LOGO

for the Apple II

TECHNICAL MANUAL

L O G O

for the Apple II

TECHNICAL MANUAL

Harold Abelson and Leigh Klotz, Jr.

Logo Project
Massachusetts Institute of Technology

Terrapin, Inc.
222 Third Street
Cambridge, Massachusetts 02142
(617) 492-8816

Table of Contents

1. Preparing to Use Logo 3

1.1. Configuration 3
1.2. The Keyboard 3
1.3. Loading and Starting Logo 5
1.3.1. On Apples with Autostart ROM 5
1.3.2. On Apples without Autostart ROM 6
1.4. Bugs in the Logo System 6

2. Use of the Logo System 9

2.1. Modes of Using the Screen 9
2.1.1. Nodraw Mode 9
2.1.2. Edit Mode 9
2.1.3. Draw Mode 10
2.2. Editing 12
2.2.1. Line Editor 13
2.2.2. Screen Editor 13
2.3. Using Apple Peripherals 16
2.3.1. Printing Procedures on a Printer 17
2.3.2. Printing Pictures 18
2.4. Color Control 20
2.4.1. Drawing on Colored Backgrounds 20
2.4.2. Drawing without Color Control 21
2.5. The Logo File System 21
2.5.1. Disk Files 22
2.5.2. Saving Pictures 23
2.5.3. Configuring File Diskettes 23

3. Logo System Primitives 25

3.1. Graphics Commands 25
3.2. Numeric Operations 28
3.3. Word and List Operations 29

3.4. Defining and Editing Procedures 31
3.5. Naming 33
3.6. Conditionals 33
3.7. Control 34
3.8. Input and Output 36
3.9. Filing and Managing Workspace 38
3.10. Debugging 39
3.11. Miscellaneous Commands 40

4. The Utilities Disk 43

4.1. Program Descriptions 44
4.1.1. Demonstration Programs 49

5. Changing the Turtle Shape 53

6. Assembly Language Interfaces to Logo 59

6.1. .EXAMINE and .DEPOSIT 59
6.2. Writing Your Own Machine-Language Routines 60
6.3. The Logo Assembler 62
6.3.1. Using the Assembler to Write I/O Routines 63
6.3.2. Syntax of Input to the Assembler 64
6.3.3. Saving Assembled Routines on Disk 67
6.4. Example: Generating music 67
6.5. Useful Memory Addresses 73

7. Miscellaneous Information 77

7.1. Using the Logo System as a Text Editor 77
7.1.1. Printing Files 78
7.2. Self-starting files 79
7.3. Printing to Disk Files 80
7.4. Various System Parameters 82
7.5. Memory Organization Chart 85

Index 87

CHANGES FOR TERRAPIN LOGO VERSION 3.0

Version 3.0 of Terrapin Logo differs from versions 1.0-1.3 in the following respects:

The command to kill a line in the editor is now <CTRL>X instead of <CTRL>K. (This change was necessary to permit use of all four cursor keys.) The most recently deleted line/region can now be Yanked back into the editor with <CTRL>Y.

The <DELETE> key on the Apple IIe can be used to delete backwards one character (just as the <ESC> key does). All four arrow keys may be used in the editor to move the cursor.

Six new primitives have been added: MEMBER?, EMPTY?, ITEM, COUNT, LOCAL, SETDISK

MEMBER? takes two inputs, and outputs "TRUE if the first is a member of the second.
MEMBER? "A "QUARK outputs "TRUE
MEMBER? "A [A B C] outputs "TRUE
MEMBER? "Z [A B C] outputs "FALSE

EMPTY? outputs "TRUE if its input is the empty word or the empty list.
EMPTY? " outputs "TRUE
EMPTY? [ ] outputs "TRUE
EMPTY? "BOB outputs "FALSE

ITEM takes two inputs, a number and a list/word, and outputs the nth element of the second input.
ITEM 3 "TURTLE outputs "R
ITEM 2 [ED RALPH TRIXIE] outputs "RALPH
ITEM 3 [FRED ETHEL [LITTLE RICKY]] outputs [LITTLE RICKY]

COUNT returns the number of elements in its input, which can be either a word or a list.
COUNT "ABC outputs 3
COUNT [SNEEZY GRUMPY SLEEPY DOC] outputs 4
COUNT [SNEEZY GRUMPY [SNOW WHITE]] outputs 3

LOCAL allows the creation of a local variable not declared in the title line, e.g.

```
TO DEMO
 LOCAL "STUFF
 MAKE "STUFF REQUEST
 IF EMPTY? :STUFF PRINT [SHY?] ELSE PRINT :STUFF
END
```

SETDISK takes two inputs, a drive number and a slot number, and directs all subsequent file commands to the specified drive. Default is SETDISK 1 6.

Garbage collection of truly worthless atoms (GCTWA) has been added, with the result that the error message NO STORAGE LEFT! does not crop up inexplicably during long sessions.

The command to recall the previous line in immediate mode, <CTRL>P, can now be used after the REPEAT, RUN, and DEFINE commands as well.

Self-starting files can now be created more easily. Simply create a global variable named STARTUP. Its value should be a list containing the name(s) of the procedure(s) to be automatically run when the file is read in.

For example, if you tell Logo MAKE "STARTUP [FOO] and then save the workspace (SAVE "BAR), Logo will run FOO whenever the file BAR is read in.

PO, SAVE, EDIT, and ERASE can now take a list of procedures as input. The PSAVE utility is thus obsolete; now you may type (SAVE "FILENAME [PROC1 PROC2 PROC3...]) instead. You must type the parentheses.

The DOS primitive no longer supports the MON, NOMON, and VERIFY options. The space following the option name is no longer unnecessary; for instance, typing DOS [BLOADPICTURE] will give an error. Adding a space after BLOAD will correct the problem. Also, addresses given to the DOS primitive may be expressed in either decimal or hexadecimal form.

The changes to the DOS also allow you to read Apple DOS text files. A particular effect of this is that Apple (LCSI) Logo files can be read into Terrapin Logo using the standard READ primitive. (Of course, the procedures in such files will not execute properly until syntax corrections are made.)

Preface

This is a reference manual for an implementation of the Logo system for the Apple II computer. The configuration required to run Logo on the Apple is an Apple II computer with one "floppy disk" drive and 48K bytes of memory, and with an additional 16K memory extension. The system includes the Logo language together with fully integrated interactive "turtle" graphics, screen editor, and disk file system.[1] The present manual assumes that the reader is generally familiar with Logo and Logo programming, and provides technical information about the implementation of Logo for the Apple II, together with the list of primitive commands included in the system.

This implementation of Logo was carried out by the Logo Group at the Massachusetts Institute of Technology, in a project that was partially supported by grants from the National Science Foundation. The interpreter was implemented by Stephen Hain and Leigh Klotz. The text editor, graphics, and file systems were implemented by Patrick Sobalvarro. This work is based on previous Logo systems developed by members of the MIT Logo Group with support from the National Science Foundation, and on a specification of the Logo interpreter developed at MIT with support from Texas Instruments, Inc. The Logo for the Apple II implementation project is under the supervision of Professor Harold Abelson.

---

[1] In addition, the present implementation is designed so that people familiar with programming in assembly language on the Apple can modify and extend the Logo system's capabilities, interface it to new peripherals, and so on. To aid in this process, the Logo disk includes a Logo program written by Leigh Klotz that serves as a 6502 assembler.

Chapter 1
Preparing to Use Logo

This chapter discusses the hardware configuration required to run Logo and points out some uses of the Apple keyboard that are idiosyncratic to the Logo system. It also describes how to load and start Logo on the Apple.

1.1. Configuration

This version of Logo requires an Apple II or Apple II Plus computer, together with a "floppy disk" drive, 48K of memory, and an additional 16K memory extension.[1] The system includes two DOS 3.3 diskettes. One is the Logo Language disk, used for starting Logo. The other is the Logo Utilities diskette, containing the Logo programs described on page 43.

