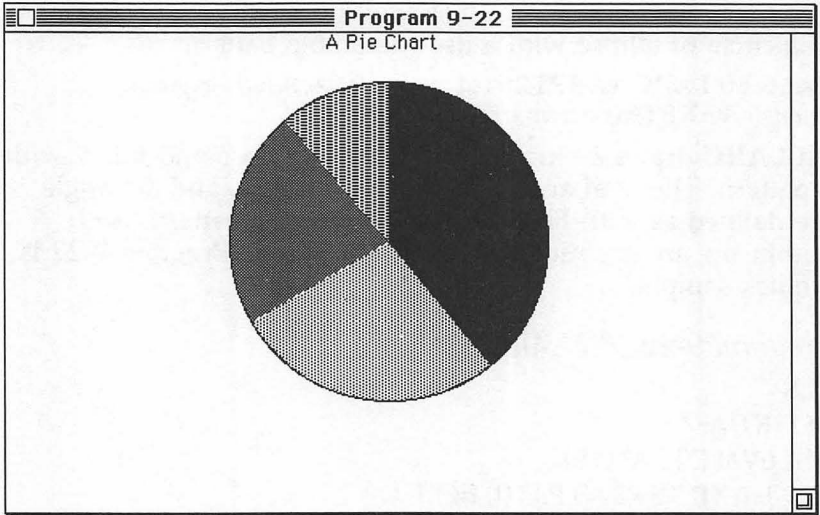
be drawn. The **FOR-NEXT** loop draws the pie sections themselves, using I to generate four pie sections. Note that the **READ** STARTANGLE,ARCANGLE statement reads the values from **DATA** statements. **FILLARC** is used to draw the four sections.

Since each pattern requires four array elements, the pattern used to fill pie section I starts at element I*4 of the pattern array PAT(15). The entire pie is outlined by a call to **FRAMEOVAL**.

Erasing Arcs

To erase sections of circles or ellipses, use **ERASEARC**:

CALL ERASEARC(VARPTR(*oval array element***),***start angle,arc angle***)**

ERASEARC erases a pie section from a circle or ellipse and replaces it with the current background pattern. The oval array element, start angle, and arc angle are defined just as with **FRAMEARC**. Type in and run Program 9-23 for a demonstration of **ERASEARC**'s use. The optical illusion takes at least a minute to accomplish.

Program 9-23. ERASEARC

```
CLS
DEFINT A-Z
DIM OVAL(3),PAT(11)
FOR I=0 TO 11:READ PAT(I):NEXT I
CALL BACKPAT(VARPTR(PAT(4)))
OVAL(0)=47:OVAL(1)=147:OVAL(2)=193:OVAL(3)=293
CALL FILLOVAL(VARPTR(OVAL(0)),VARPTR(PAT(8)))
OVAL(0)=50:OVAL(1)=150:OVAL(2)=190:OVAL(3)=290
CALL FILLOVAL(VARPTR(OVAL(0)),VARPTR(PAT(4)))
DELTA=5
FOR ANG=15 TO 360 STEP 15
  FOR A=ANG-15 TO ANG STEP DELTA
    CALL FILLARC(VARPTR(OVAL(0)),0,A,VARPTR(PAT(8)))
  NEXT A
  FOR A = 0 TO 360 STEP DELTA
    CALL FILLARC(VARPTR(OVAL(0)),A+ANG,DELTA,VARPTR(P
AT(8)))
    CALL ERASEARC(VARPTR(OVAL(0)),A,DELTA)
  NEXT A
NEXT ANG
CALL BACKPAT(VARPTR(PAT(0)))
END
DATA 0,0,0,0
DATA 4420,4420,4420,4420
DATA -1,-1,-1,-1
```

Program 9-23 creates an optical illusion with moving pie sections. The test of your powers of observation is to correctly decide whether the gray or black section is moving.

As in the previous program, two arrays are dimensioned—OVAL(3) and PAT(11). The pattern values are read by the three **DATA** statements at the end of the program listing. The background pattern is set to light gray with **BACKPAT**. An outer circle is defined in OVAL(3), then drawn and filled with **FILLOVAL**. An inner circle is created in the same way. DELTA controls the rotational speed of the pie section in motion.

Figure 9-24. *The pie section is animated with FILLARC and ERASEARC.*
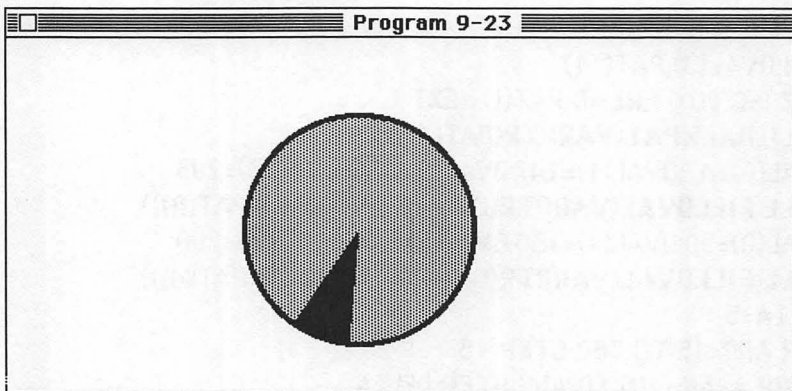
The main section of the program, the **FOR-NEXT** loop, sets the angular width of the animated pie section to ANG. This value increases by 15 degrees during each 360-degree rotation. **FILLARC** is called a number of times to get this done. The next **FOR-NEXT** causes the pie section to rotate once around the inner circle, performed by drawing the leading edge with **FILLARC** and then erasing the trailing edge with **ERASEARC**. When the illusion is over, the background pattern is reset by **BACKPAT**.

Arc Inversion
If a pie section must be inverted—white turned to black or black turned to white—call the ROM routine **INVERTARC**:

CALL INVERTARC(VARPTR(*oval array element***),***start angle,arc angle***)**

INVERTARC works similarly to the previous two commands except that it inverts a pie section from a circle or ellipse via an arc. Program 9-24 puts **INVERTARC** to work. Exit the program by selecting *Stop* from the *Run* menu. Beware of a possible delayed mouse response.

Program 9-24. INVERTARC

```
CLS
DEFINT A-Z
DIM OVAL(3)
FOR I=0 TO 75 STEP 25
```

```
OVAL(0)=30+I: OVAL(1)=140+I: OVAL(2)=230-I: OVAL(3)=340-I
CALL FRAMEOVAL(VARPTR(OVAL(0)))
NEXT I
CALL MOVETO(140,130):CALL LINE(199,0)
CALL MOVETO(240,30):CALL LINE(0,199)
OVAL(0)=31: OVAL(1)=141: OVAL(2)=229: OVAL(3)=339
FOR I=1 TO 10
   X=165+INT(150*RND(1)):Y=55+INT(150*RND(1))
   PSET(X,Y):PSET(X+1,Y):PSET(X,Y+1):PSET(X+1,Y+1)
NEXT I
CALL INVERTARC(VARPTR(OVAL(0)),0,5)

LOOP:
FOR ANG=0 TO 355 STEP 5
   CALL INVERTARC(VARPTR(OVAL(0)),ANG,5)
   CALL INVERTARC(VARPTR(OVAL(0)),ANG+5,5)
NEXT ANG
GOTO LOOP
```

In some ways, Program 9-24 may resemble Program 9-23, in that a darkened pie slice moves throughout a 360-degree circle. Most of this program's code should be easy to identify.

The array OVAL(3) is used to define the radar grid's scaling rings, which in turn are drawn by a short **FOR-NEXT** loop and a call to **FRAMEOVAL**. Note that the difference between the topmost and bottommost coordinates is the same as the difference between the leftmost and rightmost positions, making the ovals circles.

Two lines calling **MOVETO** and **LINE** draw the cross-hairs before the second **FOR-NEXT** loop randomly places ten black blips on the "screen." The initial radar beam is drawn by **INVERTARC**.

The main part of the program is the LOOP subroutine, which updates the grid 72 times per single revolution of the radar beam. The old radar beam is erased with one call to **INVERTARC**, while the new one is drawn with another. Each beam is 5 degrees wide, as you can see from both lines.

Figure 9-25. *INVERTARC is used to draw and animate a radar screen.*
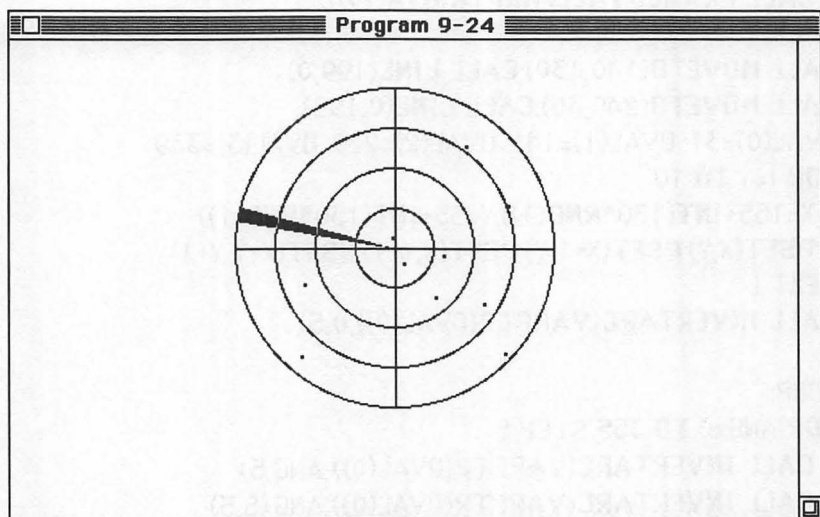
And Last, Painting

Finally, oval segments are painted with the ROM routine **PAINTARC**:

CALL PAINTARC(VARPTR(*oval array element*),*start angle,arc angle*)

PAINTARC draws an arc from an oval or ellipse defined like that in **FRAMEARC** and fills it with the current pen pattern. Program 9-25 is a fairly sophisticated example of using each of the Macintosh's **PAINT** ROM routines. Select the various buttons to control pen attributes and shapes that are to be framed or painted. Choose the *Quit* button to end the program.

Program 9-25. All the PAINT There Is

```
CLS
DEFINT A-Z
CALL TEXTMODE(1)
DIM RECT(3),PAT(15),BUTTN(3,23)
PHBUT=1:PWBUT=1:PATBUT=1:BKGBUT=4:SBUT=1
FOR I=0 TO 15:READ PAT(I):NEXT I
CALL MOVETO(25,12)
PRINT "Type";SPC(5);"Pen Height";SPC(2);"Pen Width";SPC(4);"P
```

218

```
attern";SPC(4);"Background";SPC(3);"Shape"
FOR I=0 TO 23
  T=((I /4) - (I \ 4))*100+20:L=(I\4)*80+5
  BUTTN(0,I)=T:BUTTN(1,I)=L:BUTTN(2,I)=T+20:BUTTN(3,I)=L+70
  CALL FRAMEROUNDRECT(VARPTR(BUTTN(0,I)),10,10)
  READ LB$,W
  CALL MOVETO(L+5+W,T+13):PRINT LB$;
NEXT I
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,4)),10,10)
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,8)),10,10)
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,12)),10,10)
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,19)),10,10)
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,20)),10,10)
RECT(0)=140:RECT(1)=100:RECT(2)=250:RECT(3)=380
LINE (0,120)- STEP (500,0)

READMOUSE:
WHILE MOUSE(0)<1:WEND
X=MOUSE(1):Y=MOUSE(2)
BUTSEL= -1:B=0
WHILE (BUTSEL<0) AND (B<24)
  IF (X>BUTTN(1,B)) AND (X<BUTTN(3,B)) AND (Y>BUTTN(0,B)) AN
D (Y<BUTTN(2,B)) THEN BUTSEL=B
  B=B+1
WEND
IF BUTSEL<>-1 THEN ON BUTSEL+1 GOSUB FRAME,PAINT,ERAES,Q
UIT,PENHT,PENHT,PENHT,PENHT,PENWD,PENWD,PENWD,PENWD,PAT
,PAT,PAT,PAT,BKPAT,BKPAT,BKPAT,BKPAT,SHAPE,SHAPE,SHAPE,
SHAPE
GOTO READMOUSE

FRAME:
  ON SBUT GOTO FR,FRR,FO,FA

  FR:  CALL FRAMERECT(VARPTR(RECT(0))):GOTO ENDFRAME

  FRR: CALL FRAMEROUNDRECT(VARPTR(RECT(0)),20,20):GOT
O ENDFRAME
```

```
FO:  CALL FRAMEOVAL(VARPTR(RECT(0))):GOTO ENDFRAME

FA:  CALL FRAMEARC(VARPTR(RECT(0)),45,90)

ENDFRAME: RETURN

PAINT:
  ON SBUT GOTO PR,PRR,PO,PA

  PR:  CALL PAINTRECT(VARPTR(RECT(0))):GOTO ENDPAINT

  PRR: CALL PAINTROUNDRECT(VARPTR(RECT(0)),20,20):GOT
O ENDPAINT

  PO:  CALL PAINTOVAL(VARPTR(RECT(0))):GOTO ENDPAINT

  PA:  CALL PAINTARC(VARPTR(RECT(0)),45,90)

ENDPAINT: RETURN

ERAES:
  ON SBUT GOTO ER,ERRR,EO,EA

  ER:   CALL ERASERECT(VARPTR(RECT(0))):GOTO ENDERAES

  ERRR: CALL ERASEROUNDRECT(VARPTR(RECT(0)),20,20):GOT
O ENDERAES

  EO:   CALL ERASEOVAL(VARPTR(RECT(0))):GOTO ENDERAES

  EA:   CALL ERASEARC(VARPTR(RECT(0)),45,90)

ENDERAES: RETURN

QUIT:
  CALL PENNORMAL
```

```
ENDERAES: RETURN

QUIT:
  CALL PENNORMAL
  CALL TEXTMODE(0)
  CALL BACKPAT(VARPTR(PAT(12)))
  END
  RETURN

PENHT:
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,PHBUT+3)),10,1
0)
  PHBUT=BUTSEL-3
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,PHBUT+3)),10,1
0)
  CALL PENSIZE(PWBUT,PHBUT)
  RETURN

PENWD:
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,PWBUT+7)),10,1
0)
  PWBUT=BUTSEL-7
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,PWBUT+7)),10,1
0)
  CALL PENSIZE(PWBUT,PHBUT)
  RETURN

PAT:
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,PATBUT+11)),10
,10)
  PATBUT=BUTSEL-11
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,PATBUT+11)),10
,10)
  CALL PENPAT(VARPTR(PAT((PATBUT-1)*4)))
  RETURN

BKPAT:
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BKGBUT+15)),10
```

```
,10)
  BKGBUT=BUTSEL-15
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BKGBUT+15)),10
,10)
  CALL BACKPAT(VARPTR(PAT((BKGBUT-1)*4)))
  RETURN

SHAPE:
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,SBUT+19)),10,1
0)
  SBUT=BUTSEL-19
  CALL INVERTROUNDRECT(VARPTR(BUTTN(0,SBUT+19)),10,1
0)
  RETURN
DATA -1,-1,-1,-1
DATA 21930,21930,21930,21930
DATA 4420,4420,4420,4420
DATA 0,0,0,0
DATA "Frame",12,"Paint",14,"Erase",14,"Quit",19
DATA "1",25,"2",25,"3",25,"4",25
DATA "1",25,"2",25,"3",25,"4",25
DATA "Black",14,"Gray",18,"Lt. Gray",8,"White",11
DATA "Black",14,"Gray",18,"Lt. Gray",8,"White",11
DATA "Rect.",12,"Rnd. Rect.",0,"Oval",12,"Arc",16
```

Program 9-25 draws shapes based on the current selection of options—a rectangle, round-cornered rectangle, oval, or an arc. The first three shapes can be framed, painted, and erased. The ROM routines prefixed with **FRAME**, **PAINT**, and **ERASE** can be tested with different pen attributes and background patterns. This program can be used to test the effects of combining the various graphic controls provided by the Macintosh ROM routines.

There are six sets of four buttons each functioning independently. The buttons control which ROM routine is used. For example, if the *Oval* and *Paint* buttons are selected, **PAINTOVAL** is performed. The other buttons are used with **PENSIZE**, **PENPAT**, and **BACKPAT**. Selecting *Quit* ends the program.

Figure 9-26. *This is a sample of one of the images which can be created with this pattern and pen attributes tester. Different pen heights, pen widths, and pen patterns can be selected to compare the effects of using the FRAME, PRINT, and ERASE ROM routines.*
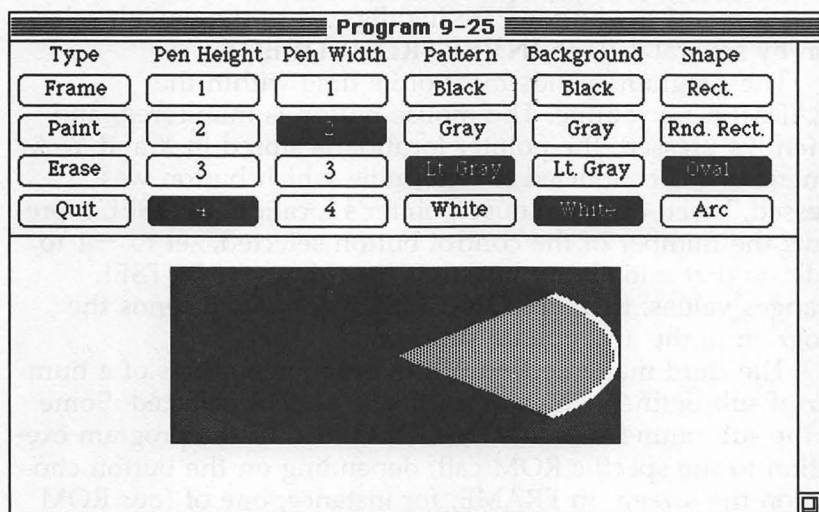
The program is divided into three major sections: initialization of variables and screen setup, scan for and decoding mouse input, and the set of subroutines which process the program options.

The first section ends at the READMOUSE subroutine. After all variables are made integers by **DEFINT**, the textmode is set to OR mode so that the control buttons can be labeled properly. (See the next chapter for details on the Macintosh's text ROM routines.) Using *Copy* mode causes some of the buttons to be erased when they're labeled. Three arrays used by the program are dimensioned: RECT(3), a rectangle array when drawing rectangles and round-cornered rectangles, and an oval array for drawing ovals, oval sections, and arcs; PAT(15), a pattern array for the four patterns (black, gray, light gray, and white); and BUTTN(3,23), 24 rectangle arrays for the selection buttons. In this two-dimensional array, the rectangle array element of button *x* is equal to BUTTN(0,*x*). This array element contains the value of the location of the top of button *x*. The location values of the left, bottom, and right sides of

button x are stored in BUTTN(1,x), BUTTN(2,x), and BUTTN(3,x).

The remainder of this section of the program initializes several global variables, reads the pattern information from **DATA** statements, displays titles, and draws each of the buttons. Some of the buttons are highlighted to the default selection by several calls to **INVERTROUNDRECT**.

The program spends most of its time within the READMOUSE routine. The mouse button is monitored, and when it's pressed, the pointer location is stored in X and Y. A generic test is conducted to determine which button was pressed, based on the mouse pointer's location. BUTSEL represents the number of the control button selected, set to -1 to indicate that initially no button is held down. If BUTSEL changes values, then the **ON-GOSUB** command sends the program to the appropriate subroutine.

The third major section of the program consists of a number of subroutines which perform the options selected. Some of the subroutines use an **ON-GOTO** to shift the program execution to the specific ROM call, depending on the button chosen on the screen. In FRAME, for instance, one of four ROM routines can be accessed—**FRAMERECT, FRAME-ROUNDRECT, FRAMEOVAL,** or **FRAMEARC**—depending on which of the four shapes was picked.

Polygon Manipulation

In Microsoft BASIC 2.0, there are several ROM routines available for manipulating polygons. Polygons are defined by a set of points and lines that connect these points.

Figure 9-27 shows how polygon points are defined in the data array. The example has six sides which meet at six points, labeled clockwise from point 1. (Each of the points in the figure is exaggerated and does not stand out when the polygon is displayed on the screen.) The coordinates of each of the points are indicated with their labels. The polygon is defined by an integer polygon data array.

The polygon data array contains three sections of data. The first is an array element which contains the number of bytes in the array. This must be the first array element, and is also known as the *polygon data array element*. The number of bytes is determined by multiplying the number of polygon data array elements by two, since each element is an integer

Figure 9-27. *Definition of a polygon.*

EN	Polygon Data Array		
0	38	— Number of bytes	Size of the polygon data array.
1	50	— Topmost point	
2	20	— Leftmost point	The rectangle that defines the bounding limits of the polygon.
3	190	— Bottommost point	
4	100	— Rightmost point	
5	50	— Vertical location of point 1	
6	20	— Horizontal location of point 1	
7	50	— Vertical location of point 2	
8	40	— Horizontal location of point 2	
9	170	— Vertical location of point 3	
10	40	— Horizontal location of point 3	Coordinates of the endpoints of the lines that when connected form the polygon.
11	170	— Vertical location of point 4	
12	100	— Horizontal location of point 4	
13	190	— Vertical location of point 5	
14	100	— Horizontal location of point 5	
15	190	— Vertical location of point 6	
16	100	— Horizontal location of point 6	
17	50	— Vertical location of point 1	
18	20	— Horizontal location of point 1	

which occupies two bytes of memory.