In addition to the Language diskette and the Utilities diskette, you will need diskettes for saving programs that you write. The procedure for creating Logo file diskettes is explained in section 2.5.3.

1.2. The Keyboard

There are a few differences between the way that the Apple keyboard is used in Logo, as opposed to other languages such as BASIC and Pascal. These are explained in the paragraphs below.

The SHIFT key

Logo, like most Apple II systems, uses only uppercase letters, so the shift key is not used for typing capital letters. The shift key is used, however, when a given key contains two different symbols. For instance,

---

[1] Two popular memory extensions that can be used for Logo are the "Language Card" distributed by Apple Computer, Inc. and the RAMCard distributed by Microsoft Consumer Products.

the character "(" is typed as shift-9.

Brackets

Logo uses the open and close bracket characters "[" and "]". These are not marked on any key on the Apple keyboard. They are typed (when using Logo) as shift-N and shift-M, respectively. Included with your Logo package is a set of transparent stickers with brackets and other special characters on them. If you haven't already done so, you should affix these stickers to your keyboard as indicated in the instructions attached to the stickers.

The CTRL key

The key on the lower left of the keyboard marked CTRL (abbreviation for "control") is used to input so-called "control characters". CTRL is used like a shift key. For example, to type "control G" hold down the key marked CTRL and press the "G" key (rather than trying to press both CTRL and "G" simultaneously). Throughout this manual, we specify control symbols by the prefix "CTRL", as in "CTRL-G".

The arrow keys

On the right hand side of the Apple keyboard there are two keys marked with left and right-arrows. These are used in editing, to move the cursor to the left and right.

The ESC key

The key marked ESC (abbreviation for "escape") located at the upper left of the keyboard is also used in Logo editing. When pressed, it deletes the previous character that was typed. You might want to write the word "DELETE" on the ESC key.

The REPEAT key

If you type a character and then hold down the key marked REPEAT, the keyboard will repeatedly transmit the character for as long as REPEAT is held down. REPEAT is occasionally useful in editing, in combination with

the arrow keys.

The RESET key

When using Logo, Do not press the RESET key, ever! The Apple RESET mechanism unfortunately has been designed so as to be incompatible with the use of complex interactive programming languages like Logo, and pressing RESET will abort operation of the Logo system. Over the years, Apple users have come up with ingenious methods for dealing with the frustrating problem that the RESET key is often hit by mistake, even by experienced typists. These solutions range from prying off the key-top to pasting cardboard over the key to buying or constructing plastic shields to put over the key. The newer model Apples also have a switch that disables the RESET key and forces the user to type CTRL-RESET in order to obtain the original RESET function.

Whatever solution you choose, beware that pressing RESET while using Logo will cause you to lose all your work and necessitate reloading the system following the procedure given in section 1.3.

1.3. Loading and Starting Logo

The method for starting Logo differs, depending on whether the Apple computer you are using has "Autostart ROM" (for example, as is standard on the Apple II Plus) or does not (for example, an unmodified Apple II).

1.3.1. On Apples with Autostart ROM

- With the computer turned off, load the Logo diskette into the disk drive and turn the power on.
- A "]" character should appear, followed by the message

```
Loading, please wait...
```

After about 30 seconds, Logo should start and a welcome message:

```
The Terrapin Logo Language
(C) 1981 MIT
(C) 1982 Terrapin, Inc.
Welcome to Logo
```

should appear on the screen, followed by a line beginning with a question mark indicating that Logo is ready to accept commands.

- Remove the Logo disk from the drive and put it in a safe place.

1.3.2. On Apples without Autostart ROM

- Place the Logo Language diskette in the disk drive and turn on the power.
- Type 6 CTRL-P and press return.

1.4. Bugs in the Logo System

Logo is, to our knowledge, the most complex and extensive program written for the Apple II, and Version 1.1 still contains a few very obscure bugs that may cause the system to crash. The clearest symptom of a bug is when the computer stops executing Logo and instead returns to the Apple monitor with the message

```
CONGRATULATIONS! YOU FOUND A BUG!
TYPE 300.311 <RETURN> AND WRITE DOWN
THE RESULT.  THEN TYPE CTRL-Y <RETURN>.
```

If you wish to report the bug, write down the indicated information together with what you were doing in Logo at the time.

In most cases, operation of the Logo system can be successfully continued by using the Apple monitor to restart Logo: type CTRL-Y and RETURN.[2] But even if this works, you should not assume that all is well.

---

[2] This monitor command restarts Logo at the "warm boot" address. See the addresses listed with .BPT on page 40.

The safest thing to do is to immediately attempt to save your workspace in some temporary file; then reload Logo from disk and read your procedures back in. For very serious bugs, the CTRL-Y method may not work, in which case, the only safe recourse is to restart Logo using the procedure given in 1.3.

Chapter 2
Use of the Logo System

The Logo system includes a full interpreter for the Logo language, a complete text editor for editing procedure definitions, and an integrated "turtle graphics" system. This chapter provides notes on how these different functions interact.

2.1. Modes of Using the Screen

The Logo system uses the display screen in three different ways, or "modes."

2.1.1. Nodraw Mode

This is the mode in which the system starts. Logo prompts the user for a command with a question mark, followed by a blinking square called the "cursor." You may type in command lines, terminated with RETURN. Logo executes the line and prints a response, if appropriate.

Whenever the cursor is visible and blinking, Logo is waiting for you to type something, and will do nothing else until you do.

The system includes a flexible line editor that allows you to correct any typing errors in a command line which you have typed in DRAW or NODRAW mode. The available editing operations are the ones described on page 14 corresponding to the keys: ESC, arrow keys, CTRL-A, CTRL-D, CTRL-E, CTRL-G, CTRL-K, CTRL-P.

2.1.2. Edit Mode

Executing the commands TO or EDIT places Logo in edit mode. For example, if you enter Logo and type

```
TO POLY :SIDE :ANGLE
```

followed by RETURN, the system will enter the screen editor with the typed

line of text on the screen. Logo indicates that it is in EDIT mode by printing "EDIT: Ctrl-C to define, ctrl-G to abort" in reverse-color letters at the bottom of the screen.

At this point you can use all of the editing operations described on page 14 to create and/or edit the text for the procedure. Typing CTRL-C will exit the editor, cause the procedure to be defined according to the text you have typed, and enter nodraw mode. Typing CTRL-G aborts the edit. Logo will return to nodraw mode without any procedures being defined. If you begin editing a procedure, and decide that you don't want to change it after all (or would like to start over), type CTRL-G. The procedure you were editing will not be changed.

In this mode, RETURN is just another character which causes the cursor to move to the next line. In EDIT mode, CTRL-C causes Logo to evaluate the contents of the edit buffer just as RETURN in DRAW and NODRAW modes causes Logo to evaluate the line just typed. See section 2.2.

To edit the most recently defined procedure, type just EDIT (or its abbreviation, ED).[1]

2.1.3. Draw Mode

In draw mode, you use the turtle for drawing on the screen. If you attempt to execute any turtle command while in nodraw mode, the system will enter draw mode before executing the command. The NODRAW command (abbreviated ND) exits draw mode and enters nodraw mode. Actually, there are different types of draw mode.

---

[1] In DRAW mode, this edits the current definition of the procedure most recently defined or PO'd. In NODRAW mode, however, typing EDIT with no inputs returns to EDIT mode with the contents of the edit buffer intact. For example, after a READ or SAVE, everything read or saved will be in the edit buffer. If you had aborted the definition of a procedure with CTRL-G, the edit buffer's contents at the time you typed CTRL-G will still be in the edit buffer; you can retrieve it by typing EDIT not followed by a procedure name. Typing EDIT followed by the procedure name would edit the procedure as it was last defined. This is all very complicated, but is entirely intuitive.