The second group of polygon data array elements is a set of four elements to define the location of the maximum extremes of the polygon's points. The first element in this group defines the topmost point, while the second, third, and fourth elements define the leftmost, bottommost, and rightmost points, respectively. In Figure 9-27, the topmost points are points 1 and 2, which have 50 as their vertical positions. The leftmost points are points 1 and 6 (20), the bottommost points are 5 and 6 (190), and the rightmost points are 4 and 5 (100). The four elements in this group thus have values of 50, 20, 190, and 100.

The third group of polygon data array elements is a list of coordinates defining the endpoints of the polygon. In order to have a line connecting all of the endpoints, the points must be

listed in the order in which the lines are to be drawn. That means that the first point must also be listed after the last point so that the lines connect the last point to the first. Each point first lists its vertical position in one element, then its horizontal position in the next. If there are six points (as in the figure), there would be seven pairs of numbers in this group. (Again, remember that point 1's coordinates must be repeated to close the polygon.)

Combining these three groups of numbers defines a polygon. Polygon data array elements are numbered in the figure under the column labeled EN (Element Number). If the first element was POLYGON%(0), then the last element would be POLYGON%(18). The number of array elements required by the polygon definition would be 19, making the number of bytes required 38.

A ROM routine which deals with polygons is **FRAMEPOLY**:

CALL FRAMEPOLY(VARPTR(*polygon data array element***))**

This draws the outline, or frame, of the polygon defined by an integer polygon data array. The size of the lines used to draw the polygon can be changed with **PENSIZE**.

Another ROM routine to draw polygons is **FILLPOLY**:

CALL FILLPOLY(VARPTR(*polygon data array element***),VARPTR(***pattern array element***))**

This fills a polygon defined as with **FRAMEPOLY** with a pattern defined as with **FILLRECT**.

Erasing polygons is performed by the ROM routine **ERASEPOLY**:

CALL ERASEPOLY(VARPTR(*polygon data array element***))**

ERASEPOLY erases a defined polygon with the background pattern.

Inverting polygons is performed with **INVERTPOLY**:

CALL INVERTPOLY(VARPTR(*polygon data array element***))**

This inverts a polygon defined as in **FRAMEPOLY**.

PAINTPOLY fills polygons:

CALL PAINTPOLY(VARPTR(*polygon data array element***))**

This fills a polygon defined as in **FRAMEPOLY** with a pattern defined by **PENPAT**.

Take a look at what Program 9-26 does. It uses all the various ROM routines dedicated to manipulating polygons.

Program 9-26. Polygons in BASIC

```
CLS
DEFINT A-Z
DIM POLY1(18),POLY2(22),POLY3(22),POLY4(50),POLY5(22),POLY6
(22),PAT(7)
FOR I=0 TO 18:READ POLY1(I):NEXT I
FOR I=0 TO 22:READ POLY2(I):NEXT I
FOR I=0 TO 22:READ POLY3(I):NEXT I
FOR I=0 TO 50:READ POLY4(I):NEXT I
FOR I=0 TO 22:READ POLY5(I):NEXT I
FOR I=0 TO 22:READ POLY6(I):NEXT I
FOR I=0 TO 3:READ PAT(I):NEXT I
CALL PENSIZE(2,2)
CALL FRAMEPOLY(VARPTR(POLY1(0)))
CALL FRAMEPOLY(VARPTR(POLY2(0)))
CALL FRAMEPOLY(VARPTR(POLY3(0)))
CALL FRAMEPOLY(VARPTR(POLY4(0)))
CALL FRAMEPOLY(VARPTR(POLY5(0)))
CALL FRAMEPOLY(VARPTR(POLY6(0)))
CALL PENSIZE(1,1)
CALL MOVETO(20,20):PRINT "FRAMEPOLY: Click Mouse Button."
WHILE MOUSE(0)<1:WEND
CALL FILLPOLY(VARPTR(POLY1(0)),VARPTR(PAT(0)))
CALL FILLPOLY(VARPTR(POLY2(0)),VARPTR(PAT(0)))
CALL FILLPOLY(VARPTR(POLY3(0)),VARPTR(PAT(4)))
CALL FILLPOLY(VARPTR(POLY4(0)),VARPTR(PAT(0)))
CALL FILLPOLY(VARPTR(POLY5(0)),VARPTR(PAT(0)))
CALL FILLPOLY(VARPTR(POLY6(0)),VARPTR(PAT(4)))
CALL MOVETO(20,20):PRINT "FILLPOLY: Click Mouse Button."
WHILE MOUSE(0)<1:WEND
CALL ERASEPOLY(VARPTR(POLY1(0)))
CALL ERASEPOLY(VARPTR(POLY2(0)))
CALL ERASEPOLY(VARPTR(POLY3(0)))
CALL ERASEPOLY(VARPTR(POLY4(0)))
CALL ERASEPOLY(VARPTR(POLY5(0)))
CALL ERASEPOLY(VARPTR(POLY6(0)))
CALL MOVETO(20,20):PRINT "ERASEPOLY: Click Mouse Button."
WHILE MOUSE(0)<1:WEND
```

227

```
CALL INVERTPOLY(VARPTR(POLY1(0)))
CALL INVERTPOLY(VARPTR(POLY2(0)))
CALL INVERTPOLY(VARPTR(POLY3(0)))
CALL INVERTPOLY(VARPTR(POLY4(0)))
CALL INVERTPOLY(VARPTR(POLY5(0)))
CALL INVERTPOLY(VARPTR(POLY6(0)))
CALL MOVETO(20,20):PRINT "INVERTPOLY: Click Mouse Button."
WHILE MOUSE(0)<1:WEND
CALL PENPAT(VARPTR(PAT(0)))
CALL PAINTPOLY(VARPTR(POLY1(0)))
CALL PAINTPOLY(VARPTR(POLY2(0)))
CALL PAINTPOLY(VARPTR(POLY4(0)))
CALL PAINTPOLY(VARPTR(POLY5(0)))
CALL PENPAT(VARPTR(PAT(4)))
CALL PAINTPOLY(VARPTR(POLY3(0)))
CALL PAINTPOLY(VARPTR(POLY6(0)))
CALL MOVETO(20,20):PRINT "PAINTPOLY: Click Mouse Button."
WHILE MOUSE(0)<1:WEND
END
DATA 38:'bytes for L
DATA 50,20,190,100:'bounds
DATA 50,20,50,40,170,40,170,100,190,100,190,20,50,20:'point
s y,x pairs
DATA 46:'bytes for O
DATA 50,120,190,200:'bounds
DATA 70,120,50,140,50,180,70,200,170,200,190,180,190,140,
170,120,70,120:'points y,x pairs
DATA 46:'bytes for O
DATA 70,140,170,180:'bounds
DATA 80,140,70,150,70,170,80,180,160,180,170,170,170,150,
160,140,80,140:'points y,x pairs
DATA 102:'bytes for G
DATA 50,20,190,100:'bounds
DATA 70,220,50,240,50,280,70,300,100,300,100,280,80,280,7
0,270,70,250,80,240,160,240,170,250,170,270,160,280,140,28
0,140,260,120,260,120,300,170,300,190,280,190,240,170,220
,70,220:'points y,x pairs
```

```
DATA 46:'bytes for O
DATA 50,320,190,400:'bounds
DATA 70,320,50,340,50,380,70,400,170,400,190,380,190,340,
170,320,70,320:'points y,x pairs
DATA 46:'bytes for O
DATA 70,340,170,380:'bounds
DATA 80,340,70,350,70,370,80,380,160,380,170,370,170,350,
160,340,80,340:'points y,x pairs
DATA 21930,21930,21930,21930
DATA 0,0,0,0
```

Program 9-26 draws the word *LOGO* with the aid of the various polygon ROM routines. The word is first displayed with **FRAMEPOLY**. A message indicates which ROM routine is in use. Pressing the mouse button displays the polygon with **FILLPOLY, ERASEPOLY, INVERTPOLY,** and **PAINTPOLY**.

Figure 9-28. *Different polygon-oriented ROM routines are demonstrated by Program 9-26.*
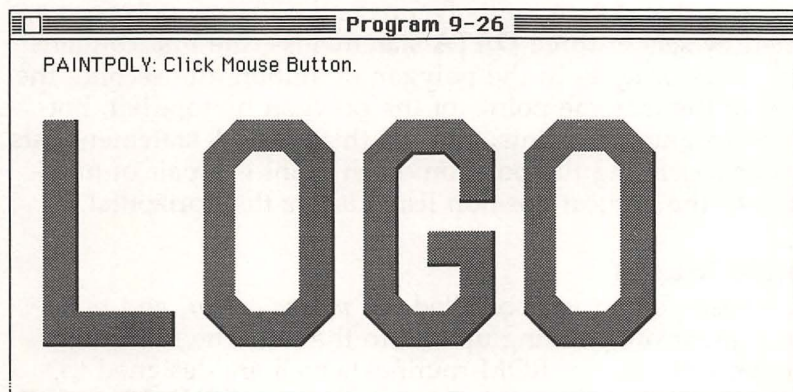
The program code is divided into three sections. The first defines and initializes variables, defines the polygons, and prepares the *Output* window. Seven arrays are defined within the program—the first six are polygon data arrays, obvious when looking at their names. The array POLY1(18) represents the letter *L*; POLY2(22) and POLY3(22) represent the outside and inside ovals respectively for the first *O*; POLY4(50) represents

G; and POLY5(22) and POLY6(22) represent the ovals for the second *O*. The patterns used with the **FILLPOLY**, **BACKPAT**, and **PENPAT** are stored in PAT(7). The next seven lines contain **FOR-NEXT** loops which read the polygon and pattern definitions from **DATA** statements at the end of the program.

The next section calls each of the polygon ROM routines. **PENSIZE** is called prior to drawing the polygons with **FRAMEPOLY**. After the polygons are drawn, **PENSIZE** resets the pen size. A message is positioned at the top left corner of the *Output* window with **MOVETO** and **PRINT**. A **WHILE-WEND** loop stalls program execution until the mouse button is pressed. The second polygon-related ROM routine is **FILLPOLY**; all the polygons are drawn with a gray fill pattern except for the centers of the *O*'s, which are drawn with a white fill pattern and so must be drawn after the outer ovals. **ERASEPOLY** leaves a shadow of the word because **FRAMEPOLY** was called with a larger pen size. **INVERTPOLY** makes the originally white lettering look black. **PENPAT** is called before **PAINTPOLY** in order to define the pattern with which to fill the polygons (gray).

The third section of the program is comprised of **DATA** statements that define the polygons and patterns. Polygons are defined by sets of three **DATA** statements—the first contains the number of bytes in the polygon definition; the second, the values of the extreme points of the polygon by top, left, bottom, and rightmost points; and the third **DATA** statement lists the points defining the polygon. Each point is a pair of numbers with the vertical position listed *before* the horizontal.

Mouse Masks

The mouse pointer is also called the *mouse cursor*, and typically is an arrow pointing up and to the left. The Macintosh includes a number of ROM routines which are designed to manipulate this pointer, including one called **OBSURECURSOR**:

CALL OBSCURECURSOR

OBSCURECURSOR hides the mouse pointer until the mouse is moved. Program 9-27 is an example.

Program 9-27. OBSURECURSOR

```
CLS
DEFINT A-Z
```

```
DIM RECT(3)
RECT(0)=30:RECT(1)=80:RECT(2)=170:RECT(3)=380
CALL PENSIZE(3,3)
CALL FRAMERECT(VARPTR(RECT(0)))
RECT(0)=35:RECT(1)=85:RECT(2)=165:RECT(3)=375
CALL PENNORMAL
CALL FRAMERECT(VARPTR(RECT(0)))
CALL TEXTMODE(1)
CALL TEXTFONT(0)
CALL MOVETO(97,55):PRINT "Move the mouse. Press any key to
"
CALL MOVE(95,0):PRINT "OBSCURE the mouse pointer. Move the
"
CALL MOVE(95,0):PRINT "mouse to show the mouse pointer aga
in."
CALL MOVE(95,0):PRINT "Press the 'Q' key to exit this program
"

GETKEY:
   K$=INKEY$
   IF K$="Q" OR K$="q" THEN GOTO ENDPROG
   IF K$<>"" THEN CALL OBSCURECURSOR
GOTO GETKEY

ENDPROG:
CALL TEXTMODE(0)
CALL TEXTFONT(1)
```

Program 9-27 displays a dialog box containing instructions. The mouse pointer remains visible until any key other than Q is pressed.

The rectangle array RECT(3) defines the dialog box. The outer perimeter, drawn with a pen size three times larger than normal, is set with **PENSIZE** and drawn with **FRAMERECT**. The inner perimeter is drawn after the pen is reset with **PENNORMAL**. To prevent the text within the dialog box from obscuring the outline, textmode is set to the OR mode with **TEXTMODE**. (The following chapter covers **TEXTMODE** in detail.) The System font is chosen to display the text within

Figure 9-29. *OBSCURECURSOR hides the pointer until the mouse is moved.*

```
┌─▣▤▤▤▤▤▤▤▤▤▤▤▤▤ Program 9-27 ▤▤▤▤▤▤▤▤▤▤▤▤▤┐
│                                                    │
│   ┌──────────────────────────────────────┐         │
│   │ Move the mouse. Press any key to      │         │
│   │ OBSCURE the mouse pointer. Move the   │         │
│   │ mouse to show the mouse pointer again.│         │
│   │ Press the 'Q' key to exit this program.│        │
│   │                                        │        │
│   └──────────────────────────────────────┘         │
│                                                    │
│                    ▸                               │
│                                                    │
└────────────────────────────────────────────────────┘
```
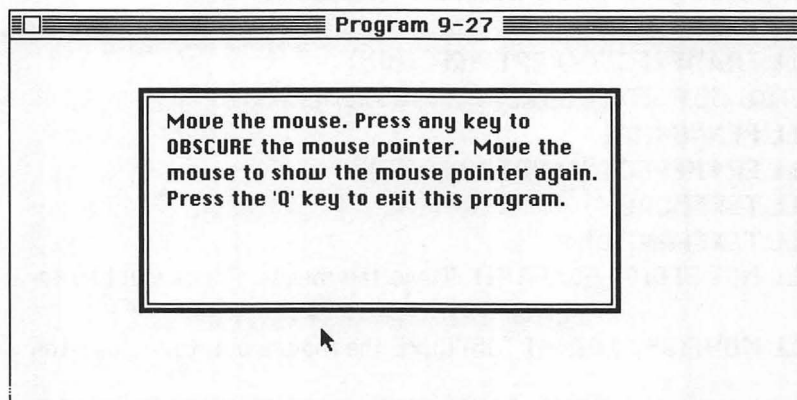
the dialog box with **TEXTFONT**(0).

The majority of the program's time is spent in the GETKEY loop, where the keyboard is monitored with **INKEY$**. If a key is pressed, the value of K$ will not be a null string, and **OBSURECURSOR** will be called. As soon as the mouse pointer is moved again, it becomes visible.

Cursor Invisible

The pointer can be made to hide so that even mouse activity cannot make it visible:

CALL HIDECURSOR

To make the mouse pointer visible again, use:

CALL SHOWCURSOR

Try out Program 9-28. Simply follow the directions in the dialog box.

Program 9-28. Hide and Seek

```
CLS
DEFINT A-Z
DIM RECT(3)
RECT(0)=30:RECT(1)=80:RECT(2)=170:RECT(3)=380
CALL PENSIZE(3,3)
CALL FRAMERECT(VARPTR(RECT(0)))
RECT(0)=35:RECT(1)=85:RECT(2)=165:RECT(3)=375
```

```
CALL PENNORMAL
CALL FRAMERECT(VARPTR(RECT(0)))
CALL TEXTMODE(1)
CALL TEXTFONT(0)
CALL MOVETO(97,55):PRINT "Press any key to hide the mouse p
ointer."
CALL MOVE(95,0):PRINT "Press another key to show the mouse"
CALL MOVE(95,0):PRINT "pointer again."
CALL MOVE(95,0):PRINT "Press the 'Q' key to exit this program
"


GETKEY1:
  K$=INKEY$
  IF K$="Q" THEN GOTO ENDPROG
  IF K$<>"" THEN CALL HIDECURSOR ELSE GOTO GETKEY1

GETKEY2:
  K$=INKEY$
  IF K$="Q" OR K$="q" THEN GOTO ENDPROG
  IF K$<>"" THEN CALL SHOWCURSOR ELSE GOTO GETKEY2
GOTO GETKEY1

ENDPROG:
CALL TEXTMODE(0)
CALL TEXTFONT(1)
CALL SHOWCURSOR
```

Press a key and the mouse pointer is obscured. Move the mouse and notice that it stays hidden. To make it visible again, press another key. Q terminates the program.

Only minor differences appear in this program when compared with Program 9-27. The addition of another GETKEY subroutine is about it. The first keypress, as read by GETKEY1, calls **HIDECURSOR**, while the second keypress, read by GETKEY2, calls **SHOWCURSOR**. It's that simple.

Figure 9-30. *HIDECURSOR and SHOWCURSOR control the visibility of the mouse pointer.*

```
▤□══════════════ Program 9-28 ══════════════

        ┌─────────────────────────────────────────┐
        │ Press any key to hide the mouse pointer. │
        │ Press another key to show it again.      │
        │                                          │
        │ Press the 'Q' key to exit this program.  │
        │                                          │
        │                                          │
        │                                      ▶    │
        └─────────────────────────────────────────┘
```
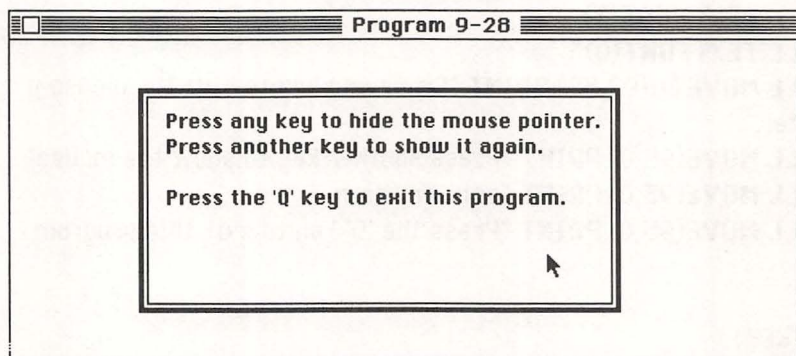
Changing Mouse Tails

Microsoft BASIC lets you change the shape of the mouse pointer or mouse cursor by using a ROM routine and an integer cursor array. Figure 9-31 illustrates the process of placing appropriate values in the array. Refer to it often as you read through this next section.

The cursor array has 34 elements, numbered 0 through 33, divided into two sections of 16 elements each. The first section is the image of the mouse cursor, the second the mask for that image. The third section of two elements is the cursor *hot spot*.

The cursor in Figure 9-31 consists of 16 rows with 16 pixels each. The shape of the cursor is determined by black pixels in the grid. Each row is represented by a value in one of the first 16 elements of the array. (The topmost row is represented by the first cursor array element, element 0.) The elements are numbered at the right edge under the heading CAEN (Cursor Array Element Number). The value for a row is determined by adding only the black pixels' values. The values for each individual pixel are indicated on the left. Where none of the pixels in a row is set to black, the value is zero. The values for each of the cursor array elements are listed on the top right side under Cursor Data.