Splitscreen mode is the normal way in which draw mode is used. Four lines at the bottom of the screen are reserved for text, and the rest of the screen shows the field in which the turtle moves. The turtle field actually extends to the bottom of the screen and so is partially masked by the four-line text region. In fullscreen mode the text region disappears and you can see the entire turtle field. You can still type commands, but they will not be visible. If the system needs to type an error message, it will first enter splitscreen mode so that the message will be visible.

You can use the characters CTRL-F and CTRL-S to switch back and forth between splitscreen and fullscreen mode. Pressing CTRL-F while in splitscreen mode will enter fullscreen mode. Pressing CTRL-S will restore splitscreen mode. It is also sometimes convenient to be able to switch back and forth under program control. The commands SPLITSCREEN and FULLSCREEN are provided for this purpose.

In draw mode, Logo displays just four lines of text. This is frequently an inconvenience, since error messages are sometimes longer than four lines. If you type CTRL-T while in graphics mode, the turtle picture will disappear and you can use the entire screen for text, just as in nodraw mode. The difference is that you are actually still in draw mode: turtle commands can be executed, although you will not see the picture being drawn. The CTRL-T command is especially useful when an error message in DRAW mode is more than four lines long. CTRL-T is equivalent to the TEXTSCREEN primitive. The only way to make the graphics screen visible after using CTRL-T is to type CTRL-F to return to fullscreen mode, or CTRL-S to go back to splitscreen mode.

TEXTSCREEN is different from NODRAW. NODRAW clears the text screen, clears the graphics screen, and resets all the graphics parameters (pencolor, turtle visibility, pen state, background color, and wrapping mode).

Here is a list of control characters not related to editing functions. All are available in draw mode and nodraw mode. Some exist in edit mode, also, and are specially indicated.

Non-editing Control Characters

CTRL-F	In graphics mode, gives full graphics screen.
CTRL-G	In edit mode, exits the editor without processing the edited text. In draw or nodraw mode, stops execution and returns control to toplevel.
CTRL-S	In graphics mode, gives mixed text/graphics screen.
CTRL-T	In graphics mode, gives full text screen.
CTRL-W	Stops program execution. Repeatedly typing CTRL-W will cause Logo to stop after printing the next line (or the next list element if lists are being printed). Typing any character other than CTRL-W or CTRL-G will resume normal processing. Try CTRL-W in conjunction with the repeat key to obtain "slow motion" effects. See TRACE (page 39).
CTRL-Z	Causes Logo to pause. You may type anything and Logo will execute it as if it were a line of the current procedure. Type CO or CONTINUE to continue.
CTRL-SHIFT-M	Restores output to the screen. See OUTDEV, page 36.
CTRL-SHIFT-P	This generates an underscore character. It is a regular printing character, available all three modes.

2.2. Editing

The Logo system contains a fully-integrated screen editor, and a compatible line editor. The screen editor is used for defining Logo procedures in EDIT mode, and the line editor is used for typing Logo commands to be executed in DRAW and NODRAW modes.

2.2.1. Line Editor

While you are typing a line of characters to Logo, you can ignore the line editor until you need it. If you mistype a character, you can rub it out with the ESCAPE key. If you forget to put a word at the beginning of the line, you place the cursor there with CTRL-A and type. The characters will push the rest of the line to the right; nothing will be lost or overwritten. If you want to insert characters anywhere in the line, simply move the cursor there with the arrow keys, and type what you want. To go to the end of the line, type CTRL-E. To delete the character at the cursor, use CTRL-D.

To end the line and have Logo act upon it, type RETURN. It is not necessary for the cursor to be at the end of the line; all characters you see on the line will be read by Logo. To delete all characters from the cursor to the end of the line, use CTRL-K.

Lines typed to Logo may wrap around to the next screen line. The editing commands will still work on them exactly as if the line did not spread over more than forty characters.

Logo remembers the most recently typed line in both draw and nodraw modes so that you can insert it into the current line by typing CTRL-P. Unfortunately, in the current implementation, RUN, REPEAT, DEFINE, and all filing commands, such as SAVE, cause Logo to forget the last line typed.

2.2.2. Screen Editor

For defining procedures, Logo has a screen editor which you enter by typing EDIT, ED or TO, followed by the name of the procedure you wish to define.

Once Logo is in "edit mode" the characters you type will appear on the screen. Pressing RETURN will cause the cursor to move down to the next line (If the cursor was not at the end of the line, it will split the current line into two lines.) It will not cause Logo to execute the line.

Various other commands are available for editing the line on which the cursor appears, and moving to other lines. To move to the Next line, type CTRL-N. To move to the Previous line, type CTRL-P.

Lines may be of any length, as long as they fit in the edit buffer. Lines which are longer than 40 characters "wrap around" the screen. You can tell they are continued lines because an exclamation point ("!") is shown in the last screen column. This mark is not a part of the procedure being typed, and serves only as a reminder that the line does not actually end at that point.[2]

Although Logo procedures are seldom more than a few lines long, the text you may edit is not limited to one screen page. If the text you are typing begins to overflow the current page, the system will automatically shift the display so that the current line is in the middle of the screen. If you type CTRL-P or CTRL-N and move to either the top or bottom edge of the screen, the next page will appear. The CTRL-F key inside edit mode moves immediately to the next page of text. To move back to the previous page, type CTRL-B. If you are on the first page, CTRL-B will move to the top of it; similarly, on the last page, CTRL-F will move to the end. If the text you are editing is more than one page long, you can use the CTRL-L command to center the current line on the screen.

To exit the editor and have the procedure you typed be defined, type CTRL-C. To exit without having the procedure defined, type CTRL-G. After typing CTRL-G, you can return to the editor with ED or EDIT, and have all the text still there.[3]

The text you type in the editor doesn't have to be a procedure. It can be any Logo commands which are not graphics or file-system commands. See Section 7.1 to find out how to use Logo for editing text, saving it on disk, and printing it.

Here is a summary of the editing commands available:

---

[2]There is a slight difference here between edit mode and draw or nodraw mode; only edit mode displays the exclamation marks.

[3]Providing you didn't use graphics or filing in the meantime

Keyboard Editing Commands

ESC	Rubs out the character immediately to the left of the cursor and moves the cursor one space to the left.
arrow keys	Moves the cursor one character to the left (or right), without rubbing out any character.
CTRL-A	Moves the cursor to the beginning of the current line. CTRL-A was chosen for this command because it lies at the beginning of a row of the keyboard, and is the first letter in the alphabet.
CTRL-B	When editing more than one screenful of text, moves the cursor one screenful of text backwards, or to the beginning of the buffer if not that much text precedes the cursor.
CTRL-C	Exits the editor. Processes the edited text.
CTRL-D	Deletes the character at the current cursor position, that is, the character over which the cursor is flashing.
CTRL-E	Moves the cursor to the end of the current line.
CTRL-F	When editing more than one screenful of text, moves the cursor one screenful of text forward, or to the end of the buffer if not that much text follows the cursor.
CTRL-G	In edit mode, exits the editor without processing the edited text. In all modes, stops execution and returns control to toplevel.
CTRL-K	Deletes all characters on the current line to the right of the cursor. This is known as killing a line of text.
CTRL-L	In edit mode, scrolls the text so that the line containing the cursor is at the center of the screen.
CTRL-N	Moves the cursor down to the next line.
CTRL-O	Opens a new line at the cursor position. That is, CTRL-O is equivalent to typing RETURN and then CTRL-P. It is most useful for adding new lines in the middle of procedures.

CTRL-P — In edit mode, moves the cursor to the previous line. In draw or nodraw mode, retrieves previous input line so that it can be edited and/or re-executed.[4]

2.3. Using Apple Peripherals

Logo's ordinary input and output operations deal with the Apple keyboard, the screen, and one disk drive. There are also commands for reading input from up to four game paddles that can be attached to the Apple. (See the PADDLE and PADDLEBUTTON primitives.) In addition, Logo provides the OUTDEV primitive for accessing output devices other than the screen. This command takes one input that specifies a slot on the Apple board at which a peripheral interface card should be attached.[5] The OUTDEV command causes any subsequent output that would normally go to the Apple screen to be directed at the device in the specified slot. Unlike the BASIC PR # command, OUTDEV does not direct typein to the alternate device. The Logo screen editor and top-level line editor are unaffected. Using OUTDEV with an input of 0 will reset the output device to screen.

Typing CTRL-SHIFT-M will redirect output to the screen. It is equivalent to executing an OUTDEV 0, but takes effect immediately, even if Logo is in the process of printing something to a printer.[6]

---

[4] In draw and nodraw mode, REPEAT, RUN, DEFINE, and all file-system commands cause Logo to forget the previous line. Additionally, due to line-length restrictions, Logo might not remember all of the previous line if it was very long.

[5] It is also possible to use OUTDEV to designate a user-supplied assembly language routine that should be called in place of the normal character output routine. See section 6.3.1.

[6] But not if the Apple is "hung" waiting for the printer to receive a character. Typically, this condition occurs when the printer is off or otherwise disabled.

2.3.1. Printing Procedures on a Printer

The following examples will work if you have a printer attached to Apple slot 1.

To obtain a paper printout of a procedure called CIRCLE, you could type

```
OUTDEV 1
PRINTOUT CIRCLE
OUTDEV 0
```

Alternatively, you could type

```
HPO "CIRCLE
```

after you define HPO as

```
TO HPO :PROCEDURENAME
 OUTDEV 1
 RUN LIST "PRINTOUT :PROCEDURENAME
 OUTDEV 0
END
```

HPO means "hardcopy printout." HPO "ALL will list all procedures and names. The following procedure takes a list of procedures as an input and prints them out on the printer in the order they appear in the list. It's useful for final listings of programs where you want the procedures printed out in a certain order.

```
TO HPL :LIST
 IF :LIST=[] STOP
 HPO FIRST :LIST
 HPL BF :LIST
END
```

Once you have defined HPO and HPL, this command line will print out the procedures BIRD, HEAD, WINGS, TAIL, and LEGS, in that order:

```
HPL [BIRD HEAD WINGS TAIL LEGS]
```

To list all the procedures, but no names, type

```
HPO "PROCEDURES
```

2.3.2. Printing Pictures

Many Apple-compatible printers support printing pictures ("screen dumps"). Before you can print a copy of the turtle-graphics screen on your printer, there are a few things you need to know.

Since the aspect ratio (squareness of the dots that make up the image) of a printer is different from that of a video monitor or television, figures that look square on the screen will come out rectangular when printed on paper. If you are producing output especially for printing, you might want to determine the proper aspect ratio to use with your printer. Frequently you can compromise by setting the aspect ratio to be between that of the monitor and that of the printer, with negligible bad effects. See the description of the .ASPECT primitive, page 40.

Saving pictures on disk and printing them later is by far the most common (and inexpensive) method of obtaining screen hardcopy. You can store pictures on disk with the SAVEPICT (p. 23) command. SAVEPICT saves the picture as an Apple DOS binary file to which ".PICT" is appended to differentiate it from other files. The turtle-graphics screen is stored in memory in the primary high-resolution graphics page.

Commercially available programs will load these picture files and print them out on a wide variety of printers. The GRAPHTRIX package[7] is commonly available in computer stores and is easy to use.

Orange Microsystems Grappler

The Grappler interface card connects the Apple II to many popular printers. Different versions of the interface are required for different printers; however, the following procedure will print the screen on any of them, assuming the interface card is in slot 1.

```
TO HC     ;hardcopy of graphics screen
 OUTDEV 1
 (PRINT1 CHAR 9 "G CHAR 13 )
 OUTDEV 0
END
```

---

[7] written by Data Transforms, 906 E. Fifth Ave., Denver CO 80218.

The Grappler will print pictures with various options, including reverse color, rotation, and magnification. The characters specifying these modes should be placed immediately after the "G above. Refer to the Grappler card manual for a detailed description of Grappler modes and options.

Silentype

The Silentype printer, available from Apple Computer dealers, has the built-in capability to print pictures. Assuming that the printer is in slot 1, the following Logo procedure will print the screen:

```
TO HC     ;Silentype hardcopy of graphics screen.
 OUTDEV 1
 PRINT1 CHAR 17
 OUTDEV 0
END
```

The ascii character with code 17 is a request to the Silentype to print the screen. You may find it more attractive to print the screen in "inverse" mode, or to use other options. See the Silentype documentation for the appropriate control information. Experimentation has shown the following set of options to work well:

```
TO HC     ;improved version for Silentype
 OUTDEV 7
 .DEPOSIT 53008 7     ;set printer to darkest print
 .DEPOSIT 53007 128   ;set to unidirectional printing
 .DEPOSIT 53012 0     ;set to inverse printing
 PRINT1 CHAR 17       ;print the screen
 .DEPOSIT 53007 0     ;turn off unidirectional printing
 OUTDEV 0
END
```

IDS Color Printer

Use the Prism Print software (Integral Data Systems order number 9100-002-644) for printing saved pictures. There is currently no interface for printing color pictures without leaving Logo.

2.4. Color Control

If you have a color TV monitor, you can use the PENCOLOR command (abbreviated PC) to change the color of the lines that the turtle draws. You can also use the BACKGROUND command (abbreviated BG) to make the turtle draw on backgrounds of various colors. Both PENCOLOR and BACKGROUND take a number 0 through 6 as input. The correspondence of colors to numbers is as follows:[8]

number	color
0	black
1	white
2	green
3	violet
4	orange
5	blue

Drawing with PENCOLOR 6 "reverses" the color of all dots that the turtle passes over. The actual color produced depends on both the background color and the color of the dot the turtle is passing over but in all cases, reversing the color of a dot and then reversing it again will restore the original color. PENCOLOR 6 is most useful with black-and-white graphics.

If you don't explicitly give any BACKGROUND or PENCOLOR commands, Logo will default to BACKGROUND 0 and PENCOLOR 1.

2.4.1. Drawing on Colored Backgrounds

When drawing on a colored background (2 through 5), only two of the four colors -- green, violet, blue, orange -- are available. When the background is green or violet, blue and orange cannot be used: PENCOLOR 4 will draw in green and PENCOLOR 5 will draw in violet.

---

[8]The actual color that appears on the screen corresponding to any of these color names can vary greatly depending on the adjustment of the TV monitor. Also, if you have a black and white monitor, the "colors" will appear as striped vertical lines.

When the background in blue or orange, PENCOLOR 2 will draw in orange and PENCOLOR 3 will draw in blue. Also, if you draw a picture on the screen and then change the background color, the colors of the lines in the picture may change, or the lines may become distorted in unexpected ways; however, returning to the background color in which the lines were drawn will always restore their original appearance.[9]

2.4.2. Drawing without Color Control

In order to obtain clear colors with the Apple computer, the Logo system must draw lines more thickly than would otherwise be necessary. This means that drawings will not look as precise as they could if one drew only thin lines. If you don't care about color, you will obtain better looking drawings by using thin lines. To do this, select BACKGROUND 6. In BACKGROUND 6, PENCOLOR 0 gives black, 1 through 5 give "white," and 6 gives "reverse." The reason that "white" is in quotes is that "white" lines may not always appear white on a color monitor.[10]

2.5. The Logo File System

The Logo file system allows you to save procedure definitions on floppy disk. A user may have many files on a single disk, and the files are distinguished by the fact that they are named. The names of the files are listed in the disk catalog.