The cursor mask is defined in the same manner as the image for the pointer—the top row is stored in cursor array element 16, while the bottom is stored in 31. The mask controls

234

Figure 9-31. *Definition of a mouse pointer.*

Mouse Pointer Cursor

Cursor Data	CREN
0	0
128	1
128	2
128	3
2032	4
1040	5
1040	6
15390	7
1040	8
1040	9
2032	10
128	11
128	12
128	13
0	14
0	15

Pixel Value

-32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
1

Mouse Pointer Mask

Mask Data	CREN
448	16
448	17
448	18
4088	19
4088	20
3096	21
31775	22
31775	23
31775	24
3096	25
4088	26
4088	27
448	28
448	29
448	30
0	31

HOT SPOT DATA	CREN
7	32
8	33
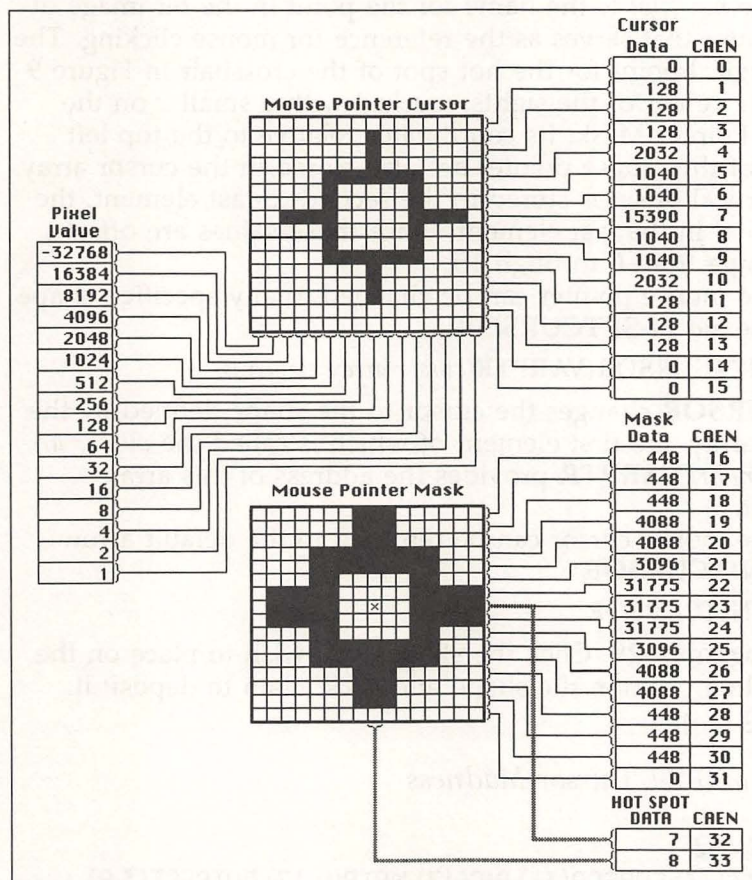
how to draw the mouse cursor. If the pixel in the image is not black and the corresponding pixel in the mask is also not black, the screen background shows through the cursor. However, if the pixel in the pointer is black and the corresponding pixel in the mask is *not* black, the background is displayed as its inverse. This makes the pointer image appear over any background. When the pixel in the image is not black and the corresponding pixel in the mask is black, the background is replaced by white—often done to place a white frame around the pointer image. Finally, when the pixel in the pointer is black and the corresponding pixel in the mask is also black, the background is replaced by black. That's how the cursor is

most commonly drawn.

The *hot spot* is the name for the point in the bit image of the pointer that serves as the reference for mouse clicking. The most logical point for the hot spot of the crosshair in Figure 9-31 is the center of the sights, marked with a small x on the Mouse Pointer Mask. Its coordinates relative to the top left corner of the mouse pointer are also stored in the cursor array. The vertical offset is stored in the second-to-last element, the horizontal in the last element. Since these values are offsets, they range from 0 through 15.

The mouse pointer can be changed to any specified shape with the aid of **SETCURSOR**:

CALL SETCURSOR(VARPTR(*cursor array element***))**

SETCURSOR changes the cursor to the shape defined by the cursor array, the first element of which is called the *cursor array element*. **VARPTR** provides the address of this array element.

The mouse cursor can be returned to the default arrow with **INITCURSOR**:

CALL INITCURSOR

Try Program 9-29. Click the shapes you wish to place on the easel, then position the shape and click again to deposit it. Press Q to quit.

Program 9-29. Cursor Madness

```
CLS
DEFINT A-Z
DIM EASEL(3),CURSOR(33),PIC(17),MPTR(9,17),BUTRECT(3,9)
FOR I=0 TO 9
  READ MPTR(I,0),MPTR(I,1)
  FOR J=1 TO MPTR(I,1)
    READ MPTR(I,J+1)
    PIC(J+1)=MPTR(I,J+1)
  NEXT J
  BUTRECT(0,I)=10+I*26:BUTRECT(1,I)=10:BUTRECT(2,I)=BUTREC
T(0,I)+26:BUTRECT(3,I)=36
  CALL FRAMERECT(VARPTR(BUTRECT(0,I)))
  PIC(0)=MPTR(I,0):PIC(1)=MPTR(I,1)
  PUT (BUTRECT(1,I)+13-PIC(0)\2,BUTRECT(0,I)+13-PIC(1)\2),P
IC,OR
```

```
NEXT I
EASEL(0)=10:EASEL(1)=40:EASEL(2)=270:EASEL(3)=480
CALL PENSIZE(3,3):CALL FRAMERECT(VARPTR(EASEL(0)))
EASEL(0)=13:EASEL(1)=43:EASEL(2)=267:EASEL(3)=477
FOR I=2 TO MPTR(0,1)+1
  CURSOR(I-2)=MPTR(0,I)
NEXT I
CALL SETCURSOR(VARPTR(CURSOR(0)))
FOR I=0 TO MPTR(0,1)+1:PIC(I)=MPTR(0,I):NEXT I

READMOUSE:
K$="":WHILE MOUSE(0)<1 AND K$="":K$=INKEY$:WEND
IF K$="Q" OR K$="q" THEN FINPROG
X=MOUSE(5):Y=MOUSE(6)
IF X>EASEL(1) AND X<EASEL(3)-PIC(0) AND Y>EASEL(0) AND Y<EA
SEL(2)-PIC(1) THEN CALL HIDECURSOR:PUT (X,Y),PIC,OR:CALL
 SHOWCURSOR:GOTO READMOUSE
BUTSEL=-1:I=0
WHILE BUTSEL<0 AND I<10
    IF X>BUTRECT(1,I) AND X<BUTRECT(3,I) AND Y>BUTRECT(0,I) A
ND Y<BUTRECT(2,I) THEN BUTSEL=I
    I=I+1
WEND
IF BUTSEL=-1 THEN READMOUSE
FOR I=0 TO 15: CURSOR(I)=0:NEXT I
FOR I=2 TO MPTR(BUTSEL,1)+1:CURSOR(I-2)=MPTR(BUTSEL,I):NEX
T I
FOR I=0 TO MPTR(BUTSEL,1)+1:PIC(I)=MPTR(BUTSEL,I):NEXT I
CALL SETCURSOR(VARPTR(CURSOR(0)))
GOTO READMOUSE

FINPROG:
CALL INITCURSOR
END
DATA 7,14,-512,17408,17408,17408,17408,-32256,-32256,-5
12,-32256,-512,-32256,-32256,-32256,31744
DATA 16,5,-1,21845,-21846,21845,-1
DATA 9,15,-7296,-27520,-27520,27392,5120,27392,-27520,-
27520,-5248,2048,2048,4096,4096,4096,4096
```

DATA 15,12,-2,-32766,-32766,-2,16388,27308,8200,10920,41
12,6832,2080,4064
DATA 1,5,-32768,-32768,-32768,-32768,-32768
DATA 12,16,32736,-32752,-24688,-24496,-24496,-24496,-24
496,-24496,-24688,-32752,-31792,-32752,-32752,-16,-3275
2,32736
DATA 13,8,32752,-32760,-32536,-16408,-32536,-32760,-327
60,32752
DATA 11,8,8320,8320,8320,8320,8320,8320,16448,-32
DATA 7,14,-512,-32256,-32256,-32256,-32256,-32256,-3225
6,-32256,31744,4096,4096,4096,14336,-512
DATA 10,9,-512,-21632,-10688,-21952,-10688,-21632,-1075
2,-22016,31744

Figure 9-32. *The mouse pointer can be set to any of the symbols on the left. The shapes can be placed on the easel by clicking the mouse button.*

Program 9-29
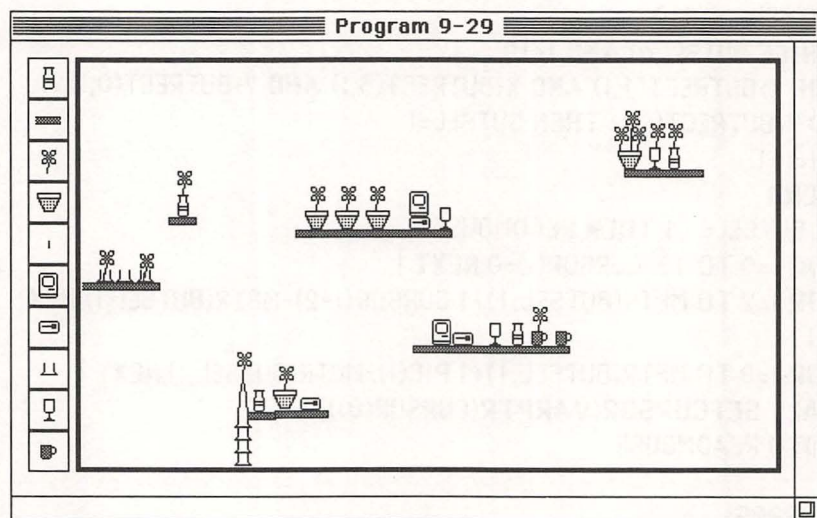
The arrays to look for are CURSOR(33), which holds the shape of the pointer, and PIC(17), which contains the shapes drawn on the easel. This array holds the bit image of the currently selected shape and is used in conjunction with **PUT**.

Since the largest shape is 16 pixels wide and 16 pixels high, this array is dimensioned to 18 elements. The first element, PIC(0), contains the width of the currently selected shape; the second, PIC(1), contains the height. Another important array, MPTR(9,17) stores the bit images of the ten shapes. MPTR(x,0) defines the width of shape x, while the array element MPTR(x,1) defines the height. The remaining 16 elements from MPTR(x,2) to MPTR(x,17) define the shape's bit image.

The **FOR** I=0 **TO** 9 loop is also worth looking at. I is the box (numbered 0–9—they're on the far left of the *Output* screen). The next line reads the width and height of the shape to be placed in box I from **DATA** statements. The number of data elements required to define the bit image of shape I is equal to the height of the shape. The nested **FOR-NEXT** loop uses this value to read the correct number of bit image data items. The bit image data is also stored in PIC(17) so that it can be drawn inside the box.

The **FOR** I=2 **TO** MPTR(0,1)+1 loop initializes the mouse cursor to the first shape before **SETCURSOR** changes the pointer.

User input is monitored through fairly standard means—looking for a keypress with **INKEY$** and for a mouse button press with **MOUSE**(0). If the mouse button is clicked, the pointer is hidden with **HIDECURSOR** before the shape is drawn, then displayed with **SHOWCURSOR**.

Selecting a new cursor shape is taken care of in the last part of the program. BUTSEL indicates if a box was selected. If BUTSEL is still -1, none was, and input is again monitored. Otherwise, BUTSEL equals the box number selected and the mouse pointer has to be changed to the shape defined within that selected box. The image portion of the existing cursor array is cleared by setting the first 16 elements to 0—**FOR** I=0 **TO** 15:CURSOR(I)=0:**NEXT** I. The next two lines define the bit image portion of the cursor array to the shape defined within the selected box. Finally, the pointer shape is set by **SETCURSOR**.

CHAPTER

10
Text ROM
Routines

10
Text ROM Routines

Since printing text in the *Output* window is a graphics operation on the Macintosh, it's easy to manipulate that text. One ROM routine used to control how text is drawn on the screen is **TEXTMODE**:

CALL TEXTMODE(*mode*)

- **TEXTMODE** draws text on the screen in one of four ways, specified by *mode*.
- The default mode is 0, called the *Overwrite* mode. Regardless of what's on the screen, any printed text will overwrite what is currently displayed. The area around the text will also overwrite the background.
- The second mode is 1, called the *OR* mode, since the resulting display is the same as performing a binary **OR** operation in which text appears but does not erase bits in the background. Text printed in this mode is always black, but cannot be seen over a black background.
- The third mode is 2, the *XOR* mode. This is like the default mode in that text always shows through any background. However, text appears as the inverse of the background and can be difficult to read on backgrounds in the gray ranges.
- The final mode is 3, called *Black is Changed* mode. Text printed in this mode appears as white wherever the background is black; the darker the background, the easier it is to see. It cannot be seen on white.

Program 10-1 shows you the various effects of **TEXTMODE**'s execution on three different backgrounds— white, gray, and black.

Program 10-1. TEXTMODE
```
CLS
DEFINT A-Z
```

```
DIM RECT(3),PAT(11)
FOR I=0 TO 11:READ PAT(I):NEXT I
CALL MOVETO(30,20):PRINT "Modes"
CALL MOVETO(200,20):PRINT "Background Patterns"
FOR MODE=0 TO 3
  READ MODE$
  CALL MOVETO(10,MODE*50+60):PRINT MODE$
  FOR P=0 TO 2
    RECT(0)=MODE*50+44:RECT(1)=P*120+125:RECT(2)=RECT(0)+
20:RECT(3)=RECT(1)+110
    CALL FILLRECT(VARPTR(RECT(0)),VARPTR(PAT(P*4)))
    CALL FRAMERECT(VARPTR(RECT(0)))
    CALL TEXTMODE(MODE)
    CALL MOVETO(RECT(1)+10,RECT(2)-6):PRINT "THIS IS A TE
ST";
    CALL TEXTMODE(0)
  NEXT P
NEXT MODE
WHILE MOUSE(0)=0:WEND
END
DATA 0,0,0,0
DATA 21930,21930,21930,21930
DATA -1,-1,-1,-1
DATA "Overwrite","Ored","Xored","Black is Changed"
```

You'll see a set of three test background patterns, each containing text.

Of the two arrays, the most important is PAT(11), which contains the 12 elements storing three patterns. The pattern array element PAT(0) contains white, and PAT(4) and PAT(8) contain gray and black, respectively. The pattern information is read from **DATA** statements at the end of the listing.

After the column and row labels are assigned and the framed boxes drawn, the program uses a **FOR-NEXT** loop (**FOR** MODE=0 **TO** 3) to call each of the four text modes. P represents the pattern number on which the text string is printed. Notice that between each call to **TEXTMODE** it's set back to default—**TEXTMODE(0)**—before the next iteration of the loop.

T E N

Figure 10-1. *This table shows what will happen when the string* THIS IS A TEST *is printed in different modes on three different backgrounds.*

Different Faces

Text can be displayed with different typefaces by using
TEXTFACE:

CALL TEXTFACE(*face*)

- The different typefaces which can be displayed are specified by *face*. This value forms a bit mask with which to select a set of typeface attributes. Each attribute has an associated value. A value for *face* is the sum of its associated values.
- By default, a value of 0 is assumed—no attributes are given. This creates plain text.
- Attributes, and their associated values in parentheses, are *Bold* (1), *Italic* (2), *Underline* (4), *Outlined* (8), *Shadow* (16), *Condensed Spacing* (32), and *Extended Spacing* (64).

There are 128 possible combinations of type styles. Program 10-2 shows you all these combinations. Simply select the buttons at the top of the *Output* window to toggle the attributes.

Program 10-2. TEXTFACE

```
CLS
DEFINT A-Z
DIM RECT(3),BUTTN(3,6),BUTSEL(6)
CALL TEXTFACE(0):CALL TEXTMODE(1)
ST$="This is a TEST!":FACE=0
RECT(0)=80:RECT(1)=140:RECT(2)=110:RECT(3)=360
FOR BUT=0 TO 6
  BUTTN(0,BUT)=10:BUTTN(1,BUT)=BUT*65+20:BUTTN(2,BUT)=30
:BUTTN(3,BUT)=BUTTN(1,BUT)+60
  CALL FRAMEROUNDRECT(VARPTR(BUTTN(0,BUT)),10,10)
  READ LAB$(BUT)
  CALL MOVETO(BUTTN(1,BUT)+3,BUTTN(2,BUT)-6):PRINT LAB$
(BUT);
NEXT BUT
CALL MOVETO(150,100):PRINT ST$

GETBUTTON:
WHILE MOUSE(0)<1:WEND
X=MOUSE(1):Y=MOUSE(2)
BUT=-1:I=0
WHILE BUT=-1 AND I<7
  IF Y> BUTTN(0,I) AND X>BUTTN(1,I) AND Y<BUTTN(2,I) AND X<B
UTTN(3,I) THEN BUT=I
  I=I+1
WEND
IF BUT=-1 THEN GETBUTTON
IF BUTSEL(BUT)=0 THEN BUTSEL(BUT)=1:FACE=FACE+2^BUT ELSE
 BUTSEL(BUT)=0:FACE=FACE-2^BUT
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUT)),10,10)
CALL ERASERECT(VARPTR(RECT(0)))
CALL TEXTFACE(FACE)
CALL MOVETO(150,100):PRINT ST$
GOTO GETBUTTON
END
DATA " Bold"," Italic","Underln","Outlined","Shadow","Condnse
","Extended"
```

By choosing one or more of the seven buttons at the top of the screen, you can see all 128 typeface combinations. Click on the button to call the attribute, then click on it again to turn it off. Quit the program by selecting *Stop* from the *Run* menu.

Figure 10-2. *Different typefaces can be displayed by using TEXTFACE. All the permutations of the type styles are set by selecting the desired attribute from one of the buttons at the top of the screen. In this case,* Bold, Underln, Outlined, *and* Shadow *are selected.*
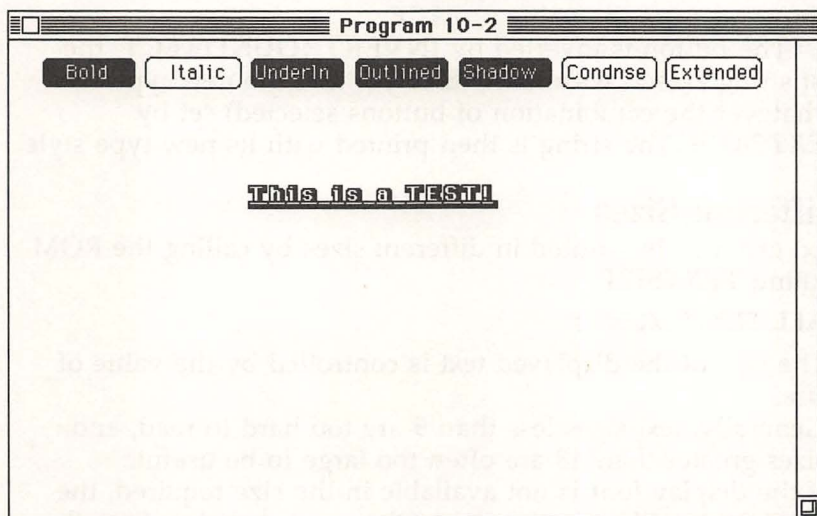
Much of this program is similar to those you've already seen in other chapters. Arrays are defined and the screen display is created, complete with labels, buttons, and the message *This is a TEST!*.

One important thing to note is that **TEXTMODE** uses mode 1 so that the buttons will not be partially erased when they're labeled. **TEXTFACE** is initially set with face 0—plain text.

BUTSEL, the one array to watch for, is a status array containing seven elements to match the seven buttons. If BUTSEL(x) is 0, then the attribute represented by button x is not activated. The opposite is true if BUTSEL(x) is equal to 1.

The main part of the program, GETBUTTON, waits for a press of the mouse button. BUT is the status variable for the button. It's assumed that the program starts with BUT set to −1, meaning no button has been pressed. Pointer locations are recorded once a button press is found, then compared with button locations.

After a button press is monitored, the program determines whether or not to toggle the attribute on or off, depending on what the button's corresponding status array element contains. If the value is 0, then the attribute value must be added to the value of FACE. Otherwise, the associated attribute value is subtracted from the value of FACE.

The button is inverted by **INVERTROUNDRECT**, the test string erased with **ERASERECT,** and the new typeface (whatever the combination of buttons selected) set by **TEXTFACE**. The string is then printed with its new type style.

Different Sizes

Text can also be printed in different sizes by calling the ROM routine **TEXTSIZE**:

CALL TEXTSIZE(*size*)

- The size of the displayed text is controlled by the value of *size*.
- Generally, text sizes less than 9 are too hard to read, and sizes greater than 48 are often too large to be useful.
- If the display font is not available in the size required, the Macintosh will scale the text to the requested size. Scaled fonts can be difficult to read, so it's a good idea to pick a size in an even multiple of an existing font size.

Program 10-3 shows you several sizes of text, again by letting you select buttons at the top of the screen.

Program 10-3. TEXTSIZE

```
CLS
DEFINT A-Z
DIM RECT(3),BUTTN(3,9),SIZE(9)
CALL TEXTFACE(0):CALL TEXTMODE(1):CALL TEXTSIZE(12)
ST$="This is a TEST!":BUTSEL=2
RECT(0)=60:RECT(1)=50:RECT(2)=210:RECT(3)=470
FOR BUT=0 TO 9
```

```
BUTTN(0,BUT)=10:BUTTN(1,BUT)=BUT*40+30:BUTTN(2,BUT)=30
:BUTTN(3,BUT)=BUTTN(1,BUT)+35
  CALL FRAMEROUNDRECT(VARPTR(BUTTN(0,BUT)),10,10)
  READ SIZE(BUT)
  CALL MOVETO(BUTTN(1,BUT)+3,BUTTN(2,BUT)-6):PRINT SIZE(
BUT);
NEXT BUT
CALL MOVETO(70,150):PRINT ST$;
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUTSEL)),10,10)

GETBUTTON:
WHILE MOUSE(0)<1:WEND
X=MOUSE(1):Y=MOUSE(2)
BUT= -1:I=0
WHILE BUT= -1 AND I<10
  IF Y> BUTTN(0,I) AND X>BUTTN(1,I) AND Y<BUTTN(2,I) AND X<B
UTTN(3,I) THEN BUT=I
  I=I+1
WEND
IF BUT= -1 THEN GETBUTTON
IF BUTSEL=BUT THEN GETBUTTON
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUTSEL)),10,10)
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUT)),10,10)
BUTSEL=BUT
CALL ERASERECT(VARPTR(RECT(0)))
CALL TEXTSIZE(SIZE(BUTSEL))
CALL MOVETO(70,150):PRINT ST$;
GOTO GETBUTTON
END
DATA 9,10,12,14,16,18,20,24,36,48
```

Program 10-3 shows you ten possible text sizes. Activate a text size by clicking on one of the buttons at the top of the *Output* window. Choose *Stop* from the *Run* menu to quit the program.

T E N

Figure 10-3. *The size of displayable text is controlled with TEXTSIZE.*


```
━━━━━━━━━━━━━━━━━━━━━ Program 10-3 ━━━━━━━━━━━
  ( 9 )( 10 )( 12 )( 14 )( 16 )( 18 )( 20 )( 24 )( 36 )( 48 )



        This is a TEST!
```
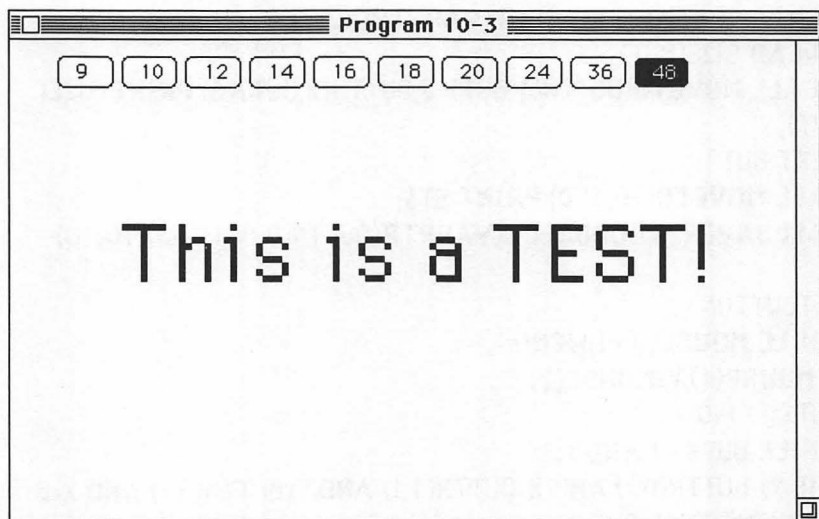
Except for the results of clicking the buttons, this is almost identical to Program 10-2. The display is created and buttons are monitored. Note that initially, **TEXTSIZE** uses 12, for a 12-point type size.

Once a button is selected, after determining which button it is and if it's already chosen (by comparing BUTSEL to BUT), the newly selected button is inverted with **INVERTROUND-RECT**, the test string (ST$) is erased with **ERASERECT**, and the new type size is set by calling **TEXTSIZE**.

Different Fonts

Not only can you change the size of text, but you can also alter the particular font of text to display:

CALL TEXTFONT(*font number*)

- This changes the font used to display text in the *Output* window to the specified *font number*.
- The default font displayed has a font number of 1. This will be either the New York or Geneva font, depending upon the *Finder* and System on the disk. (The Microsoft BASIC disk uses Geneva as the default font. That's the font you see on the *Output* window.)

250

T E N

• The font used in the pull-down menus and dialog boxes is the System font and has a font number of 0. (Chicago is the System font on the Microsoft BASIC 2.0 disk.)
• Fonts, with their font numbers in parentheses, are New York (2), Geneva (3), Monaco (4), Venice (5), London (6), Athens (7), San Francisco (8), Toronto (9), Seattle (10), and Cairo (11).

Program 10-4. TEXTFONT

```
CLS
PRINT "This is the standard application font"
CALL TEXTFONT(0)
PRINT "And this is a sample of the system font."
CALL TEXTFONT(1)
```

Figure 10-4. *TEXTFONT allows you to change the display font.*

```
▤☐▤▤▤▤▤▤▤▤▤▤▤▤▤ Program 10-4 ▤▤▤▤▤▤▤▤▤▤
This is the standard application font
And this is a sample of the system font.
```
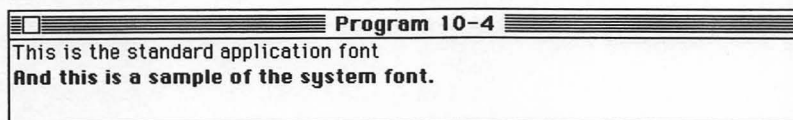
This displays two strings in different fonts. First, the default font 1 will be displayed before the display font is changed to 0 by **TEXTFONT**. The second line thus appears in the System font. The default font is reset with another call to **TEXTFONT**.

Obviously, you can't call a font unless it's present on the disk you're working with. If you're creating an application, for instance, and want to access the Toronto font (font 9), at least one size of that font must be on the disk. (Use the *Font Mover* utility found on your *System Disk* to move additional fonts to your applications disk.)

CHAPTER

11
Software
Tools

11
Software Tools

Utilities that aid program development and subroutines incorporated into a program are often called *software tools*. Examples are such programs as a pattern, cursor, or shape editor, and subroutines for saving and loading data, activating input boxes, and scanning for dialog button selection. The programs in this chapter are utilities using software tools with an emphasis on graphics and Macintosh-unique features. These tools can save considerable time when you create your own graphic programs.

The Pattern Editor

Providing a user interface involves drawing an input screen with a background pattern. Determining the values which define the pattern is time-consuming and involves a lot of hand calculation, which can lead to mistakes. Moreover, you can't get a true idea of what more complex patterns look like until you test them in your program. "The Pattern Editor" lets you quickly create, edit, and save patterns, while providing a view of the pattern and the data defining it. The pattern is edited by toggling bits in the 8 × 8 bit pixel representation or by entering values for any of the four integers of the pattern array.

The input screen in Figure 11-1 has four input areas on the left: pattern array element input boxes, a pixel toggle box, six control buttons, and an input box for filenames. The large easel on the right displays the pattern defined by the pixel toggle box. Initially, the pattern is 1 (white), so the pattern array element input boxes contain zeros and the pixels in the toggle box are all clear, or white. By default, the filename is *Test.pat*. Entering values into the pattern array element input boxes is done by clicking in one of the four. When it turns black, you can type a value from −32768 to 32767. If you select the wrong box, just hit the Return key and the value for that array element returns. Otherwise, your input replaces the

255

ELEVEN

previous value for that element, and both the appropriate toggle boxes and the pattern display box on the right are changed.

Figure 11-1. *"The Pattern Editor" allows you to create, edit, and save patterns.*
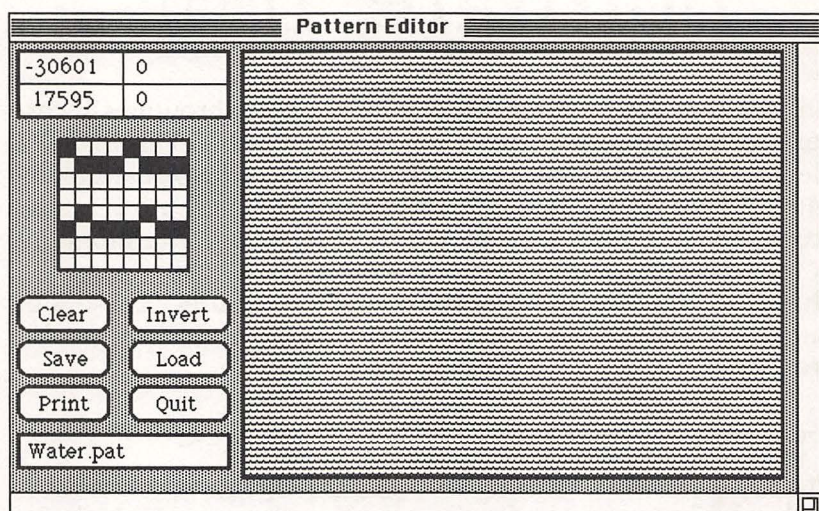
The toggle boxes are inverted whenever one of them is selected with the mouse. Doing so updates the appropriate pattern array element box and pattern display box. The control buttons are activated by clicking within their boundaries. *Clear* resets the pattern to white. *Invert* reverses each of the pixels in the toggle box. *Save* stores the pattern array into the file named by the filename input box and *Load* retrieves the pattern from disk. *Load* checks for the existence of the pattern file and, if found, updates the screen. Otherwise, the filename is changed to *File Not Found*. *Print* performs a screen dump to the printer and *Quit* ends the program.

Program 11-1. The Pattern Editor

```
GOSUB initvars
GOSUB drawscreen
chkm:
GOSUB scanmouse:'scan for mouse activity
IF BUTSEL> -1 THEN GOSUB invertbut:ON BUTSEL+1 GOSUB Clea
rbutton,Invertbutton,Savebutton,Loadbutton,Printbutton,Quitbutt
```

```
on:GOSUB invertbut:GOTO chkm
IF INBOXSEL> -1 THEN ON INBOXSEL+1 GOSUB filenameinbox,val
ueinboxes,valueinboxes,valueinboxes,valueinboxes:GOTO chkm
GOTO chkm

drawbox:
TRECT(0)=RECT(B,0)-1:TRECT(1)=RECT(B,1)-1:TRECT(2)=RECT(B,2
)+2:TRECT(3)=RECT(B,3)+2
CALL ERASERECT(VARPTR(TRECT(0))):CALL PENSIZE(2,2):CA
LL FRAMERECT(VARPTR(TRECT(0))):CALL PENSIZE(1,1)
RETURN

drawbutton:
TRECT(0)=BUTTN(B,0)-1:TRECT(1)=BUTTN(B,1)-1: TRECT(2)=BUTT
N(B,2)+2:TRECT(3)=BUTTN(B,3)+2
CALL ERASEROUNDRECT(VARPTR(TRECT(0)),15,15):CALL PEN
SIZE(2,2):CALL FRAMEROUNDRECT(VARPTR(TRECT(0)),15,15)
:CALL PENSIZE(1,1)
CALL MOVETO(TRECT(1)+BUTTN(B,4),TRECT(2)-7):PRINT BUTTN
$(B);
RETURN

drawinputbox:
TRECT(0)=INBOX(B,0)-1:TRECT(1)=INBOX(B,1)-1: TRECT(2)=INBOX
(B,2)+2:TRECT(3)=INBOX(B,3)+2
CALL ERASERECT(VARPTR(TRECT(0))):CALL PENSIZE(2,2):CA
LL FRAMERECT(VARPTR(TRECT(0))):CALL PENSIZE(1,1)
RETURN

clearinbox:
TRECT(0)=INBOX(B,0)+1:TRECT(1)=INBOX(B,1)+1: TRECT(2)=INBOX
(B,2):TRECT(3)=INBOX(B,3)
CALL ERASERECT(VARPTR(TRECT(0)))
RETURN

printinbox: 'b with st$
CALL MOVETO(INBOX(B,1)+4,INBOX(B,2)-4):PRINT ST$;
```

```
RETURN

getinput: 'for inbox default in zt,zt$
EDIT FIELD 1,zt$,(INBOX(B,1)+2,INBOX(B,0)+1)-(INBOX(B,3)-1,IN
BOX(B,2)-1)

idle:
dact=DIALOG(0)
IF dact<>6 THEN idle
ST$=EDIT$(1):ST=VAL(ST$)
EDIT FIELD CLOSE 1
GOSUB drawinputbox
IF LEN(ST$)=0 THEN ST$=ZT$:ST=ZT
GOSUB printinbox
RETURN

invertbut:
TRECT(0)=BUTTN(BUTSEL,0)-1:TRECT(1)=BUTTN(BUTSEL,1)-1: TR
ECT(2)=BUTTN(BUTSEL,2)+2:TRECT(3)=BUTTN(BUTSEL,3)+2
CALL INVERTROUNDRECT(VARPTR(TRECT(0)),15,15)
RETURN

redrawpattern:
TRECT(0)=RECT(3,0)+1:TRECT(1)=RECT(3,1)+1:TRECT(2)=RECT(3,2
):TRECT(3)=RECT(3,3):CALL FILLRECT(VARPTR(TRECT(0)),VAR
PTR(PATTERN(0)))
RETURN

scanmouse: 'activity
IF MOUSE(0)<>-1 THEN scanmouse
MX=MOUSE(1):MY=MOUSE(2):BUTSEL= -1:GOSUB checkbutpress:I
F BUTSEL> -1 THEN RETURN:'check for button press
INBOXSEL= -1:GOSUB checkibox:IF INBOXSEL> -1 THEN RETURN:
  check fox input
IF (MX>RECT(2,1)) AND (MX<RECT(2,3)) AND (MY>RECT(2,0)) AND
(MY<RECT(2,2)) THEN GOSUB bitboxtoggle:GOTO scanmouse:'sca
n for bit box toggle
```

```
RETURN

printvalues:
FOR B=1 TO 4:GOSUB clearinbox:ST$=STR$(PATTERN(B-1)):GOS
UB printinbox:NEXT B:'clear and print values
RETURN

updatebitbox: 'for all patterns
FOR I=0 TO 3:GOSUB updatepatbit:NEXT I
RETURN

updatepatbit: 'box pattern(i)
IF (PATTERN(I) AND &H8000) THEN R=I*2:C=0:GOSUB setbit ELS
E R=I*2:C=0:GOSUB clearbit
FOR J=14 TO 8 STEP -1:IF (PATTERN(I) AND 2^J) THEN R=I*2:C=
15-J:GOSUB setbit ELSE R=I*2:C=15-J:GOSUB clearbit
NEXT J
FOR J=7 TO 0 STEP -1:IF (PATTERN(I) AND 2^J) THEN R=I*2+1:C
=7-J:GOSUB setbit ELSE R=I*2+1:C=7-J:GOSUB clearbit
NEXT J
RETURN

clearbit: 'in bit box
TRECT(0)=RECT(2,0)+1+10*R:TRECT(1)=RECT(2,1)+1+10*C:TRECT
(2)=TRECT(0)+9:TRECT(3)=TRECT(1)+9:CALL FILLRECT(VARPTR
(TRECT(0)),VARPTR(PAT(0)))
RETURN

setbit: 'in bit box
TRECT(0)=RECT(2,0)+1+10*R:TRECT(1)=RECT(2,1)+1+10*C:TRECT
(2)=TRECT(0)+9:TRECT(3)=TRECT(1)+9:CALL FILLRECT(VARPTR
(TRECT(0)),VARPTR(PAT(8)))
RETURN

checkbutpress:
B=0:WHILE (BUTSEL= -1) AND (B<=MAXBUTTON): IF (MX>BUTTN(B
,1)) AND (MX<BUTTN(B,3)) AND (MY>BUTTN(B,0)) AND (MY<BUTTN
(B,2)) THEN BUTSEL=B
```

```
B=B+1:WEND
RETURN
```

checkibox: 'input
```
B=0:WHILE (INBOXSEL= -1) AND (B<=MAXINBOX): IF (MX>INBOX(B,
1)) AND (MX<INBOX(B,3)) AND (MY>INBOX(B,0)) AND (MY<INBOX(B,
2)) THEN INBOXSEL=B
B=B+1:WEND
RETURN
```

Clearbutton:
```
FOR I=0 TO 3:PATTERN(I)=0:NEXT I:GOSUB printvalues:GOSUB d
rawbitbox:GOSUB redrawpattern:'update pattern,bitbox and valu
es
RETURN
```

Invertbutton:
```
FOR I=0 TO 3:PATTERN(I)=(PATTERN(I) XOR &HFFFF):NEXT I:GOS
UB printvalues:GOSUB updatebitbox:GOSUB redrawpattern
RETURN
```

Savebutton:
```
OPEN "O",#1,FILENAME$:FOR I=0 TO 3:PRINT#1,PATTERN(I):NEX
T I:CLOSE #1
RETURN
```

Loadbutton:
```
ON ERROR GOTO chkerr
OPEN "I",#1,FILENAME$:FOR I=0 TO 3:INPUT#1,PATTERN(I):NEXT
I:CLOSE #1:GOSUB updatebitbox:GOSUB printvalues:GOSUB red
rawpattern
```

chkerr:
```
IF ERR=53 THEN FILENAME$="File Not Found":B=0:GOSUB clearin
box:ST$=FILENAME$:GOSUB printinbox:RESUME lbret
lbret:
RETURN
```

260

ELEVEN

Printbutton:
'LCOPY
RETURN

Quitbutton:
END

filenameinbox:
ZT=0:ZT$=FILENAME$:B=0:GOSUB getinput:FILENAME$=ST$
RETURN

valueinboxes:
ZT=PATTERN(INBOXSEL-1):ZT$=STR$(ZT):B=INBOXSEL:GOSUB get
input:PATTERN(INBOXSEL-1)=ST:I=INBOXSEL-1:GOSUB updatepatb
it:GOSUB redrawpattern
RETURN

bitboxtoggle:
C=(MX-RECT(2,1))\10:R=(MY-RECT(2,0))\10:PAT=R\2: I=15-(R-2*
PAT)*8-C:B=PAT+1:
J= &H8000:IF I<15 THEN J=2^I
PATTERN(PAT)=PATTERN(PAT) XOR J:GOSUB clearinbox:ST$=STR
$(PATTERN(PAT)):GOSUB printinbox
TRECT(0)=RECT(2,0)+1+10*R:TRECT(1)=RECT(2,1)+1+10*C:TRECT
(2)=TRECT(0)+9:TRECT(3)=TRECT(1)+9:CALL INVERTRECT(VARP
TR(TRECT(0))):GOSUB redrawpattern
RETURN

drawscreen:
TRECT(0)=RECT(0,0):TRECT(1)=RECT(0,1):TRECT(2)=RECT(0,2):TR
ECT(3)=RECT(0,3):CALL FILLRECT(VARPTR(TRECT(0)),VARPTR
(PAT(4)))
FOR B=1 TO 3:GOSUB drawbox:NEXT B:'draw boxes
CALL MOVETO(RECT(1,1),RECT(1,0)+17):CALL LINE(130,0):CAL
L MOVE (0,1):CALL LINE(-130,0):CALL MOVETO(RECT(1,1)+65,R
ECT(1,0)):CALL LINE(0,35):CALL MOVE(1,0):CALL LINE(0,-35)
GOSUB drawbitbox
FOR B=0 TO 5:GOSUB drawbutton:NEXT B:'draw buttons

B=0:**GOSUB** drawinputbox:'draw file name input box
GOSUB printvalues: print pattern values
ST$=FILENAME$:B=0:**GOSUB** clearinbox:**GOSUB** printinbox
RETURN

drawbitbox:
B=2:**GOSUB** drawbox:**FOR** I=10 **TO** 70 **STEP** 10:**CALL MOVETO**(RE
CT(2,1)+I,RECT(2,0)):**CALL LINE**(0,80):**CALL MOVETO**(RECT(2,1)
,RECT(2,0)+I):**CALL LINE**(80,0):**NEXT** I
RETURN

initvars:
CLS:DEFINT A-Z:**CALL TEXTMODE**(1):**DIM** PATTERN(3),PAT(11),
TRECT(3),BUTTN(5,4),BUTTN$(5),INBOX(4,3),RECT(3,3)
FILENAME$="Test.PAT":MAXBUTTON=5:MAXINBOX=4
FOR I=0 **TO** 11:**READ** PAT(I):**NEXT** I:'backgrounds
FOR I=0 **TO** 3:**FOR** J=0 **TO** 3:**READ** RECT(I,J):**NEXT** J,I
FOR I=0 **TO** 5:**FOR** J=0 **TO** 4:**READ** BUTTN(I,J):**NEXT** J:**READ** BUTT
N$(I):**NEXT** I
FOR I=0 **TO** 4:**FOR** J=0 **TO** 3:**READ** INBOX(I,J):**NEXT** J,I
RETURN

DATA 0,0,0,0,4420,4420,4420,4420,-1,-1,-1,-1:'patterns
DATA 0,0,342,512:'screen
DATA 5,5,40,136:'values
DATA 60,30,140,110:'bitbox
DATA 5,145,270,480:'pattern box
DATA 158,5,178,60,14,Clear
DATA 158,75,178,135,10,Invert
DATA 186,5,206,60,16,Save
DATA 186,75,206,135,16,Load
DATA 214,5,234,60,14,Print
DATA 214,75,234,135,17,Quit
DATA 244,5,261,135:'File name input
DATA 5,5,22,70:'values 0-3
DATA 5,71,22,136
DATA 23,5,40,70
DATA 23,71,40,136

E L E V E N

This Pattern Editor program is divided into four major sections—input scanning, input processing subroutines, screen and variable initialization, and program data. The most frequently used sections generally come first to optimize execution speed.