---

[9] These strange effects are the result of a compromise with the Apple computer color system, which does not allow, for example, green dots to appear very close to orange dots.

[10] In particular, "white" vertical lines will be either red or green, depending on their position.

2.5.1. Disk Files

When you use Logo, you should normally have a Logo file diskette mounted in the disk drive. File diskettes may be created as described in section 2.5.3 below. If you want to save your procedure definitions, use the SAVE command. For example,

```
SAVE "MYSTUFF
```

will save all the procedure definitions and names currently in workspace in a file named MYSTUFF. There can be both a procedure and a file with the same name, however SAVE saves everything in the workspace and will not save only the procedure by that name. If you already had a file of that name, the old one will be deleted. The READ command takes a file name as input and reads the procedures and names from that file into the workspace. The procedures and names will be added to the ones currently in workspace.[11]

Notice that the file names given as inputs to SAVE and READ are preceded by a quote and have no following quote.

The CATALOG command lists all the files on the disk. Logo workspace files will be listed with the characters ".LOGO" appended to the name. For example, the file created by

```
SAVE "MYSTUFF
```

will be listed in the catalog as MYSTUFF.LOGO. Do not include the .LOGO part of the name when you use the READ or SAVE commands.

To remove a file from the disk, use the ERASEFILE command, which takes as input the name of the file to be erased.

---

[11] Logo filing makes use of the same memory area as for drawing pictures and editing procedures. Issuing any filing command while in draw mode will first move you to nodraw mode.

2.5.2. Saving Pictures

In addition to saving procedure definitions, Logo also allows you to save a graphics screen image on the disk, so that it can be read back in and displayed. To do this, use SAVEPICT and READPICT.

SAVEPICT, which is similar to SAVE, takes a name as input. It saves on the disk the picture currently on the turtle graphics screen. (SAVEPICT should only be done when you are in graphics mode.) READPICT reads in a picture that was saved by SAVEPICT, and displays this picture on the screen. When you do a CATALOG you will notice that ".PICT" is appended to picture files just as the ".LOGO" is appended to regular Logo files. Do not include the ".PICT" part of the name when you use the READPICT command. (However, you do need the ".PICT" if you access the file from outside of Logo.)

To erase a picture from the disk, use the ERASEPICT command, which takes as input the name of the picture to be erased.

You can use any of a number of commercial software packages for printing saved picture files on printers. Logo picture files are stored in the standard binary file format. In memory, they are located in the primary graphics page.

2.5.3. Configuring File Diskettes

Logo files are saved on regular floppy diskettes that have been appropriately configured. Here are instructions for users with Applesoft BASIC Apples to create Logo file diskettes.

1. Put the Logo Utilities disk into the Apple and switch on the power. The Utilities disk is not the disk that you use to start Logo, but is the one containing the demonstration and utility programs.
2. Type

   ```
   LOAD HELLO
   ```

   and press RETURN and wait for the computer to stop.
3. Remove the Utilities disk from the drive and insert a new disk.

Beware that this disk will be re-initialized and any previous information on it will be destroyed.

4. Type

```
INIT HELLO
```

Press RETURN and wait. When the computer stops (after about a minute) the disk has been initialized and can be used for storing Logo files.

Users of Integer BASIC Apples should use the following method for creating file diskettes:

1. Remove the write-protect tab from the Utilities Disk and insert it in the disk drive. Turn on your Apple. It will print "LANGUAGE NOT AVAILABLE" and a prompt (">"). Type the following commands:

```
10 PRINT "CTRL-D CATALOG"
UNLOCK HELLO
DELETE HELLO
SAVE HELLO
LOCK HELLO
```

2. IMPORTANT: Remove the Utilities Diskette from the drive, and replace its write-protect tab.
3. Place the blank diskette in the drive and proceed with step 2 of the Applesoft instructions above.

Chapter 3
Logo System Primitives

Logo is a full-scale, powerful computer language. It includes commands for graphics, arithmetic operations and list processing. It also incorporates a real-time screen editor that can be used both for editing command lines as they are typed and for editing procedure definitions.

3.1. Graphics Commands

BACK	Moves the turtle in the opposite direction from which it is pointing by the amount specified. Abbreviated: BK.
BACKGROUND	Takes a number 0 through 6 as input and sets the color of the graphics screen background as described in section 2.4. Abbreviated: BG.
CLEARSCREEN	Clears the graphics screen. Does not change the turtle's position, the pen state, or the turtle being hidden or shown. Abbreviated: CS.
DRAW	Clears the graphics screen, homes the turtle to the center of the screen, shows the turtle, and puts the pen down. It does not change the background or pen color.
FORWARD	Moves the turtle in the direction in which it is pointing by the amount specified. Abbreviated: FD.
FULLSCREEN	In graphics mode, gives full graphics screen. Complementary to SPLITSCREEN. Equivalent to interrupt character CTRL-F.
HEADING	Outputs the turtle's heading as a decimal number. The heading ranges from 0 to less than 360. When the turtle has a heading of 0 it is pointing straight up.
HIDETURTLE	Makes the turtle pointer disappear. Abbreviated: HT.
HOME	Moves the turtle to the center of the screen, pointing

straight up.

LEFT — Rotates the turtle. Takes an input that specifies the number of degrees to rotate. Abbreviated: LT.

NODRAW — Exits graphics mode, giving a clear text page with the cursor homed in the upper left-hand corner of the screen. Abbreviated: ND.

NOWRAP — Exits wrapping mode. Any command that would normally cause the turtle to move off one edge of the screen and onto the opposite edge instead results in an error.

PENCOLOR — Takes a number from 0 through 6 and sets the color of the lines that the turtle will draw as described in section 2.4. Abbreviated: PC.

PENDOWN — Causes the turtle to leave a trail when it moves. This is the default state and it is changed by PENUP. Abbreviated: PD.

PENUP — Causes the turtle to move without leaving a trail. Abbreviated: PU.

RIGHT — Rotates the turtle. Takes an input that specifies the number of degrees to rotate. Abbreviated: RT.

SETHEADING — Rotates the turtle to the direction specified. Input determines number of degrees. Zero is straight up, with heading increasing clockwise. Abbreviated: SETH.

SETX — Moves the turtle horizontally to the specified coordinate.

SETXY — Takes two numeric inputs. Moves the turtle to the specified point. *0,0* is screen center. When the y-coordinate (second input) is negative, it must be enclosed by parentheses.

SETY — Moves the turtle vertically to the specified coordinate.

SHOWTURTLE — Makes the turtle pointer appear. This is the default state and it is changed by HIDETURTLE. Abbreviated:

ST.

SPLITSCREEN — In graphics mode, gives mixed text/graphics screen, which is the default state. Complementary to FULLSCREEN. Equivalent to interrupt character CTRL-S.

TEXTSCREEN — In graphics mode, gives full text screen. See SPLITSCREEN, FULLSCREEN. Equivalent to interrupt character CTRL-T.

TOWARDS — Takes two numbers as inputs. These are interpreted as the *x* and *y* coordinates of the point on the screen. TOWARDS outputs the heading from the turtle to the point. That is, SETHEADING TOWARDS :X :Y will make the turtle face towards point *x,y*. Compare with ATAN.

TURTLESTATE — Takes no inputs. Outputs a list of four items giving information about the state of the turtle. The format of the list is as follows: The first element is TRUE or FALSE for pen down or pen up, then TRUE or FALSE for show or hide turtle, then background color, then pen color. Abbreviated TS.

WRAP — Places the graphics system in wrapping mode. Any time the turtle moves off the edge of the screen, it reappears at the opposite edge. Wrap mode is the default, and is exited only by the NOWRAP command.

XCOR — Outputs the turtle's x-coordinate as a decimal number.

YCOR — Outputs the turtle's y-coordinate as a decimal number.