The first task is to access the subroutine *initvars*, which defines and initializes all variables. That routine uses **TEXTMODE** to make all text display nondestructive, as well as defines the required arrays. PATTERN(3) is the pattern array manipulated by the program, and PAT(11) contains three patterns used within the program. TRECT(3) is a temporary rectangle array to draw the input screen. The six control buttons are represented by two arrays—BUTTN(5,4), which defines six rectangle arrays for the buttons, and BUTTN$(5), which holds the button labels. INBOX(4,3) defines five rectangle arrays for the array element input boxes and the filename box. RECT(3,3) contains four rectangle arrays for the screen, the border around the element boxes, the border around the toggle boxes, and display box.

FILENAME$ is initialized along with MAXBUTTON and MAXINBOX. Before the subroutine returns, a number of **FOR-NEXT** loops read pattern and rectangle information from **DATA** statements at the end of the program.

Once the variables are initialized, the *drawscreen* subroutine is called. This creates the input screen you see on your Macintosh. In that routine, gray fills the background (**FILLRECT**) before the *drawbox* routine is accessed. The program uses *drawbox* to draw the element, toggle, display, and file input boxes. **FILLRECT** and **FRAMERECT** prepare the boxes, and **PENSIZE** draws two-pixel-thick borders around them.

Back in the *drawscreen* subroutine, several **CALL LINE** statements split the element box into four separate boxes. Next, the *drawbitbox* subroutine is called to draw the grid work within the toggle box. Buttons are created by accessing the *drawbutton* routine; *drawinputbox* creates the filename box at the bottom of the screen; *printvalues* places the pattern array elements in their respective places; and *clearinbox* clears the input box and places a value there. Look at each of these subroutines to see what commands are used for each task.

The main portion of the program is the loop at the beginning of the listing; *scanmouse* is called to wait for mouse input.

E L E V E N

When found, it determines in which of the four areas, if any, the input was made. Glance at the *scanmouse* subroutine. The statement **IF MOUSE(0)<> −1 THEN** scanmouse waits for a mouse click, ignoring double and triple clicks. MX and MY retain the horizontal and vertical positions of the click and set BUTSEL to −1 before calling *checkbutpress*, the button select detection subroutine.

In that part of the program, note that if BUTSEL changes, it contains the button number selected. Checks are conducted to insure that MX and MY are within the boundary of button *B*. If so, BUTSEL is updated. Back in *scanmouse*, another line checks INBOXSEL and calls *checkibox* to determine if an input box was selected. An **IF-THEN** statement in *scanmouse* compares the click position with the bit toggle box and, if a match is made, calls the *bitboxtoggle* subroutine to update the pattern information and screen.

The *bitboxtoggle* routine calulates C and R, the column and row of the pixel within the toggle box, and determines PAT, the array element, and I, the bit (0–15) to toggle. The binary mask needed to toggle the pixel, J, is designated, after which the bit **XOR**ing the element with the mask is toggled. The input box for that element is also updated. Finally, *redrawpattern* is called to redraw the pattern display box using **FILLRECT**.

If one of the buttons is selected, the **IF BUTSEL> −1**... statement at the beginning of the program calls *invertbut* to invert the button while the process indicated by the button is performed. Depending on the button selected, one of a number of subroutines is performed.

Clearbutton sets the elements to zero, and calls *printvalues* to change the element input boxes, *drawbitbox* to redraw the toggle box, and *redrawpatterns* to update the pattern display box.

Invertbutton inverts each element by being **XOR**ed with &HFFFF before the toggle box is updated by calling *updatebitbox*. In this subroutine, the element number (I) is set and the next routine (*updatepatbit*) is accessed to display the bits of that element. Bit positions are calculated and one of two additional subroutines, either *setbit* or *clearbit*, is called to draw the squares black or white, respectively.

Savebutton opens the file, writes the pattern data to it, and closes it. *Loadbutton* works much the same way, with the addi-

E L E V E N

tion of an error-checking line added in case the file doesn't exist.

Printbutton performs an **LCOPY**, and **END** executes when *Quitbutton* is called.

When the filename input box is selected, *filenameinbox* is called; *valueinboxes* is called if another input box is selected. The initial value and string of the filename input box is stored in ZT and ZT$ prior to calling a general-purpose input processing routine (*getinput*).

You'll notice some new commands used in this subroutine. **EDIT FIELD** allows you to perform text-editing operations on the input, such as using the Backspace key to delete. **EDIT FIELD** opens an input box to edit, and the program loops around *idle* until **DIALOG(0)** returns a value of 6, indicating that a Return was typed. The input is stored in ST$ with **EDIT$** and its numeric equivalent in ST. The edit field is closed before the input box is redrawn (necessary since closing the field erases that portion of the screen).

A number of the subroutines, such as those which load and save data from and to disk, can be pulled from a program like this and used in your own applications.

The Cursor Editor

"The Cursor Editor," Program 11-2, allows you to design and experiment with mouse pointers, which can then be saved to disk and used by your own BASIC programs. It minimizes this aspect of program development by calculating the cursor array data as you edit.

Figure 11-2 presents the input screen and its four input areas—input boxes for cursor array data and a filename, control buttons, and cursor and mask pixel toggle boxes. There's a white, gray, and black cursor text area between the two toggle boxes. The figure shows a cursor being tested on the black background. Values are entered into the input boxes in the same manner as with the Pattern Editor since the same subroutines are used. The toggle boxes and buttons are also used the same way.

You'll see 24 buttons, some with the same label. Those on the right affect the mask, while those on the left affect the cursor. *Flip H* and *Flip V* transpose their respective images

ELEVEN

Figure 11-2. *This utility creates and edits mouse pointers. The two columns of numbers represent the cursor data required for its definition.*
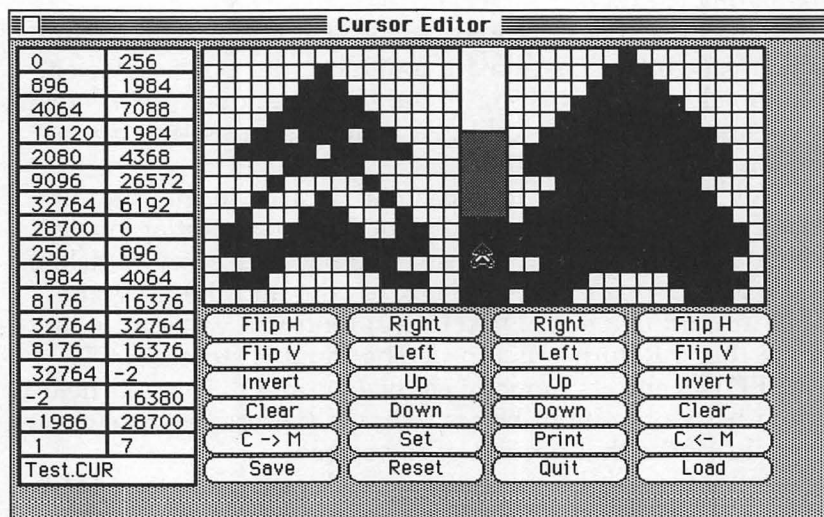
horizontally and vertically. *Invert* reverses the pixels of an image and *Clear* makes it white. *Right, Left, Up,* and *Down* shift the images one pixel in the respective direction. *C->M* copies the image of the cursor to the mask, while *M<-C* does the opposite. *Save* stores the cursor array data in the file specified by the bottommost input box and *Load* retrieves this data for further editing. *Set* changes the cursor to that defined by the cursor array data, while *Reset* returns the pointer to an arrow. *Print* dumps the screen to the printer and *Quit* terminates the program.

Program 11-2. The Cursor Editor

```
GOSUB initvars
GOSUB drawscreen

getm:
WHILE MOUSE(0)<1:WEND
MX=MOUSE(1):MY=MOUSE(2)
RECTSEL= -1:GOSUB chkrects:IF RECTSEL> -1 THEN ON RECTSEL
+1 GOSUB chkinboxes,chktogboxes,chktogboxes,chkbuttons
```

```
GOTO getm

drawrects:
RECT(0,B)=RECT(0,B)-1:RECT(1,B)=RECT(1,B)-1:RECT(2,B)=RECT(2
,B)+1:RECT(3,B)=RECT(3,B)+1
CALL PENSIZE(2,2):CALL ERASERECT(VARPTR(RECT(0,B))):CA
LL FRAMERECT(VARPTR(RECT(0,B))):CALL PENSIZE(1,1)
RECT(0,B)=RECT(0,B)+1:RECT(1,B)=RECT(1,B)+1:RECT(2,B)=RECT(2
,B)-1:RECT(3,B)=RECT(3,B)-1
RETURN


drawbutton:
CALL ERASEROUNDRECT(VARPTR(BUTTN(0,B)),15,15):CALL FR
AMEROUNDRECT(VARPTR(BUTTN(0,B)),15,15)
CALL MOVETO(BUTTN(1,B)+BUTTN(4,B),BUTTN(2,B)-5):PRINT BU
TTN$(B);
RETURN


drawinbox: 'b
CALL ERASERECT(VARPTR(INBOX(0,B))):CALL FRAMERECT(VA
RPTR(INBOX(0,B)))
RETURN


printinbox: 'b with st$
CALL MOVETO(INBOX(1,B)+4,INBOX(0,B)+13):PRINT ST$;
RETURN


getinput: 'for inbox b with default in zt,zt$
EDIT FIELD 1,zt$,(INBOX(1,B)+2,INBOX(0,B)+1)-(INBOX(3,B)-2,IN
BOX(2,B)-2)

idle:
dact=DIALOG(0)
IF dact<>6 THEN idle
ST$=EDIT$(1):ST=VAL(ST$)
EDIT FIELD CLOSE 1
GOSUB drawinbox
IF LEN(ST$)=0 THEN ST$=ZT$:ST=ZT
```

```
GOSUB printinbox
RETURN

invertbutton:
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUTSEL)),15,15)
RETURN

setlabel:
CALL MOVETO(1000,20):PRINT BUTTN$(B);:CALL GETPEN(VAR
PTR(PL(0))):BUTTN(4,B)=((BUTTN(3,B)-BUTTN(1,B))-(PL(1)-1000
))\2
RETURN

chkrects:
B=0:WHILE (RECTSEL= -1) AND (B<=MAXRECT)
IF (MX>RECT(1,B)) AND (MX<RECT(3,B)) AND (MY>RECT(0,B)) AND
 (MY<RECT(2,B)) THEN RECTSEL=B
B=B+1:WEND
RETURN

chkinboxes:
INBOXSEL=-1:B=(MY-7)\15:WHILE (INBOXSEL= -1) AND (B<=MAXIN
BOX)
IF (MX>INBOX(1,B)) AND (MX<INBOX(3,B)) AND (MY>INBOX(0,B)) AN
D (MY<INBOX(2,B)) THEN INBOXSEL=B
B=B+1:WEND
IF INBOXSEL= -1 THEN RETURN
IF INBOXSEL=34 THEN ZT$=FILENAME$:ZT=0:B=34:GOSUB getinpu
t:FILENAME$=ST$:RETURN
IF INBOXSEL<32 THEN B=INBOXSEL:ZT$=STR$(CURS(B)):ZT=CURS(
B):GOSUB getinput:CURS(B)=ST:GOSUB redrawrow:RETURN
CALL INVERTRECT(VARPTR(INBOX(0,32))):CALL INVERTRECT
(VARPTR(INBOX(0,33)))
WHILE MOUSE(0)<1:WEND:MX=MOUSE(1):MY=MOUSE(2)
RECTSEL= -1:GOSUB chkrects:IF RECTSEL=1 OR RECTSEL=2 THEN
 CURS(33)=((MX-(RECTSEL-1)*190)-121)\10:CURS(32)=(MY-6)\1
0
ST$=STR$(CURS(32)):B=32:GOSUB drawinbox:GOSUB printinbox:
ST$=STR$(CURS(33)):B=33:GOSUB drawinbox:GOSUB printinbox
```

RETURN

chktogboxes:
X=((MX-(RECTSEL-1)*190)-121)\10:Y=(MY-6)\10
Y=15-X:IF Y=15 THEN MASK=-32768! ELSE MASK =2^Y
B=Y+(RECTSEL-1)*16:CURS(B)=CURS(B) XOR MASK
GOSUB drawinbox:ST$=STR$(CURS(B)):GOSUB printinbox
TRECT(0)=Y*10+6:TRECT(1)=X*10+121+190*(RECTSEL-1):TRECT(
2)=TRECT(0)+9:TRECT(3)=TRECT(1)+9:CALL INVERTRECT(VARP
TR(TRECT(0)))
RETURN

chkbuttons:
BUTSEL=-1:B=0:WHILE (BUTSEL= -1) AND (B<=MAXBUTTON)
IF (MX>BUTTN(1,B)) AND (MX<BUTTN(3,B)) AND (MY>BUTTN(0,B))
AND (MY<BUTTN(2,B)) THEN BUTSEL=B
B=B+1:WEND
IF BUTSEL> -1 THEN GOSUB invertbutton
ON BUTSEL+1 GOSUB fliphc,flipvc,invertc,clrc,transc,savebutto
n,rightc,leftc,upc,downc,setbutton,resetbutton,rightm,leftm,upm
,downm,printbutton,quit,fliphm,flipvm,invertm,clrm,transm,load
button:GOSUB invertbutton
RETURN

redrawrow: 'b
PIC(2)=CURS(B)
X= (B\16)*190+120:Y=(B MOD 16) *10+5:PUT (X,Y)-(X+160,Y+10)
,PIC,PSET
LINE (X,Y) - STEP (160,0):LINE (X,Y+10) - STEP (160,0)
FOR I=X TO X+160 STEP 10:LINE (I,Y)-STEP (0,10):NEXT I
RETURN

fliph: 'horizontally
FOR J=OS TO OS+7:SWAP CURS(J),CURS(2*OS+15-J):B=J:GOSUB
 redrawrow:B=2*OS+15-J:GOSUB redrawrow:NEXT J
FOR B=OS TO OS+15:GOSUB drawinbox:ST$=STR$(CURS(B)):GOS
UB printinbox:NEXT B

RETURN

fliphc:
OS=0:**GOSUB** fliph
RETURN

fliphm:
OS=16:**GOSUB** fliph
RETURN

flipv: 'vertically
FOR J=OS+0 **TO** OS+15:TV=0:**IF** -32768! **AND** CURS(J) **THEN** TV=1
FOR K=14 **TO** 1 **STEP** -1:MASK = 2^K:**IF** MASK **AND** CURS(J) **THEN**
TV=TV+2^(15-K)
NEXT K
IF 1 **AND** CURS(J) **THEN** TV=TV - 32768!
CURS(J)=TV:B=J:**GOSUB** redrawrow:**NEXT** J
FOR B=OS **TO** OS+15:**GOSUB** drawinbox:ST$=**STR$**(CURS(B)):**GOS**
UB printinbox:**NEXT** B
RETURN

flipvc:
OS=0:**GOSUB** flipv
RETURN

flipvm:
OS=16:**GOSUB** flipv
RETURN

invert:
FOR J=OS **TO** OS+15:CURS(J)=CURS(J) **XOR** &HFFFF:B=J:**GOSUB** re
drawrow:**NEXT** J
FOR B=OS **TO** OS+15:**GOSUB** drawinbox:ST$=**STR$**(CURS(B)):**GOS**
UB printinbox:**NEXT** B
RETURN

invertc:
OS=0:**GOSUB** invert

RETURN

invertm:
OS=16:GOSUB invert
RETURN

clr:
FOR J=OS TO OS+15:CURS(J)=0:NEXT J
FOR B=OS TO OS+15:GOSUB drawinbox:ST$=STR$(CURS(B)):GOS
UB printinbox:NEXT B
RETURN

clrc:
OS=0:GOSUB clr:GOSUB drawcm
RETURN

clrm:
OS=16:GOSUB clr:OS=1:GOSUB drawcm
RETURN

transfer:
FOR J=0 TO 15:CURS(TS+J)=CURS(OS+J):B=TS+J:GOSUB redrawro
w:NEXT J
FOR B=TS TO TS+15:GOSUB drawinbox:ST$=STR$(CURS(B)):GOS
UB printinbox:NEXT B
RETURN

transc:
OS=0:TS=16:GOSUB transfer
RETURN

transm:
OS=16:TS=0:GOSUB transfer
RETURN

savebutton:
OPEN "O",#1,FILENAME$:FOR I=0 TO 33:WRITE#1,CURS(I):NEXT I
:CLOSE#1

RETURN

loadbutton:
```
ON ERROR GOTO chkerr
OPEN "I",#1,FILENAME$:FOR I=0 TO 33:INPUT #1,CURS(I):NEXT I
:CLOSE #1:FOR B=0 TO 31:GOSUB redrawrow:NEXT B:FOR B=0 T
O 33:GOSUB drawinbox:ST$=STR$(CURS(B)):GOSUB printinbox:N
EXT B:RETURN
```

chkerr:
```
IF ERR=53 THEN FILENAME$="File not found":B=34:GOSUB drawi
nbox:ST$=FILENAME$:GOSUB printinbox:RESUME lbret
```

lbret:
RETURN

printbutton:
```
LCOPY
RETURN
```

quit: 'button
```
CALL INITCURSOR
END
```

rightbutton:
```
FOR J=OS TO OS+15:TV=0:IF CURS(J) AND 1 THEN TV= -32768!
IF CURS(J) AND -32768! THEN TV=TV+16384:CURS(J)=CURS(J) X
OR -32768!
CURS(J)=(CURS(J)\2) OR TV:B=J:GOSUB redrawrow:NEXT J
FOR B=OS TO OS+15:GOSUB drawinbox:ST$=STR$(CURS(B)):GOS
UB printinbox:NEXT B
RETURN
```

rightc:
```
OS=0:GOSUB rightbutton
RETURN
```

rightm:

272

```
OS=16:GOSUB rightbutton
RETURN

leftbutton:
FOR J=OS TO OS+15:TV=0:IF CURS(J) AND -32768! THEN TV=1:C
URS(J)=CURS(J) XOR -32768!
IF CURS(J) AND 16384 THEN TV=TV-32768!:CURS(J)=CURS(J) XO
R 16384
CURS(J)=(CURS(J)*2) OR TV:B=J:GOSUB redrawrow:NEXT J
FOR B=OS TO OS+15:GOSUB drawinbox:ST$=STR$(CURS(B)):GOS
UB printinbox:NEXT B
RETURN

leftc:
OS=0:GOSUB leftbutton
RETURN

leftm:
OS=16:GOSUB leftbutton
RETURN

upbutton:
TV=CURS(OS):FOR J=OS+1 TO OS+15:CURS(J-1)=CURS(J):NEXT J:C
URS(OS+15)=TV
FOR B=OS TO OS+15:GOSUB redrawrow:GOSUB drawinbox:ST$=S
TR$(CURS(B)):GOSUB printinbox:NEXT B
RETURN

upc:
OS=0:GOSUB upbutton
RETURN

upm:
OS=16:GOSUB upbutton
RETURN

downbutton:
TV=CURS(OS+15):FOR J=OS+14 TO OS STEP -1:CURS(J+1)=CURS(
J):NEXT J:CURS(OS)=TV
```

273

```
FOR B=OS TO OS+15:GOSUB redrawrow:GOSUB drawinbox:ST$=S
TR$(CURS(B)):GOSUB printinbox:NEXT B
RETURN

downc:
OS=0:GOSUB downbutton
RETURN

downm:
OS=16:GOSUB downbutton
RETURN

setbutton:
CALL SETCURSOR(VARPTR(CURS(0)))
RETURN

resetbutton:
CALL INITCURSOR
RETURN

drawscreen:
CALL BACKPAT(VARPTR(PAT(4))):CLS:CALL BACKPAT(VARP
TR(PAT(0)))
FOR B=0 TO MAXRECT-1:GOSUB drawrects:NEXT B:'drawrects
FOR B=0 TO MAXINBOX:GOSUB drawinbox:NEXT B:'draw inboxes
OS=0:GOSUB drawcm:OS=1:GOSUB drawcm
FOR B=0 TO MAXBUTTON:GOSUB setlabel:GOSUB drawbutton:NEX
T B
ST$=FILENAME$:B=34:GOSUB printinbox
FOR B=0 TO 33:ST$=STR$(CURS(B)):GOSUB printinbox:NEXT B
TRECT(0)=4:TRECT(1)=281:TRECT(2)=58:TRECT(3)=310:CALL FIL
LRECT(VARPTR(TRECT(0)),VARPTR(PAT(0))):CALL FRAMEREC
T(VARPTR(TRECT(0)))
TRECT(0)=58:TRECT(2)=112:CALL FILLRECT(VARPTR(TRECT(0)
),VARPTR(PAT(8))):CALL FRAMERECT(VARPTR(TRECT(0)))
TRECT(0)=112:TRECT(2)=167:CALL FILLRECT(VARPTR(TRECT(0
)),VARPTR(PAT(12))):CALL FRAMERECT(VARPTR(TRECT(0)))
```

RETURN

drawcm: 'cursor and mask box
B=1+OS:**GOSUB** drawrects
FOR I=1 **TO** 15:**LINE** (OS*190+120+I*10,5) - **STEP** (0,160):**LINE**
(OS*190+120,5+I*10) - **STEP** (160,0):**NEXT** I
RETURN

initvars:
CLS:DEFINT A-Z:**CALL TEXTMODE**(1):**CALL TEXTSIZE**(12):MAXR
ECT=3:MAXBUTTON=23:MAXINBOX=34
DIM CURS(33),PAT(15),TRECT(3),RECT(3,MAXRECT),BUTTN(4,MAX
BUTTON),BUTTN$(MAXBUTTON),INBOX(3,MAXINBOX),PL(1),PIC(2)
FILENAME$="Test.CUR":PIC(0)=16:PIC(1)=1
FOR I=0 **TO** 15:**READ** PAT(I):**NEXT** I
FOR I=0 **TO** 3:**FOR** J=0 **TO** 3:**READ** RECT(J,I):**NEXT** J,I
FOR I=0 **TO** MAXINBOX-1:INBOX(0,I)=5+(I\2)*15:INBOX(1,I)=5+(I-(
I\2)*2)*55:INBOX(2,I)=INBOX(0,I)+15:INBOX(3,I)=INBOX(1,I)+55:**N
EXT** I
FOR I=0 **TO** 3:**READ** INBOX(I,MAXINBOX):**NEXT** I
FOR I=0 **TO** MAXBUTTON:BUTTN(0,I)=170+(I **MOD** 6)*18:BUTTN(1,
I)=(I\6)*90+120:BUTTN(2,I)=BUTTN(0,I)+17:BUTTN(3,I)=BUTTN(1,
I)+85:**READ** BUTTN$(I):**NEXT** I
RETURN

DATA 0,0,0,0,4420,4420,4420,4420,21930,21930,21930,21930
,-1,-1,-1,-1:'patterns
DATA 5,5,278,115:'input boxes
DATA 5,120,166,281:'cursor
DATA 5,310,166,471:'mask
DATA 170,120,277,475:'buttons
DATA 260,5,278,115:'filename inbox
DATA "Flip H","Flip V","Invert","Clear","C -> M","Save"
DATA "Right","Left","Up","Down","Set","Reset"
DATA "Right","Left","Up","Down","Print","Quit"
DATA "Flip H","Flip V","Invert","Clear","C <- M","Load"

E L E V E N

Program 11-2 is divided into variable and screen initialization, input scanning, support routines, and data. Like the Pattern Editor, this program first initializes variables, draws the screen, and then loops through a series of input processing routines to determine what action to take. Instead of going into every detail of the program, new subroutines, software tools, and programming techniques are highlighted.