3.2. Numeric Operations

+	Addition
–	Subtraction (two inputs) and negation (one input).
*	Multiplication
/	Division (always outputs a decimal value).
>	Outputs TRUE if its first input is greater than its second, FALSE otherwise.
<	Outputs TRUE if its first input is less than its second, FALSE otherwise.
ATAN	Takes two inputs and then outputs (in degrees) the arctangent of the quotient. The output ranges from 0 to less than 360, with the quadrant corresponding to the signs of the two inputs. If the second input is negative, it must be enclosed by parentheses.
COS	Outputs the cosine of its input (as an angle in degrees).
INTEGER	Takes one numeric input and outputs the integer part, ignoring the fractional part.
NUMBER?	Outputs TRUE if its input is a number. See also WORD? and LIST?.
QUOTIENT	Outputs the integer quotient of its two inputs. (If the inputs are not integers, it first rounds them to the nearest integer.) If the second input is negative, it must be enclosed by parentheses.
RANDOM	Takes one input -- a positive integer n -- and outputs an integer between 0 and n-1. Identical sequences of calls to RANDOM will yield repeatable sequences of random numbers each time Logo is restarted unless the seed for the random number generator is RANDOMIZEd.
RANDOMIZE	Randomizes the seed for RANDOM. If given an explicit input, sets the random number seed to that

number. For example, after each execution of (RANDOMIZE 259) the same sequence of random numbers will be generated. Different numbers result in different sequences. Note that ( ) are needed around RANDOMIZE if an input is used, such as (RANDOMIZE 259) above.

REMAINDER — Outputs the integer remainder of its first input divided by its second. (If the inputs are not integers, it first rounds them to the nearest integer.) If the second input is negative, it must be enclosed by parentheses.

ROUND — Outputs the nearest integer to its input.

SIN — Outputs the sine of its input (as an angle in degrees).

SQRT — Takes a positive number as input and outputs the square root of that number.

3.3. Word and List Operations

= — If both inputs are numbers, compares them to see if they are numerically equal. If both inputs are words, compares them to see if they are identical character strings. (In this case, a space is needed before the = sign.) If both inputs are lists, compares them to see if their corresponding elements are equal. Outputs TRUE or FALSE accordingly.

BUTFIRST — If the input is a list, outputs a list containing all but the first element. If the input is a word, outputs a word containing all but the first character. Abbreviation: BF. Gives an error when called with the empty word or the empty list.

BUTLAST — If input is a list, outputs a list containing all but the last element. If input is a word, outputs a word containing all but the last character. Abbreviation: BL. Gives an error when called with the empty word or the empty

list.

FIRST — If input is a list, outputs the first element. If input is a word, outputs the first character. Gives an error when called with the empty word or the empty list.

FPUT — Takes two inputs. Second input must be a list. Outputs a list consisting of the first input followed by the elements of the second input. Therefore, if the first input is a list, for example FPUT [A B][C D], the result will be [[A B]C D]. See also LPUT, LIST, and SENTENCE.

LAST — If input is a list outputs the last element. If input is a word, outputs the last character. Gives an error when called with the empty word or the empty list.

LIST — Takes a variable number of inputs (two by default) and outputs a list of the inputs. Therefore, if the first and second inputs are lists, for example LIST [A B] [C D], the result will be [[A B][C D]]. See also FPUT, LPUT, and SENTENCE. If there are more than two inputs, there must be an opening parenthesis before LIST, and a space and closing parenthesis after the last input.

LIST? — Outputs TRUE if its input is a list. See also WORD? and NUMBER?.

LPUT — Takes two inputs. Second input must be a list. Outputs a list consisting of the elements of the second input followed by the first input. Therefore, if the first input is a list, for example LPUT [A B] [C D], the result will be [C D[A B]]. See also FPUT, LIST, and SENTENCE.

SENTENCE — Variable number of inputs (default 2). If inputs are all lists, combines all their elements into a single list. If any inputs are words (or numbers), they are regarded as one-word lists in performing this operation. Therefore, if the first and second inputs are lists, for

example SENTENCE [A B] [C D], the result will be [A B C D]. If there are more than two inputs, there must be an opening parenthesis before SENTENCE, and a space and closing parenthesis after the last input. See also FPUT, LPUT, and LIST. Abbreviated: SE.

WORD — Variable number of inputs (default is 2). Outputs a word that is the concatenation of the characters of its inputs (which must be words). If there are more than two inputs, there must be an opening parenthesis before WORD, and a space and closing parenthesis after the last input.

WORD? — Outputs TRUE if its input is a word. Since numbers are treated as words, the result will also be TRUE for a number. See also LIST? and NUMBER?.

3.4. Defining and Editing Procedures

DEFINE — Takes two inputs. First is a name, second is a list. Each element of this list must be a list itself. The first element of the list is a list of inputs to the procedure. (If there are no inputs to the procedure, the first element should be the empty list.) Each subsequent element is a list corresponding to one line of the procedure being defined. For example, DEFINE "TRIANGLE [[:SIZE][REPEAT 3[FD :SIZE RT 120]]]. See TEXT. Note that one normally uses TO rather than DEFINE in order to define procedures.

EDIT — Enters edit mode. If a procedure name is included as an input, that procedure will be in the editor. If no input is specified, enters edit mode with the previous contents of the screen editor buffer, or the most recently defined (or PO'd) procedure if the previous contents are unretrievable. Can also take auxiliary words: ALL, NAMES, PROCEDURES. See page 12 for

a description of keystroke commands inside the editor. Abbreviated: ED.

END — Terminates a procedure definition that is typed in to the editor. It is not necessary to type END at the end of the final definition. But if you are defining more than one procedure at a time, the separate procedure definitions must be separated by END statements.

ERASE — Erases designated procedure from workspace. Can also take qualifiers ALL, NAMES, PROCEDURES. Signals an error if there is no procedure with the given name. For convenience, the input to erase is not evaluated (i.e. Logo will not try and run the procedure being erased.); to erase a procedure called LOOKUP, type ERASE LOOKUP. Abbreviated: ER. To erase a list of procedures, the following procedure can be used.

```
TO ERPROCS :PROCLIST
 IF :PROCLIST = [] STOP
 RUN LIST "ERASE FIRST :PROCLIST
 ERPROCS BUTFIRST :PROCLIST
END
```

ERNAME — Takes a name as input and removes that name from the library. Signals an error if the name is not used. Note that unlike ERASE, the input to ERNAME is evaluated. Thus, to erase the name TEMP, type ERNAME "TEMP. If ERNAME TEMP is typed, TEMP is assumed to be a procedure and Logo tries to run it.

TEXT — Takes a procedure name as input and outputs procedure text as a list. The procedure name must start with " or Logo will run the procedure. If the procedure has not been defined, TEXT outputs []. If instead the input is the name of a Logo primitive, it outputs the primitive's name (i.e., the input). See DEFINE.

TO — Begins procedure definition. Takes a variable number of inputs. Enters edit mode with the procedure named by the first input. Any following inputs are taken as inputs to the procedure named by the first input. With no inputs at all, TO enters edit mode with an empty edit buffer.

3.5. Naming

MAKE — Takes two inputs, the first of which must be a word. It treats the word as a variable, and makes the second input be the value (thing) of the variable.

THING — Outputs the value of its input, which must be a word. Note that this gives an "extra level" of evaluation. THING "XXX is equivalent to :XXX.

THING? — Outputs TRUE if its input has a value associated to it.

3.6. Conditionals

ALLOF — Takes a variable number of inputs (default is two) and outputs TRUE if all are TRUE. If there are more than two inputs, there must be an opening parenthesis before ALLOF, and a space and a closing parenthesis after the last input.

ANYOF — Takes a variable number of inputs (default is two) and outputs TRUE if at least one is TRUE. If there are more than two inputs, there must be an opening parenthesis before ANYOF, and a space and a closing parenthesis after the last input.