Because of the large number of inputs in this program, there would be a long pause after each mouse click while the program compares each input box, button, and toggle box with the location of the click. By decomposing the input areas into four zones with rectangles, the program quickly evaluates the nature of the input and ignores extraneous inputs. Therefore, along with BUTTN(4,MAXBUTTON) and INBOX(3, MAXINBOX), RECT(3,MAXRECT) represents a list of MAXRECT rectangle arrays bounding the input areas. The subroutine *chkrects* uses RECTSEL = −1 to state that no input area was selected. The mouse position is compared with all input areas just as it would be compared with all button or input box locations, and the selected input region is recorded in RECTSEL prior to exiting the subroutine. A line near the start of the program calls this routine and performs a subroutine from the **ON-GOSUB** list (*chkinboxes, chktogboxes,* and *chkbuttons*) depending on the value in RECTSEL.

A method of centering button labels is provided with the *setlabel* subroutine. The pen is positioned outside the visible screen area with **MOVETO** and prints the label of button *B*. **GETPEN** returns the new pen location in array PL(1). Element PL(1) yields the horizontal position needed to calculate the string width and the label offset stored in BUTTON(4,B).

Given the large number of input boxes and their geometric ordering, the input box scanning subroutine (*chkinboxes*) is modified so that its first line starts scanning at box B, based on the vertical mouse position. This speeds the scanning process since the input box number is estimated, requiring fewer comparisons to find the one selected.

Redrawing toggle boxes is a slow process unless you use the correlation between the binary representation of integers and block graphics drawn with **PUT**. Each element in the cursor array is an integer that defines a row of 16 pixels of the cursor or mask. If you define a block graphic that's 1 pixel high, 16 pixels wide, and the data is the cursor array element you wish to plot, it's done with **PUT**. The first line of *redrawrow* records the data with the height and width previously set when variables were initialized. The second line calculates the point to draw the row of magnified pixels for the toggle boxes. **PSET** mode destroys what was previously drawn so the remaining lines in the routine must redraw the grid.

Flipping images horizontally (top to bottom) is just a matter of exchanging array elements with **SWAP**, but vertical flipping (left to right) is a more complex process. Each bit of each row of the image must be tested with a mask (via **AND**) that is defined as a power of two, with the exception of the fifteenth bit which has the value of -32768. The existence of a bit results in the creation of another mask (preflipped) that's added to a temporary variable (via **XOR**) which replaces the row being flipped after each pixel is checked.

Shifting images up and down is performed by rotating elements of the data array. However, left and right shifting requires bit shifting. The algorithm for a left shift with wraparound is trivial in machine language, but requires four steps in BASIC. First, the high bit of the data must be checked (as in the second line of the subroutine *leftbutton*) with CURS(J) **AND** -32768. If true, the carry, TV, is 1, and the high bit must be striped with **XOR**. The second highest bit is similarly checked by a comparison with 16384, which, if found, is stripped from the image and a mask with a preshifted value of -32768 is combined with the carry. The data is multiplied by 2 to shift the remaining bits prior to the addition of the mask and carry (via **OR**). Look at Figure 11-3, which illustrates the left and right shifting process.

Figure 11-3. *BASIC commands can be used to pixel-shift block graphics using this procedure.*

Left and Right Shifting Block Graphics

Data	Carry and Mask	Left Shift
		Initial Value
		If high bit, set carry...
		...and strip high bit
		If 2nd high, set mask...
		...and strip 2nd high
		Multiply data by 2
		Add carry and mask

Data	Carry and Mask	Right Shift
		Initial Value
		If low bit set, carry...
		...and strip low bit
		If high bit, set mask...
		...and strip high bit
		Divide by two
		Add carry and mask
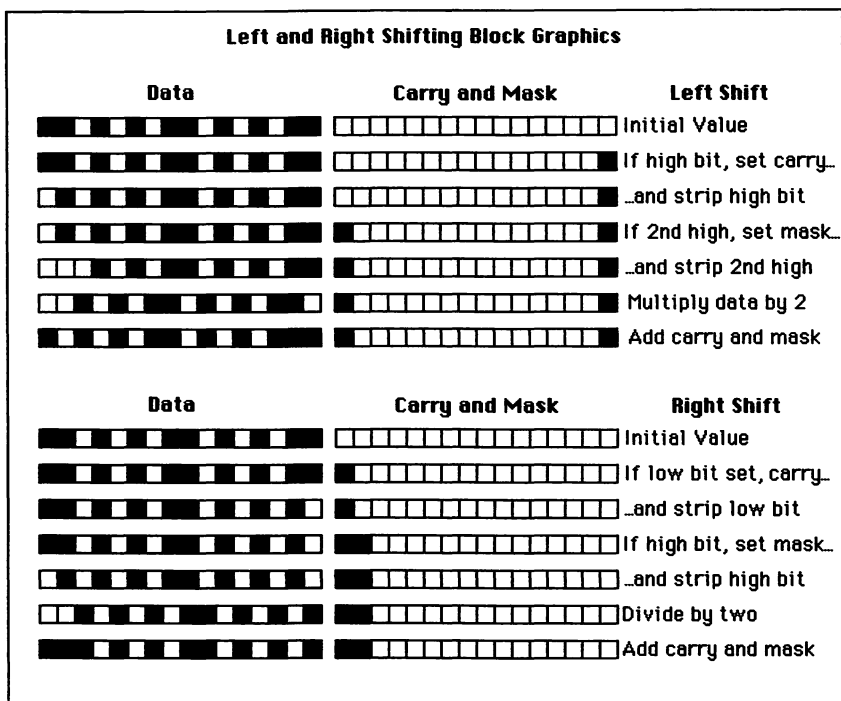
The Shape Editor

"The Shape Editor," Program 11-3, allows you to create and edit block graphics up to 64 pixels wide and 48 high for use in your own BASIC programs and applications. You can even pull graphics via the Scrapbook into the Shape Editor. Individual pixels can be toggled and the shapes can be shifted, inverted, and saved to disk.

Figure 11-4 is the input screen which has three input areas—the pixel toggle region, shape sizer, and buttons. Clicking within the toggle boxes inverts them. Clicking within the shape sizer box makes the shape the size selected. The *UP, DOWN, LEFT,* and *RIGHT* buttons shift the image in the corresponding directions. After selecting one of these buttons, click the source point in the toggle box, followed by the des-

tination point, and the image shifts the specified distance. Clicking outside the toggle box aborts the pixel-shift operations. *ERASE* and *INVERT* also require two endpoints that define the opposite corners of a rectangle to be erased or inverted. *LOAD* retrieves previously saved shapes and automatically sets the drawing size to that of the saved shape. *LOAD* and *SAVE* prompt you for a filename which becomes the default displayed if Return is pressed. *PRINT* outputs the screen to the printer and *QUIT* ends the program.

Figure 11-4. *"The Shape Editor" designs and saves shapes for use in BASIC programs. It also allows pictures to be read from the Scrapbook.*
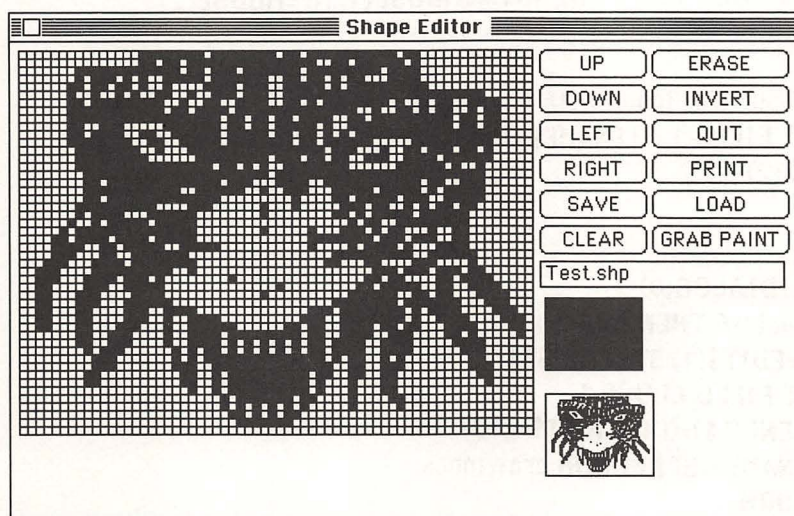
Program 11-3. The Shape Editor

```
GOSUB initvars
GOSUB drawshapebits
GOSUB drawpicbox
GOSUB drawbuttons
GOSUB drawsizer
GOSUB drawinbox

getm:
GOSUB getmouse
```

```
IF MX>5 AND MX<80*PICW+5 AND MY>5 AND MY<80*PICH+5 THEN
GOSUB setpixel:PUT (335,222),SHAPE,PSET:GOTO getm
IF MX>333 AND MX<393 AND MY>160 AND MY<205 THEN GOSUB
setpictsize:GOTO getm
GOSUB chkbut
IF BUTSEL> -1 THEN GOSUB invertbut:ON BUTSEL+1 GOSUB mov
eup,movedown,moveleft,moveright,picsave,picclear,picerase,pic
invert,quit,picprint,loadshape,grabpaint:GOSUB invertbut
GOTO getm

getmouse:
WHILE MOUSE(0)<1:WEND:MX=MOUSE(1):MY=MOUSE(2)
RETURN

getinput: 'for inbox with filename$
EDIT FIELD 1,FILENAME$,(INBOX(1)+2,INBOX(0)+1)-(INBOX(3)-2,I
NBOX(2)-2)

idle:
dact=DIALOG(0)
IF dact<>6 THEN idle
ST$=EDIT$(1):ST=VAL(ST$)
EDIT FIELD CLOSE 1
IF LEN(ST$)=0 THEN ST$=ZT$
FILENAME$=ST$:GOSUB drawinbox
RETURN

invertbut:
CALL INVERTROUNDRECT(VARPTR(BTN(0,BUTSEL)),10,10)
RETURN

setlabel:
CALL MOVETO(1000,20):PRINT BTN$(B);:CALL GETPEN(VARPT
R(PL(0))):BTN(4,B)=((BTN(3,B)-BTN(1,B))-(PL(1)-1000))\2
RETURN

setpixel:
X=MX\5:Y=MY\5:PX=X:PY=Y
```

```
GOSUB setmask
GOSUB setelm
SHAPE(ELM)=SHAPE(ELM) XOR MASK
GOSUB setdot
RETURN

setmask:
VX=PX
WHILE VX>16:VX=VX-16:WEND
TX=16-VX
IF TX=15 THEN MASK = -32768! ELSE MASK =2^TX
RETURN

setelm:
ELM=(PY-1)*PICW+(PX-1)\16+2
RETURN

setdot:
IF (MASK AND SHAPE(ELM))=0 THEN LINE (X*5+1,Y*5+1)- STEP
(3,3),30,BF ELSE LINE (X*5+1,Y*5+1)- STEP (3,3),33,BF
RETURN

setpictsize:
X=(MX-333)\15+1:Y=(MY-160)\15+1
IF X<>PICW OR Y<>PICH THEN PICW=X:PICH=Y:GOSUB drawshapeb
its:GOSUB drawpicbox:GOSUB drawsizer
RETURN

chkbut:
BUTSEL =-1:B=0
WHILE (B<MAXBUT+1) AND (BUTSEL= -1)
IF MX>BTN(1,B) AND MX<BTN(3,B) AND MY>BTN(0,B) AND MY<BTN
(2,B) THEN BUTSEL=B
B=B+1:WEND
RETURN

gettwopts:
GOSUB getmouse:X1=MX:Y1=MY
```

```
IF MX<5 OR MX>80*PICW+4 OR MY<5 OR MY>80*PICH+4 THEN X1=
0:RETURN
GOSUB getmouse:X2=MX:Y2=MY
IF MX<5 OR MX>80*PICW+4 OR MY<5 OR MY>80*PICH+4 THEN X1=
0:RETURN
X1= X1\5:Y1=Y1\5:X2=X2\5:Y2=Y2\5
IF X1>X2 THEN SWAP X1,X2
IF Y1>Y2 THEN SWAP Y1,Y2
RETURN


moveup:
GOSUB gettwopts:IF X1=0 THEN RETURN
DY=Y2-Y1:IF DY=0 THEN RETURN
FOR I=2 TO PICH*PICW*16-DY*PICW+1:SHAPE(I)=SHAPE(DY*PIC
W+I):NEXT I
FOR I=PICH*PICW*16-DY*PICW+2 TO PICH*PICW*16+1:SHAPE(I)
=0:NEXT I
GOSUB drawshapebits:GOSUB drawpicbox
RETURN


movedown:
GOSUB gettwopts:IF X1=0 THEN RETURN
DY=Y2-Y1:IF DY=0 THEN RETURN
FOR I=PICH*PICW*16+1 TO DY*PICW+2 STEP -1:SHAPE(I)=SHAP
E(I-DY*PICW):NEXT I
FOR I=DY*PICW+1 TO 2 STEP -1:SHAPE(I)=0:NEXT I
GOSUB drawshapebits:GOSUB drawpicbox
RETURN


moveleft:
GOSUB gettwopts:IF X1=0 THEN RETURN
DX=X2-X1:FOR I=0 TO PICW*PICH*16+1:SHAPE(I)=0:NEXT I
LINE(336+16*PICW,220)-STEP(0,3+PICH*16),30:GET (335+DX,2
22)-(334+PICW*16+DX,221+PICH*16),SHAPE:LINE(336+16*PICW
,220)-STEP(0,3+PICH*16),33
GOSUB drawshapebits:GOSUB drawpicbox
RETURN
```

282

```
moveright:
GOSUB gettwopts:IF X1=0 THEN RETURN
DX=X2-X1:FOR I=0 TO PICW*PICH*16+1:SHAPE(I)=0:NEXT I
LINE(333,220)-STEP(-16*PICW,3+PICH*16),30,BF:GET (335-DX
,222)-(334-DX+PICW*16,221+PICH*16),SHAPE:LINE(333,220)-S
TEP(0,3+PICH*16),33
GOSUB drawshapebits:GOSUB drawpicbox
RETURN


picsave:
GOSUB getinput
OPEN "O",#1,FILENAME$,512
WRITE#1,PICW,PICH
FOR I=0 TO PICW*PICH*16+1:WRITE #1,SHAPE(I):NEXT I
CLOSE #1
RETURN


picclear:
FOR I=2 TO 193:SHAPE(I)=0:NEXT I
GOSUB drawshapebits:GOSUB drawpicbox
RETURN


picerase:
GOSUB gettwopts:IF X1=0 THEN RETURN
X1=X1+334:X2=X2+334:Y1=Y1+221:Y2=Y2+221
LINE (X1,Y1)-(X2,Y2),30,BF
GET (335,222)-(334+PICW*16,221+PICH*16),SHAPE:GOSUB dra
wshapebits
RETURN


picinvert:
GOSUB gettwopts:IF X1=0 THEN RETURN
TRECT(1)=X1+334:TRECT(3)=X2+335:TRECT(0)=Y1+221:TRECT(2)=
Y2+222
CALL INVERTRECT(VARPTR(TRECT(0)))
GET (335,222)-(334+PICW*16,221+PICH*16),SHAPE:GOSUB dra
wshapebits
```

RETURN

quit:
END
RETURN

picprint:
LCOPY
RETURN

loadshape:
GOSUB getinput
ON ERROR GOTO chkerr
OPEN "I",**#**1,FILENAME$:**INPUT #**1,PICW,PICH:**FOR** I=0 **TO** PICW*P
ICH*16+1:**INPUT #**1,SHAPE(I):**NEXT** I:**CLOSE #**1:**GOSUB** drawsha
pebits:**GOSUB** drawpicbox:**GOSUB** drawsizer:**RETURN**
chkerr:
IF ERR = 53 **THEN** FILENAME$="Not found":**GOSUB** drawinbox:**RES**
UME reterr
reterr:
RETURN

grabpaint:
CLS:CALL MOVETO(10,270):**PRINT**"Copy a picture from the scra
pbook then click in this box";:**LINE** (470,260) **-STEP** (20,20),,b
WHILE (mx<470) **AND** (my<260):**GOSUB** getmouse:**WEND**
CLS
OPEN "CLIP:PICTURE" **FOR INPUT AS #**1
pic$=**INPUT$(LOF(**1),1)
PICTURE,pic$
CLOSE#1
bx=0:by=0:mx=0:my=0
CALL MOVETO(3,260):**PRINT**"Click around this box to adjust the
frame and in it":**PRINT**"to accept the frame.";
WHILE (mx<333) **OR** (mx>334+picw*16) **OR** (my<222) **OR** (my>22
2+pich*16)
GET (bx,by)-(bx+picw*16-1,by+pich*16-1),shape
GOSUB drawpicbox

284

```
GOSUB getmouse
IF mx>334+picw*16 AND bx>0 THEN bx=bx-1
IF mx<333 AND bx<picw*16 THEN bx=bx+1
IF my>222+pich*16 AND by>0 THEN by=by-1
IF my<222 AND by<pich*16 THEN by=by+1
WEND
CLS
GOSUB drawshapebits
GOSUB drawpicbox
GOSUB drawbuttons
GOSUB drawsizer
GOSUB drawinbox
GOSUB invertbut
RETURN

drawshapebits:
LINE (5,5) - (325,245),30,BF:SHAPE(0)=16*PICW:SHAPE(1)=16*P
ICH
FOR I=0 TO PICW-1:FOR J=0 TO 16*PICH-1
PIC(2+J)=SHAPE(2+J*PICW+I):NEXT J
PIC(0)=16:PIC(1)=PICH*16
X=5+80*I:PUT (X,5) - (X+80,5+PICH*80),PIC,PSET:NEXT I
FOR I=1 TO 16*PICW+1:LINE (I*5,5) - STEP (0,80*PICH):NEXT I
FOR I=1 TO 16*PICH+1:LINE (5,I*5) - STEP (80*PICW,0):NEXT I
RETURN

drawpicbox:
LINE (333,220) - STEP (67,51),30,BF:LINE (333,220) - STEP (3
+16*PICW,3+16*PICH),,B
PUT (335,222),SHAPE,PSET
RETURN

drawbuttons:
FOR B=0 TO MAXBUT
CALL FRAMEROUNDRECT(VARPTR(BTN(0,B)),10,10):CALL MOV
ETO(BTN(1,B)+BTN(4,B),BTN(2,B)-5):PRINT BTN$(B);
NEXT B
```

285

RETURN

drawsizer:
LINE (333,160)-**STEP**(60,45),30,bf:**LINE**(333,160)-**STEP**(60,45
),,b
LINE (333,160)-**STEP**(15*picw,15*pich),,bf
FOR I=348 **TO** 393 **STEP** 15:**LINE** (I,160) - **STEP** (0,45):**NEXT** I
FOR I=175 **TO** 205 **STEP** 15:**LINE** (333,I)- **STEP** (60,0):**NEXT** I
RETURN

drawinbox:
CALL ERASERECT(VARPTR(INBOX(0))):**CALL FRAMERECT(VAR
PTR(**INBOX(0)))
CALL MOVETO(INBOX(1)+4,INBOX(2)-5):**PRINT** FILENAME$;
RETURN

initvars:
CLS:DEFINT A-Z:**CALL TEXTMODE(**1):MAXBUT=11
DIM SHAPE(193),BTN$(MAXBUT),BTN(4,MAXBUT),PIC(49),PL(1),IN
BOX(3),PAT(7),TRECT(3)
FILENAME$="Test.shp":PICW=4:PICH=3
FOR B=0 **TO** 5:BTN(0,B)=22*B+5:BTN(1,B)=330:BTN(2,B)=BTN(0,B)
+18:BTN(3,B)=BTN(1,B)+67:**READ** BTN$(B):**GOSUB** setlabel:**NEXT**
B
FOR B=6 **TO** MAXBUT:BTN(0,B)=22*B-127:BTN(1,B)=400:BTN(2,B)
=BTN(0,B)+18:BTN(3,B)=BTN(1,B)+86:**READ** BTN$(B):**GOSUB** setla
bel:**NEXT** B
FOR I=0 **TO** 3:**READ** INBOX(I):**NEXT** I
FOR I=0 **TO** 7:**READ** PAT(I):**NEXT** I
RETURN

DATA "UP","DOWN","LEFT","RIGHT","SAVE","CLEAR","ERASE","INVE
RT","QUIT","PRINT","LOAD","GRAB PAINT"
DATA 137,330,154,483
DATA 0,0,0,0,-1,-1,-1,-1

E L E V E N

The Shape Editor is similar in construction to the Cursor Editor except that input is not tested by areas. The screen initializing is divided into five components because different operations require different parts of the screen to be redrawn. This program uses a trick to quickly redraw large images within the toggle boxes. Sixteen-pixel-wide slices of the shape are stored in an array and redrawn magnified at the top left of the screen, after which a grid is drawn over the top.