ELSE — Used in IF ... THEN ... ELSE.

IF — Used in the basic conditional form IF <condition> THEN <action1> ELSE <action2>. The <condition> is tested: If it is true <action1> is performed. If it is false

<action2> is performed. The word THEN is optional. The ELSE <action2> part need not be present. The <condition> must be a Logo expression which outputs "TRUE or "FALSE. A Logo variable whose value is "TRUE or "FALSE satisfies this condition, as do various testing functions such as <, >, =, and NOT. Both <action1> and <action2> may be any number of Logo expressions.

IFFALSE — Executes rest of line only if result of preceding TEST was false. Abbreviated: IFF

IFTRUE — Executes rest of line only if result of preceding TEST was true. Abbreviated: IFT

NOT — Outputs TRUE if its input is FALSE, FALSE if its input is TRUE.

TEST — Tests a condition to be used in conjunction with IFTRUE and IFFALSE. TEST takes one input, which must be either TRUE or FALSE. The result of the most recent TEST in each procedure is used by IFTRUE and IFFALSE, and is local to the current procedure.

THEN — Used with IF ... THEN ... ELSE ...

3.7. Control

GO — Takes a word as input and transfers to the line with that label. You can only GO to a label within the same procedure. Labels are defined by typing them at the beginning of the indicated line followed by a colon. (GO is very rarely used in Logo programming.)

GOODBYE — Clears workspace and restarts Logo.[1]

OUTPUT — Takes an input. Causes the current procedure to stop

[1] But it does not clear the user machine-language area.

and output the input to the calling procedure. If the input has to be evaluated, it outputs the result of that evaluation. Abbreviated: OP.

REPEAT — Takes a number and a list as input. RUNs the list the designated number of times.

RUN — Takes a list as input. Executes the list as if it were a typed in command line. Note: the number of characters in the list (i.e., the number of characters you would get if you printed it) given to RUN must not exceed the maximum number of characters allowed in the top-level command line, 255. Otherwise, an error is signalled. (ERASE and PO must be treated differently than other commands when they follow RUN. For an example, see the listing for ERASE and PO.)

STOP — Causes the current procedure to stop and return control to the calling procedure.[2]

TOPLEVEL — Aborts the current procedure and all calling procedures and returns control to toplevel.[3] It is not used very often in Logo programming.

---

[2]STOP does not mean the same thing as END. STOP is a primitive which when executed causes the current procedure to stop executing, and returns control to the previous procedure (or toplevel). END is used in the editor to indicate where a procedure ends. It is never executed.

[3]Note the difference between TOPLEVEL and STOP. STOP stops just the current procedure and continues execution with the calling procedure, whereas TOPLEVEL aborts execution of the whole program.

3.8. Input and Output

OUTDEV — Takes as input a number designating a slot on the Apple card. After executing this command, everything except typein will be sent to the device plugged in to the designated slot rather than to the screen. OUTDEV 0 specifies output to the screen. A "slot number" number greater than 255 is interpreted as the address of a user-supplied assembly language routine to be called in place of the usual character output primitive. See section 6.3.1. Typing CTRL-SHIFT-M will restore output to the screen.

ASCII — Takes a character as input and outputs the number that is the ASCII code of that character.

CHAR — Takes an integer as input and outputs the character whose ASCII code is that integer.

CLEARTEXT — Clears the text screen and homes the cursor.

CLEARINPUT — Clears the character input buffer of any typed-ahead characters.

CURSOR — Takes two inputs, column and row, and positions the cursor there. Columns are 0-39, rows are 0-23. *0,0* is upper left. See the CH and CV locations (page 74) to find out how to determine the cursor's current position.

PADDLE — Takes a number 0 through 3 as input, which specifies the paddle. Outputs a number 0-255 depending on the setting of the appropriate paddle dial. One example that can be used with either two paddles or a joystick is SETXY PADDLE 0 PADDLE 1.

PADDLEBUTTON — Take a number 0 through 2 as input and outputs TRUE or FALSE depending on whether the button on the corresponding paddle is pressed. One example of its use is IF PADDLEBUTTON 0 = "TRUE THEN CLEARSCREEN. On the Apple II, paddle 3 does not

have an associated paddle button.

PRINT — Variable number of inputs (default is 1). Prints the input on the screen. Lists are printed in "sentence" form, without the outermost level of brackets. The next PRINT will print on the next line of the screen. If there are multiple inputs, as in (PRINT 1 2 3), the inputs will be printed on one line, separated by spaces. Note that for multiple inputs, the entire statement must be enclosed in parentheses. If the input to PRINT is a procedure, it will not print the procedure, but will execute the procedure assuming the procedure will output something to print. (See also PRINTOUT) Abbreviated: PR.

PRINT1 — Like PRINT, but does not terminate output line with a return. With multiple inputs, does not print spaces between elements.

RC? — Outputs TRUE if a keyboard character is pending (i.e., if READCHARACTER would output immediately, without waiting for the user to press a key), otherwise outputs FALSE.

READCHARACTER — Outputs the least recent character in the character buffer, or if empty, waits for an input character. See CLEARINPUT, and section 7.4. See the explanation of the INSTANT program on the utilities disk for an example of its use. Abbreviated RC.

REQUEST — Waits for an input line to be typed by the user and terminated with RETURN. Outputs the line (as a list). Abbreviated: RQ.

3.9. Filing and Managing Workspace

CATALOG — Prints the names of files on the currently mounted disk.

DOS — Takes one input (word or list), and interprets it as commands to DOS. DOS [RENAME GMAE.LOGO, GAME.LOGO] will rename something saved with SAVE "GMAE. To "unlock" locked files (those which appear with an asterisk in the CATALOG listing) type, for example, DOS [UNLOCK ADDRESSES.LOGO]. The following DOS commands are available in this manner: DELETE, VERIFY, CATALOG, LOCK, UNLOCK, MON, NOMON, RENAME, BLOAD, BRUN, BSAVE. See the Apple DOS manual for information on the syntax of DOS commands. WARNING: This command is likely to be removed and replaced with individual primitives.

ERASEFILE — Removes from the disk a file saved with SAVE. Takes file name as input, which must begin with a " mark.

ERASEPICT — Removes a picture that has been stored on the disk using SAVEPICT. Takes picture name as input, which must begin with a " mark.

PRINTOUT — If given a procedure name as input, prints out the text of the procedure. If given no input, prints out the last procedure defined, edited or printed out. For convenience, the input is not evaluated; thus to see a procedure called CIRCLE, you would type PO CIRCLE and not PO "CIRCLE. Can also take auxiliary words: ALL, NAMES, PROCEDURES. To print out a list of procedures, the following procedure can be used.

```
TO POPROCS :PROCLIST
 IF :PROCLIST = [] STOP
 RUN LIST "PO FIRST :PROCLIST
 POPROCS BUTFIRST :PROCLIST
END
```

Abbreviated: PO. POTS is an abbreviation for PRINTOUT TITLES. (See also PRINT.)

READ — Reads a file from disk. Destroys any graphics display. Takes file name as input, which must begin with a " mark.

READPICT — Reads a picture that has been stored on disk and displays it on the graphics screen. Takes picture name as input, which must begin with a " mark.

SAVE — Saves the contents of the workspace on disk. See the Filing comments. Destroys any graphics display. Takes file name as input, which must begin with a " mark.

SAVEPICT — Save on disk the picture on the screen. Takes picture name as input, which must begin with a " mark.

3.10. Debugging

CONTINUE — Resumes execution after a PAUSE or CTRL-Z. Abbreviated: CO.

PAUSE — Stops execution and allows command lines to be evaluated in the current local environment. Equivalent to interrupt character CTRL-Z. Execution is resumed with CONTINUE, provided no errors have occurred.

NOTRACE — Turns off tracing.