The Shape Editor can use graphics from the Scrapbook once you've copied them to the Clipboard. The program offers prompts when you're to open the Scrapbook and copy the picture. It can be larger than the Shape Editor's maximum dimensions since the program clips the picture to fit into the frame you set with the shape sizer box. The clipping is controlled by clicking the mouse around the framed image. For instance, if you want to clip the picture lower, click below the frame. When you have the framing you want, click within the frame. The image is clipped into the shape editor buffer with **GET**. After clipping a frame from the Scrapbook, the main input screen is reconstructed.

The Code Maker

"The Code Maker" generates code from the data files created by the Pattern, Cursor, or Shape Editors and stores such code in files which can be merged with other BASIC programs. First, a menu screen of buttons appears, presenting four options—Pattern Code, Cursor Code, and Shape Code generation, or Quitting the program. Selecting a button with the mouse invokes a program module, each of which has individual input screens. Figure 11-5 is the input screen for the Pattern Code generator.

Selecting the *LOAD* button highlights the input filename input box (in this case it contains *Test.pat*), indicating that you must enter a name or hit Return to accept the default. Once the Pattern Editor data file is loaded, an enlargement of the bit representation and pattern is displayed to verify the selection. Selecting the *Generate Code* button highlights the output filename input box (containing *Testpat.code* in Figure 11-5) to contain the BASIC code. *Exit* returns you to the main menu.

Figure 11-5. *Pattern Code generator. The pattern is displayed along with a ten times magnification for verification before generating code. Patterns are loaded from disk by clicking the LOAD button and entering its filename. Another filename must be entered when generating code which can be merged with program files.*
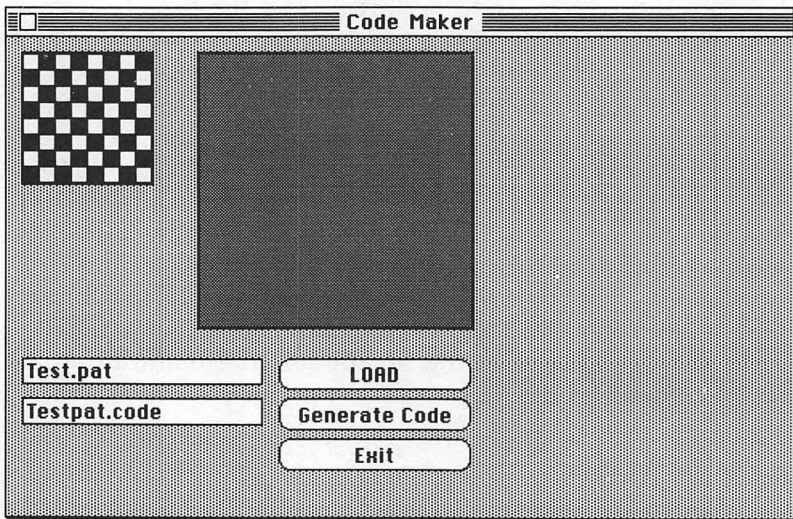
Figure 11-6 is the input screen of the Cursor Code generating module. The buttons are the same as those for Pattern, and the cursor and mask are displayed whenever Cursor Editor data is loaded.

ELEVEN

Figure 11-6. *Cursor Code generator. Both cursor and mask are displayed at ten times magnification for verification prior to code generation.*
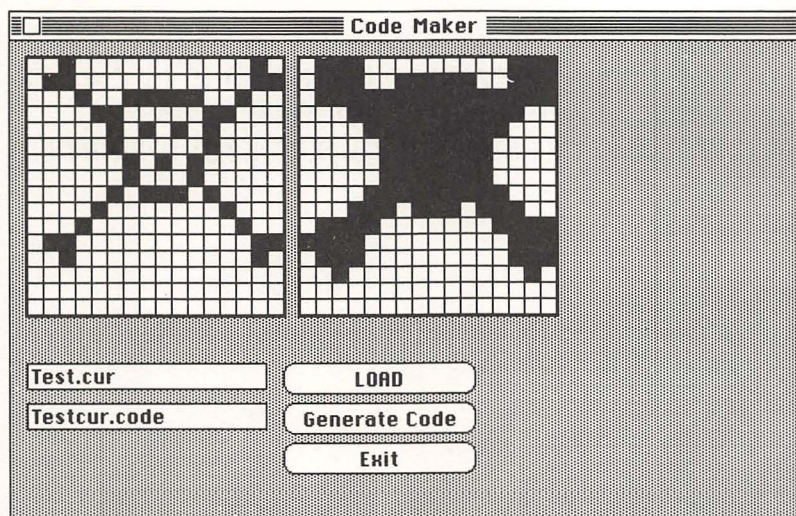
Program 11-4. *The Code Maker*
GOSUB initvars

```
screen1:
GOSUB initscreen1

getm:
WHILE MOUSE(0)<1:WEND
MX=MOUSE(1):MY=MOUSE(2)
GOSUB chkbutset1:IF BUTSEL> -1 THEN GOSUB invertbutton:ON
   BUTSEL+1 GOSUB patterns,cursors,shapes,quit:GOTO screen1
GOTO getm

drawrects:
RECT(0,B)=RECT(0,B)-1:RECT(1,B)=RECT(1,B)-1:RECT(2,B)=RECT(2
,B)+1:RECT(3,B)=RECT(3,B)+1
CALL PENSIZE(2,2):CALL ERASERECT(VARPTR(RECT(0,B))):CA
LL FRAMERECT(VARPTR(RECT(0,B))):CALL PENSIZE(1,1)
```

289

```
RECT(0,B)=RECT(0,B)+1:RECT(1,B)=RECT(1,B)+1:RECT(2,B)=RECT(2
,B)-1:RECT(3,B)=RECT(3,B)-1
RETURN

drawbutton:
CALL ERASEROUNDRECT(VARPTR(BUTTN(0,B)),15,15):CALL FR
AMEROUNDRECT(VARPTR(BUTTN(0,B)),15,15)
CALL MOVETO(BUTTN(1,B)+BUTTN(4,B),BUTTN(2,B)-5):PRINT BU
TTN$(B);
RETURN

drawinbox: 'b
CALL ERASERECT(VARPTR(INBOX(0,B))):CALL FRAMERECT(VA
RPTR(INBOX(0,B)))
RETURN

printinbox: 'b
CALL MOVETO(INBOX(1,B)+3,INBOX(2,B)-4):PRINT INBOX$(B);
RETURN

getinput: 'for inbox b with default in zt!,zt$
EDIT FIELD 1,zt$,(INBOX(1,B)+2,INBOX(0,B)+1)-(INBOX(3,B)-2,IN
BOX(2,B)-2)

idle:
dact=DIALOG(0)
IF dact<>6 THEN idle
ST$=EDIT$(1):ST=VAL(ST$)
EDIT FIELD CLOSE 1
GOSUB drawinbox
IF LEN(ST$)=0 THEN ST$=ZT$:ST=ZT
INBOX$(B)=ST$:GOSUB printinbox
RETURN

invertbutton:
CALL INVERTROUNDRECT(VARPTR(BUTTN(0,BUTSEL)),15,15)
RETURN
```

ELEVEN

```
setlabel:
CALL MOVETO(1000,20):PRINT BUTTN$(B);:CALL GETPEN(VAR
PTR(PL(0))):BUTTN(4,B)=((BUTTN(3,B)-BUTTN(1,B))-(PL(1)-1000
))\2
RETURN


chkbuttons:
BUTSEL= -1:B=B1:WHILE (BUTSEL= -1) AND (B<=B2)
IF (MX>BUTTN(1,B)) AND (MX<BUTTN(3,B)) AND (MY>BUTTN(0,B))
AND (MY<BUTTN(2,B)) THEN BUTSEL=B
B=B+1:WEND
RETURN


chkbutset1:
B1=0:B2=3:GOSUB chkbuttons
RETURN


chkbutset2:
B1=4:B2=6:GOSUB chkbuttons
RETURN


quit:
END


patterns:
GOSUB initscreen2


getm1:
WHILE MOUSE(0)<1:WEND
MX=MOUSE(1):MY=MOUSE(2)
GOSUB chkbutset2:IF BUTSEL=6 THEN GOSUB invertbutton:RETU
RN
IF BUTSEL> -1 THEN GOSUB invertbutton:ON BUTSEL-3 GOSUB l
oadpat,genpatcode:GOSUB invertbutton
GOTO getm1
RETURN


loadpat:
```

```
B=0:ZT$=INBOX$(B):ZT!=0:GOSUB getinput
ON ERROR GOTO chkerr1
OPEN "I",#1,INBOX$(B):FOR I=0 TO 3:INPUT #1,PATTERN(I):NEXT
  I:CLOSE #1:GOSUB drawpatbitboxes:GOSUB drawpatbox:RETUR
N

chkerr1:
IF ERR=53 THEN INBOX$(B)="Not Found":ST$=INBOX$(B):GOSUB d
rawinbox:GOSUB printinbox:RESUME ldret1

ldret1:
RETURN

genpatcode: 'generate button
B=1:ZT$=INBOX$(B):ZT!=0:GOSUB getinput
OPEN "O",#1,INBOX$(B)
ST$="DATA ":FOR I=0 TO 2:ST$=ST$+STR$(PATTERN(I))+",":NEXT
  I:ST$=ST$+STR$(PATTERN(3))+" "+INBOX$(0):PRINT#1,ST$
CLOSE#1
RETURN

cursors:
GOSUB initscreen3

getm2:
WHILE MOUSE(0)<1:WEND
MX=MOUSE(1):MY=MOUSE(2)
GOSUB chkbutset2:IF BUTSEL=6 THEN GOSUB invertbutton:RETU
RN
IF BUTSEL>-1 THEN GOSUB invertbutton:ON BUTSEL-3 GOSUB l
oadcursor,gencursorcode:GOSUB invertbutton
GOTO getm2
RETURN

loadcursor: 'load button
B=0:ZT$=INBOX$(B):ZT!=0:GOSUB getinput
ON ERROR GOTO chkerr2
OPEN "I",#1,INBOX$(B):FOR I=0 TO 33:INPUT #1,CURSOR(I):NEXT
```

```
l:CLOSE #1:GOSUB drawcursor:GOSUB drawmask:RETURN

chkerr2:
IF ERR=53 THEN INBOX$(B)="Not Found":ST$=INBOX$(B):GOSUB d
rawinbox:GOSUB printinbox:RESUME ldret2

ldret2:
RETURN

gencursorcode: 'generate button
B=1:ZT$=INBOX$(B):ZT!=0:GOSUB getinput
OPEN "O",#1,INBOX$(B)
ST$="DATA ":FOR I=0 TO 14:ST$=ST$+STR$(CURSOR(I))+",":NEXT
 I:ST$=ST$+STR$(CURSOR(15))+": '"+INBOX$(0):PRINT#1,ST$
ST$="DATA ":FOR I=16 TO 30:ST$=ST$+STR$(CURSOR(I))+",":NEX
T I:ST$=ST$+STR$(CURSOR(31))+": 'mask":PRINT#1,ST$
ST$="DATA ":ST$=ST$+STR$(CURSOR(32))+",":ST$=ST$+STR$(CU
RSOR(33))+": 'hot spot":PRINT#1,ST$
CLOSE#1
RETURN

shapes:
GOSUB initscreen4

getm3:
WHILE MOUSE(0)<1:WEND
MX=MOUSE(1):MY=MOUSE(2)
GOSUB chkbutset2:IF BUTSEL=6 THEN GOSUB invertbutton:RETU
RN
IF BUTSEL>-1 THEN GOSUB invertbutton:ON BUTSEL-3 GOSUB l
oadshape,genshapecode:GOSUB invertbutton
GOTO getm3
RETURN
loadshape: 'Load
B=0:ZT$=INBOX$(B):ZT!=0:GOSUB getinput
ON ERROR GOTO chkerr3
OPEN "I",#1,INBOX$(B):INPUT#1,PICW,PICH:FOR I=0 TO PICW*PI
CH*16+1:INPUT #1,SHAPE(I):NEXT I:CLOSE #1:GOSUB drawshap
ebits:GOSUB drawshape:RETURN
```

293

```
chkerr3:
IF ERR=53 THEN INBOX$(B)="Not Found":ST$=INBOX$(B):GOSUB d
rawinbox:GOSUB printinbox:RESUME ldret3


ldret3:
RETURN


genshapecode: 'generate
B=1:ZT$=INBOX$(B):ZT!=0:GOSUB getinput
OPEN "O",#1,INBOX$(B)
ST$="DATA "+STR$(SHAPE(0))+","+STR$(SHAPE(1))+": 'size of "+
INBOX$(0):PRINT #1,ST$
FOR I1=1 TO PICH*16
ST$="DATA "
FOR I2=1 TO PICW
ST$=ST$+STR$(SHAPE((I1-1)*PICW+I2+1))+","
NEXT I2:PRINT #1,LEFT$(ST$,LEN(ST$)-1):NEXT I1
CLOSE#1
RETURN


initscreen1:
GOSUB setbkgnd
FOR B=0 TO 3:GOSUB setlabel:GOSUB drawbutton:NEXT B
RETURN


setbkgnd:
CALL BACKPAT(VARPTR(PAT(4))):CLS:CALL BACKPAT(VARP
TR(PAT(0)))
RETURN


initscreen2:
GOSUB setbkgnd
GOSUB drawpatbitboxes:GOSUB drawpatbox
INBOX$(0)="Test.pat":INBOX$(1)="Testpat.code"
FOR B=0 TO 1:GOSUB drawinbox:GOSUB printinbox:NEXT B
```

```
FOR B=4 TO 6:GOSUB setlabel:GOSUB drawbutton:NEXT B
RETURN


drawpatbitboxes:
B=0:GOSUB drawrects:PIC(0)=8:PIC(1)=8
X=RECT(1,0):Y=RECT(0,0)
FOR I=0 TO 3:PIC(I*2+2)=PATTERN(I):PIC(I*2+3)=PATTERN(I) AN
D 127:TV=PATTERN(I) AND 128:PIC(I*2+3)=(PIC(I*2+3) * 256) X
OR (TV * -256):NEXT I
PUT (X,Y) - (X+80,Y+80),PIC,PSET:FOR I=X TO X+80 STEP 10:LIN
E (I,Y) - STEP (0,80):NEXT I:FOR Y=10 TO 90 STEP 10:LINE (X,Y
) - STEP (80,0):NEXT Y
RETURN


drawpatbox:
B=1:GOSUB drawrects
RECT(0,B)=RECT(0,B)+1:RECT(1,B)=RECT(1,B)+1:RECT(2,B)=RECT(2
,B)-1:RECT(3,B)=RECT(3,B)-1
CALL FILLRECT(VARPTR(RECT(0,B)),VARPTR(PATTERN(0)))
RECT(0,B)=RECT(0,B)-1:RECT(1,B)=RECT(1,B)-1:RECT(2,B)=RECT(2
,B)+1:RECT(3,B)=RECT(3,B)+1
RETURN


initscreen3:
GOSUB setbkgnd
GOSUB drawcursor:GOSUB drawmask
INBOX$(0)="Test.cur":INBOX$(1)="Testcur.code"
FOR B=0 TO 1:GOSUB drawinbox:GOSUB printinbox:NEXT B
FOR B=4 TO 6:GOSUB setlabel:GOSUB drawbutton:NEXT B
RETURN


drawmaskcursor:
B=R:GOSUB drawrects:PIC(0)=16:PIC(1)=16
X=RECT(1,R):Y=RECT(0,R)
FOR I=OS TO OS+15:PIC(2+I-OS)=CURSOR(I):NEXT I
PUT (X,Y) - (X+160,Y+160),PIC,PSET:FOR I=X TO X+160 STEP 10
:LINE (I,Y) - STEP (0,160):NEXT I:FOR Y=10 TO 170 STEP 10:LI
NE (X,Y) - STEP (160,0):NEXT Y
```

RETURN

drawcursor:
OS=0:R=2:**GOSUB** drawmaskcursor
RETURN

drawmask:
OS=16:R=3:**GOSUB** drawmaskcursor
RETURN

initscreen4:
GOSUB setbkgnd
GOSUB drawshapebits:**GOSUB** drawshape
INBOX$(0)="Test.shp":INBOX$(1)="Testshp.code"
FOR B=0 **TO** 1:**GOSUB** drawinbox:**GOSUB** printinbox:**NEXT** B
FOR B=4 **TO** 6:**GOSUB** setlabel:**GOSUB** drawbutton:**NEXT** B
RETURN

drawshapebits:
B=6:**CALL FILLRECT(VARPTR**(RECT(0,B)),**VARPTR**(PAT(4))):B=4
:RECT(2,B)=5+PICH*64:RECT(3,B)=11+PICW*64:**GOSUB** drawrects
R=B:X=RECT(1,R):Y=RECT(0,R)
SHAPE(0)=16*PICW:SHAPE(1)=16*PICH:**PUT** (X,Y) – (X+64*PICW,Y
+64*PICH),SHAPE,**PSET**:**FOR** I=X **TO** X+64*PICW **STEP** 4:**LINE** (I,Y
) – **STEP** (0,64*PICH):**NEXT** I:**FOR** Y=4 **TO** 4+64*PICH **STEP** 4:**LIN**
E (X,Y) – **STEP** (64*PICW,**O**):**NEXT** Y
RETURN

drawshape:
B=5:RECT(2,B)=10+PICH*16:RECT(3,B)=285+PICW*16:**GOSUB** dra
wrects
R=B:X=RECT(1,R)+2:Y=RECT(0,R)+2
PUT (X,Y) – (X+PICW*16,Y+PICH*16),SHAPE,**PSET**
RETURN

initvars:
CLS:**DEFINT** A-Z:**CALL TEXTMODE**(1):**CALL TEXTFONT**(0):**CALL**
TEXTSIZE(12):MAXBUTTON=6:MAXINBOX=2:MAXRECT=6

```
DIM PATTERN(3),CURSOR(33),SHAPE(194),PAT(11),RECT(3,MAXRE
CT),TRECT(3),BUTTN(4,MAXBUTTON),BUTTN$(MAXBUTTON),INBOX(
3,MAXINBOX),INBOX$(MAXINBOX),PL(1),PIC(17)
PICW=4:PICH=3:PIC(1)=1
FOR I=0 TO 11:READ PAT(I):NEXT I
FOR I=0 TO 2:INBOX(0,I)=200+I*25:INBOX(1,I)=10:INBOX(2,I)=INB
OX(0,I)+17:INBOX(3,I)=INBOX(1,I)+150:NEXT I
FOR I=0 TO 3:BUTTN(0,I)=70+25*I:BUTTN(1,I)=160:BUTTN(2,I)=BU
TTN(0,I)+20:BUTTN(3,I)=BUTTN(1,I)+180:READ BUTTN$(I):NEXT I
FOR I=4 TO 6:BUTTN(0,I)=25*I+100:BUTTN(1,I)=170:BUTTN(2,I)=B
UTTN(0,I)+20:BUTTN(3,I)=BUTTN(1,I)+120:READ BUTTN$(I):NEXT
I
FOR I=0 TO MAXRECT:FOR J=0 TO 3:READ RECT(J,I):NEXT J,I
RETURN


DATA 0,0,0,0,4420,4420,4420,4420,-1,-1,-1,-1:'patterns
DATA "Pattern Code Generator","Cursor Code Generator","Shape C
ode Generator","Quit"
DATA "LOAD","Generate Code","Exit"
DATA 10,10,91,91:'pattern bits
DATA 10,120,181,291:'pattern
DATA 10,10,171,171:'cursor
DATA 10,180,171,341:'mask
DATA 4,10,197,203:'shape bits
DATA 5,280,56,347:'shape
DATA 3,5,198,350:'erase rect
```

Code Maker is an example of combining three independent programs so that they can share the same software subroutine. The programming points of interest involve the organization of similar subroutines into the same area of the program and sequencing them in the order in which the modules call them. Most of the routines are like those in previous programs, except for code generation, and since these routines are almost identical, only one of them is discussed.

The subroutine *genshapecode* generates **DATA** statements to represent the Shape Editor's graphics. This data is read into an array by a host program and is displayed with **PUT**. It gets

a code filename by accessing *getinput*. INBOX$(B) is the storage for the three inputs: input filename and code filename. The output file is opened and ST$ is used to compose and record the first **DATA** statement. The **FOR-NEXT** loop generates the remaining code. Data elements are appended to **ST$** before the code is written with **PRINT** #1.

Software tool subroutines perform various tasks that are simple enough to be easily customized and optimized for any particular program. Thus, they greatly reduce the effort and time required to complete an application. Programs are also software tools when their output is usable by other programs or in program development. Note that programs like Pattern, Cursor, and Shape Editor and Code Maker are in effect software tools. They were used to create the game which follows.

Games on the Macintosh

Games are programmed in a manner similar to the editors, but they have a loop which processes background activities while waiting for user input. Some background activities are time-consuming and should be executed at a frequency less than the loop in order to maintain a smooth realtime effect. The slowness of BASIC makes noticeable pauses when executing triggered events such as missile detonation. The shapes in this program were created with the Shape Editor, and their code (the **DATA** statements at the end of the program listing) was created with Code Maker.

You get three turrets with which to fire missiles and eliminate an endless swarm of aliens marching slowly toward you. They're dropping bombs as they move. They march faster as the swarm size diminishes, and if they reach the ground, the game ends. You score points for each alien you destroy with your missiles, which are launched by pressing the mouse button. Your turret is positioned by sliding the mouse left or right. Eliminating the entire swarm causes a new swarm to appear at a position lower than the last. Occasionally, a spaceship flies overhead, providing another target and bonus points if you hit it. If a falling bomb hits your turret, it's eliminated from play and another is released. After three turrets are destroyed, the game ends. The score is displayed at the top left as you play.

Figure 11-7 shows the game in progress, with aliens in the center of the *Output* window, the spaceship at the top, and

turret at the bottom. The number of turrets left is displayed at the lower left. The mouse pointer is hidden during play unless you drag it outside the *Output* window, where it's revealed until you move it back. A missile can't be launched while the mouse is outside so that menu items or other windows can be selected while the game is in progress. This feature is not programmed, but is part of the environment associated with BASIC.

Figure 11-7. *The goal of "Alien" is to eliminate as many creatures as possible before they eliminate you. You have three turrets as you go for a high score. A spaceship flies overhead every 15 seconds—bonus points are awarded if you destroy it.*
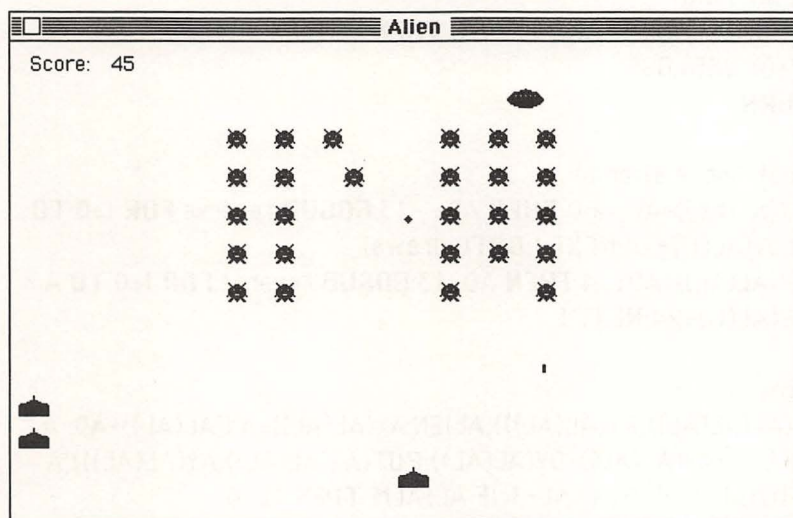
Program 11-5. Alien

```
GOSUB initvars
GOSUB initaliens:PUT(BX,265),BSE:PUT(5,240),BSE:PUT(5,220),
BSE
GOSUB showscore

loop:
GOSUB drawbase:GOSUB moveal:GOSUB moveshell:GOSUB alfir
e:GOSUB movebigal
```

```
IF TK=TKM THEN GOSUB initaliens
GOTO loop

drawbase:
M=MOUSE(0):IF M<0 THEN GOSUB fireone:
IF (MOUSE(1)<0) OR (MOUSE(2)<0) THEN CALL INITCURSOR EL
SE CALL HIDECURSOR
IF MOUSE(1)<BX-16 THEN PUT(BX,265),BSE:BX=BX-8:PUT(BX,26
5),BSE:RETURN
IF MOUSE(1)>BX+16 AND BX<475 THEN PUT(BX,265),BSE:BX=BX+
8:PUT(BX,265),BSE:RETURN
IF MOUSE(1)<BX-4 THEN PUT(BX,265),BSE:BX=BX-4:PUT(BX,265
),BSE:RETURN
IF MOUSE(1)>BX+4 AND BX<475 THEN PUT(BX,265),BSE:BX=BX+4
:PUT(BX,265),BSE
RETURN

moveal: 'move alien al
IF AX(AL(AL))+AD>460 THEN AD= -13:GOSUB reverse:FOR I=0 TO
ALM:DY(AL(I))=24:NEXT I:GOTO drawal
IF AX(AL(AL))+AD<24 THEN AD=13:GOSUB reverse:FOR I=0 TO A
LM:DY(AL(I))=24:NEXT I

drawal:
PUT(AX(AL(AL)),AY(AL(AL))),ALIEN:AX(AL(AL))=AX(AL(AL))+AD: A
Y(AL(AL))=AY(AL(AL))+DY(AL(AL)):PUT(AX(AL(AL)),AY(AL(AL))),A
LIEN:DY(AL(AL))=0:AL=AL+1:IF AL>ALM THEN AL=0
IF AY(AL(AL))>260 THEN BEEP:CALL MOVETO(10,36):PRINT"Yo
u Lose!":CALL INITCURSOR:END
IF AX(AL(AL))<MX AND AX(AL(AL))>MX+12 AND AY(AL(AL))<MY AN
D AY(AL(AL))>MY+11 THEN K=AL:GOSUB killal
RETURN

reverse:
IF ALM<>0 THEN FOR I=0 TO (ALM-1)\2:T=AL(I):AL(I)=AL(ALM-I)
:AL(ALM-I)=T:NEXT I:AL=0
RETURN
```

```
moveshell:
IF MF=0 THEN RETURN
PUT (MX,MY),MSL:MY=MY-24:IF MY<24 THEN MF=0:RETURN
IF POINT(MX,MY+2)=33 THEN HX=MX:HY=MY+2:GOSUB killchk:MF=
0:RETURN
IF POINT(MX+2,MY)=33 THEN HX=MX+2:HY=MY:GOSUB killchk:MF=
0:RETURN
IF POINT(MX+4,MY+2)=33 THEN HX=MX+4:HY=MY+2:GOSUB killchk
:MF=0:RETURN
PUT (MX,MY),MSL
RETURN

fireone:
IF MF=1 THEN RETURN
MF=1:MX=BX+7:MY=250:PUT (MX,MY),MSL
RETURN

killchk:
IF HY<40 THEN PUT(BGX,BGY),BIGAL:GOSUB killbigal:RETURN
IF (B1X<=HX) AND (B1X+1>=HX) AND (B1Y<=HY) AND (B1Y+4>=HY)
 THEN PUT(B1X,B1Y),BB:PUT (HX,HY),EX1:BF1=0:PUT (HX,HY),EX1:
RETURN
K=ALM:WHILE NOT ((AX(AL(K))<=HX) AND (AX(AL(K))+12>=HX) AN
D (AY(AL(K))<=HY) AND (AY(AL(K))+12>=HY)): K=K-1:WEND:IF K=-
1 THEN RETURN

killal:
PUT (AX(AL(K)),AY(AL(K))),ALIEN:PUT (AX(AL(K)),AY(AL(K))),EX1:
PUT (AX(AL(K)),AY(AL(K))),EX2:SC=SC+5:GOSUB showscore:PUT (
AX(AL(K)),AY(AL(K))),EX1:PUT (AX(AL(K)),AY(AL(K))),EX2:FOR J=K
 TO ALM-1:AL(J)=AL(J+1):NEXT J:ALM=ALM-1:TK=TK+1
IF AL>K THEN AL=AL-1
IF AL>ALM THEN AL=0
RETURN

killbigal:
PUT(BGX,BGY),EX3:BGF=0:SC=SC+100:GOSUB showscore
IF DBGX=10 THEN DBGX= -10:STBGX=460 ELSE DBGX=10:STBGX=4
0
```

```
PUT(BGX,BGY),EX3
RETURN

alfire:
IF BF1=1 THEN GOSUB movealbb:RETURN
IF AY(AL(0))>256 THEN RETURN
B1X=AX(AL(0))+6:B1Y=AY(AL(0))+5:BF1=1:PUT(B1X,B1Y),BB
RETURN

movealbb:
PUT (B1X,B1Y),BB:B1Y=B1Y+24:IF (POINT(B1X,B1Y+2)=33 AND B
1Y>270) THEN NBSE=NBSE-1:IF NBSE=0 THEN BEEP:CALL MOVE
TO(10,36):PRINT"You Lose!":CALL INITCURSOR:END ELSE BF1=0
:PUT(5,200+20*NBSE),BSE:PUT(BX,265),BSE:BX=10:PUT(BX,265),
BSE:RETURN
IF B1Y>280 THEN BF1=0:RETURN
PUT (B1X,B1Y),BB
RETURN

movebigal:
IF BGF=1 THEN redrawbigal
IF TIMER>BGT#+15 THEN BGT#=TIMER:BGX=STBGX:BGY=29:BGF=
1: PUT(BGX,BGY),BIGAL
RETURN

redrawbigal:
PUT(BGX,BGY),BIGAL:BGX=BGX+DBGX:IF BGX<30 OR BGX>460 THEN
 BGF=0:RETURN
IF (BGX<MX) AND (BGX+14>MX) AND MY=33 THEN PUT(MX,MY),MSL
:MF=0:GOSUB killbigal:RETURN
PUT(BGX,BGY),BIGAL:IF MF=1 AND MY=<40 AND POINT(MX+2,MY)
=30 THEN PUT(BGX,BGY),BIGAL:GOSUB killbigal
RETURN

showscore:
LINE (10,5)-(90,24),30,BF:CALL MOVETO(12,18):PRINT"Score:
";SC
```

RETURN

initaliens:
```
FOR I=0 TO 4:FOR J=0 TO 6:AY(I+J*5)=MAXALY-I*24:AX(I+J*5)=2
24-30*J: AL(I+J*5)=I+J*5:AD=13:DY(I+J*5)=0:NEXT J,I
FOR I=0 TO 34:PUT (AX(I),AY(I)),ALIEN:NEXT I
TKM=TK+35:ALM=34:AL=0:BGT*=TIMER:IF MAXALY<215 THEN MA
XALY=MAXALY+24:IF MAXALY=222 THEN MAXALY=150
```
RETURN

initvars:
```
CLS:DEFINT A-Z:CALL PENMODE(1):CALL HIDECURSOR
DIM BIGAL(25),BSE(27),ALIEN(13),MSL(6),BB(6),EX1(12),EX2(6),E
X3(12),AX(34),AY(34),AL(34),DY(34)
FOR I=0 TO 25:READ BIGAL(I):NEXT I
FOR I=0 TO 27:READ BSE(I):NEXT I
FOR I=0 TO 13:READ ALIEN(I):NEXT I
FOR I=0 TO 6:READ MSL(I):NEXT I
FOR I=0 TO 6:READ BB(I):NEXT I
FOR I=0 TO 12:READ EX1(I):NEXT I
FOR I=0 TO 6:READ EX2(I):NEXT I
FOR I=0 TO 12:READ EX3(I):NEXT I
BX=10:MF=0:TK=0:SC=0:BF1=0:BGF=0:STBGX=460:DBGX= -15:MAXA
LY=150:NBSE=3
```
RETURN

```
DATA 23, 12: 'size of space ship
DATA 16, 0,254 , 0,4095 ,-8192,8191 ,-4096,15213 ,-18432,3
2767 ,-1024,-1,-512,-1,-512,32767 ,-1024,16383 ,-2048,4095
 -8192,511 ,0:'space ship
DATA 19, 13: 'size of turret
DATA 64, 0,64 , 0,64 , 0,496 , 0,2044 , 0,32767 ,-16384,-1,-81
92,-1,-8192,-1,-8192,-1,-8192,-1,-8192,-1,-8192,-1,-8192:'t
urret
DATA 13, 12: 'size of alien
DATA 16400,10016,8128,16352,30576,32752,32752,30960,-1
6408,8128,14176 ,25136:'alien
```

ELEVEN

DATA 5,5,8192,28672,-2048,28762,8192:'msl
DATA 2,5,-16384,-16384,-16384,-16384,-16384:'bb
DATA 10,11,-31680,17792,10496,4480,-30144,17408,13312,-
13760,4480,10368,-15296:'ex1
DATA 7,5,10240,-28160,17408,-28160,10240:'ex2
DATA 10,11,-31680,17792,10496,4480,-30144,17408,13312,-
13760,4480,10368,-15296:'ex3

The first section of the game program initializes variables, creates the playing screen, calls various subroutines to update background events, and monitors and reacts to user input. This is standard procedure for a game. The loop calls subroutines which identify mouse button presses and mouse repositioning; move the turret, aliens, spaceship, missiles, and bombs; and detect collisions between bombs and turrets and between missiles and aliens.

The *drawbase* subroutine calls **MOUSE(0)** to see if the button is pressed. When pressed, the *fireone* subroutine launches a missile if one is not in flight. In that routine, MF, the missile flag, is checked—it's 1 when a missile is active and therefore another can't be fired. If none exists, the next line prepares MX and MY, the missile's horizontal and vertical positions, and draws the missile. BX is the left edge of the turret, so BX + 7 is used to launch the missile from the turret's midpoint.

MOUSE(1) and **MOUSE(2)** are checked back in *drawbase* to control the display status of the mouse pointer with **INITCURSOR** and **HIDECURSOR**, based on its position. Other lines in that routine determine the mouse position relative to the turret and move it toward the mouse. Notice that the last two lines of this routine provide for finer positioning adjustments. The deviation between the mouse and the turret must be greater than the motion increment, or the turret will oscillate between two positions when the mouse is motionless.

The *moveal* subroutine updates the position of alien AL. Each alien has a position marked by AX(AL(AL)) and AY(AL(AL)), and a vertical direction in DY(AL(AL)). AL(AL) is an alien in a list of aliens left in the swarm which numbers ALM. Since this list is constantly decreasing, but the alien to be updated is increasing, indirection is required to prevent skipping live aliens and including dead ones. AD represents

304

the horizontal motion displacement of the swarm. The first two lines determine if the aliens have to reverse direction and call the subroutine *reverse* to perform the variable changes required. The **PUT** statement redraws alien AL before a check is done to see if it's reached the ground. Other checks determine if the alien has collided with a missile, then call *killal* when this occurs.

Moveshell moves any existing missile, aborting the update if no missile is in flight. Otherwise, the missile is redrawn and repositioned before the rest of the routine looks for a collision. POINT returns 33 if contact is made at any of the three leading corners of the missile because checking every point in the missile takes too long. To prevent undetectable contacts, each alien, spaceship, and bomb is drawn in predetermined locations. There are enough of these to simulate smooth-moving graphics.

When a missile hits a target, *killchk* is called. The first line looks for a collision with the spaceship and calls *killbigal* to destroy the ship, reverse its next arrival direction, reset its flag (BGF), and increase the score. The second line of *killchk* detects the collision between the missile and an alien bomb. If the target was not the spaceship or a bomb, it must be an alien, so K is manipulated until it yields an alien to destroy. Failing to determine a target, K contains -1 and the target goes unidentified. *Killal* erases an alien, updates the score, and adjusts the list of living aliens. AL is decreased in case the next alien to move was killed. Then, AL is made equal to zero if the next alien to move was alien ALM and it was destroyed.

Alfire drops an alien bomb unless it exists, in which case *movealbb* moves the bomb. In that routine, lines recalculate its position and check to see if it has hit a turret. In such a case, the turret count NBSE is decremented and the game ends if none is left.

Movebigal updates the spaceship position with a call to *redrawbigal* if BGF indicates it exists. The spaceship is erased and its new position is recalculated before the program determines if it has flown off the screen. Other checks are made to see if it's collided with a missile. The next ship arrival time is BGT# which is compared with **TIMER** so that it can be drawn and its coordinates stored in BGX and BGY when it arrives.

Repeatedly calling these routines runs the game. More elaborate games have longer cycles and subcycles which call

ELEVEN

some of the updating routines of the main cycle. One of the main challenges of a program which involves missile collision detection is to insure that every possible collision is accounted for. POINT may yield 33 at a test point, but the program may not be able to determine what has collided with the missile. Use flags to determine if a background event is in progress or can be initiated.

Randomness can be added to give the aliens various flight paths stored in arrays. Each alien is given a random path to follow and a position along the path; these are stored in an array of path numbers and path locations. This technique speeds up the animation, enhances the game, but requires more memory to execute.

Index

307

To order your copy of *Advanced Macintosh BASIC Programming Disk*, call our toll-free US order line: 1-800-334-0868 (in NC call 919-275-9809) or send your prepaid order to:

Advanced Macintosh BASIC Programming Disk
COMPUTE! Publications
P.O. Box 5058
Greensboro, NC 27403

All orders must be prepaid (check, charge, or money order). NC residents add 4.5% sales tax.

Send _____ copies of *Advanced Macintosh BASIC Programming Disk* at $15.95 per copy.

Subtotal $_____

Shipping & Handling: $2.00/disk $_____

Sales tax (if applicable) $_____

Total payment enclosed $_____

All payments must be in U.S. funds.

□ Payment enclosed
□ Charge □ Visa □ MasterCard □ American Express

Acct. No. _____ Exp. Date _____
                                                            (Required)

Signature _____

Name _____

Address _____

City _____ State _____ Zip _____

Please allow 4-5 weeks for delivery.

45A0303

Advanced BASIC

The Macintosh—one of the most versatile personal computers today—becomes even more powerful when you can create your own programs. Microsoft BASIC 2.0 puts this power at your disposal, *if* you know how to use it.

Advanced Macintosh BASIC Programming is your reference guide and tutorial to this dynamic programming language. The advanced commands, statements, and techniques of Microsoft BASIC 2.0 are explained, then illustrated with practical example programs. You'll learn how to use various Macintosh peripherals— such as the disk drive and printer—and how to use the Mac's unique graphic capabilities. Included are concepts like creating and maintaining a mailing list, drawing business graphics, block graphic animation, creating functional dialog boxes with control buttons and generating pull-down menus.

- Learn how to create special user input routines for your own applications.
- Place shapes—from circles to polygons—anywhere on the screen.
- See how to access both sequential and relative disk files from your BASIC programs.
- Develop a series of subroutines, ranging from mouse monitors to several forms of animation, which you can plug into almost any BASIC program.
- Create new patterns, cursors, and detailed figures with an impressive set of programming utilities.
- And much more.

Advanced Macintosh BASIC Programming contains a wealth of information and almost a hundred example programs, from the sophisticated "Pattern Editor" to the arcade-style game of "Alien." With it alongside your Macintosh, you'll become an expert BASIC programmer.

All the programs in this book are available on a companion disk. See the coupon in the back for details.