TRACE — Takes no input. Causes Logo to pause before executing each procedure, and print the name of the procedure and its inputs. Typing any character other than CTRL-G or CTRL-Z will cause Logo to go on to the next line. Typing CTRL-G will cause Logo to abort to toplevel. CTRL-Z will pause, and space will continue execution. See CTRL-W, page 12.

3.11. Miscellaneous Commands

.ASPECT — Changes the vertical scale at which Logo graphics are drawn. Takes one numeric input and uses this to change the scale factor. The default value for the factor is 0.8. This command is included because not all TV monitors have the same amount of vertical deflection. Consequently, turtle programs that are supposed to draw squares and circles may instead appear to draw rectangles and ellipses. If so, the .ASPECT command can be used to attempt to compensate for the distortion. Note that changing the factor will change the limits for permissible y-coordinates.[4]

.BPT — Breaks out of Logo into the Apple monitor. (For use in Logo system debugging only.) Useful entry addresses are 1BF9, which is a "cold start" address to use after Logo has been started before; and the "warm start" address 1BFC, for attempting to recover after a system crash. After restarting Logo at the cold start address, all procedures are lost: it is just like typing GOODBYE. The warm start address leaves all variables and procedures intact.[5] The best way to return to Logo is to type CTRL-Y and RETURN.

.CALL — Calls a machine language subroutine in memory. The address of the subroutine is the first input; the second input is stored in a memory location for the routine to examine. This primitive allows users to provide their own special-purpose primitives and interface them to

---

[4] There is a problem with using values of .ASPECT that are too different from 0.8: Although lines will be drawn at the correct angle, the turtle pointer may not always appear to be pointing exactly along the line.

[5] In fact, all local variables still have the values they had at the time Logo was interrupted.

Logo. See section 6.2.

.CONTENTS — Returns a list of all words known to Logo. This includes names of variables, procedures, and words used in procedures. One use might be an editing program that, for each procedure defined, asks you whether you want to delete it. TEXT and THING? are useful primitives to use with the elements of this list. Caution: Use of this primitive interferes with garbage collection of "truly worthless atoms".[6] These are the no-longer-used words that Logo has in memory, usually as the result of typing errors. If you run short of memory, it might be because an old list from .CONTENTS is around somewhere keeping Logo from recovering the storage associated with no-longer-needed words.

.DEPOSIT — Takes two numeric inputs, an address and a value, and deposits a byte of data at a designated memory location. See section 6.1.

.EXAMINE — Takes one input. Outputs the value of the byte at the specified address. See section 6.1.

.GCOLL — Forces a garbage collection.

.NODES — Outputs the number of currently free nodes. To obtain a true count of free memory, type .GCOLL before typing .NODES.

; — Causes the rest of the line not to be evaluated. Useful for including comments in procedures and procedure titles.

---

[6] Actually, no version of Logo for any microcomputer yet implements GCTWA, so the storage is never recovered. Terrapin, however, is working on implementing this feature.

Chapter 4
The Utilities Disk

The Utilities Disk is the diskette containing demonstration and utility programs from MIT and Terrapin. One Utilities Disk is included with each Logo package.

This chapter lists the files on the Utilities Disk, and briefly describes how to run the demonstration programs included. Further explanation and examples are in the Tutorial.

Using the Utilities Disk

Start Logo as described in section 1.3, and in the Tutorial. You should see the ? prompt with the cursor flashing next to it.

Insert the Utilities disk into the drive, and type CATALOG, followed by return. This will give you the first part of a listing of the programs on the disk. You must press the space bar to see the remainder. If you type CATALOG while Logo turtle graphics are on the screen, you will see only four lines of text at a time. You can type CTRL-T to see the full text screen, and CTRL-S to return to the graphics screen.

To read a program from the disk, type READ " immediately followed by the name of the program. Do not use a closing quotation mark, and do not type ".LOGO" at the end of the name.

Logo will then print out the name of each procedure in the file as it is read in and defined. When it is finished, the ? prompt will reappear.

Some programs are *self-starting*; that is, they begin executing by themselves once you read the file. Most programs are not, and require you to type the name of the initial procedure. The initial procedure for most of the demonstration programs on the Utilities disk is the same as the name of the file. For utility programs which are intended to be initialized and used later, the procedure is usually called "SETUP".

If Logo prints the ? prompt when it finishes reading the file, you must start the program yourself. The descriptions of the programs below explain how to start each one.

Here is a sample session showing how to use a demonstration program. What you should type is shown in plain type, and what Logo prints is shown in italics.

```
?READ "ROCKET
SH DEFINED
GLOW DEFINED
TRAVEL DEFINED
CIRC DEFINED
STARS DEFINED
SHOW DEFINED
ROCKET DEFINED
?ROCKET
```

4.1. Program Descriptions

These Logo files are for various utility programs written in Logo by MIT and Terrapin. They contain information used by other files on the Utilities Disk.

ASSEMBLER	The Logo assembler procedures. See chapter 6.
AMODES	The file of names describing the 6502 addressing modes.
ADDRESSES	The file of names describing addresses in the Logo interpreter for the assembler.
OPCODES	The file of names describing the 6502 mnemonics for the assembler.
SHAPE.EDIT	The Logo shape editor, described in section 5.
FID	A file utility program. It makes deleting and renaming files convenient. It starts itself when you read it in. To restart it, type FID.

These files are utility programs provided by Terrapin and MIT.

PSAVE

The Logo SAVE primitive saves all procedures and names currently defined. Sometimes this behavior is inconvenient. The PSAVE procedure allows you to save an arbitrary list of procedures in a file. The first input to PSAVE is the filename, and the second is the list of procedures to save. As one of the procedure names, you can use the word NAMES to indicate that Logo variables should be saved. (See also the HPL procedure on page 17 which prints out a list of procedures to a printer.)

To load PSAVE, type

```
READ "PSAVE
```

To save procedures named CIRCLE, SQUARE, and TRIANGLE in the file DESIGN, type

```
PSAVE "DESIGN [CIRCLE SQUARE TRIANGLE]
```

TEACH

The Logo editor allows tremendous flexibility in defining procedures and editing, but at the cost of being complex to new users. The TEACH procedure allows the user to define procedures without entering the editor. It has the additional advantages of prompting the user for information and not clearing the turtle-graphics screen.

To use the TEACH procedure, type

```
READ "TEACH
```

Type TEACH to teach Logo a new word and END to stop it. Here is an example, with the parts that Logo prints out in italics:

```
?TEACH
NAME OF PROCEDURE>COUNT
INPUTS (IF ANY)?:N
<IF :N = 0 STOP
<PRINT :N
<COUNT :N - 1
<END
COUNT DEFINED
?
```

CURSOR

These procedures are for controlling character output. The CURSOR primitive is the only one provided for controlling the location of the text cursor on the screen. The following procedures are for performing operations not directly supported in Logo:

CURSOR.HV	Outputs a list. The first element is the cursor's horizontal position, and second, its vertical position.
CURSOR.H	Outputs just the horizontal position.
CURSOR.V	Outputs the vertical position.
CURSORPOS	Takes one input (a list) and sets the cursor to the corresponding position on the screen.
FLASHING	Causes all characters printed out after execution of this command to be in flashing characters.
INVERSE	Makes characters be in inverse video (black-on-white).
NORMAL	Restores the normal mode of white-on-black.

These procedures do not affect characters already on the screen, only those printed out afterwards. You can start a blank text file by typing TO followed by RETURN. Then type whatever text you want. When you are done, type CTRL-G and then save the file using SAVETEXT. To read a file back in, use READTEXT and then type ED followed by RETURN. If you type TO it will clear the edit buffer and you will have to use READTEXT again.

TEXTEDIT

Procedures for using the Logo editor as a text editor. These procedures allow you to use the Logo editor to read and save files of English text.

SAVETEXT :FILE Causes the contents of the editor to be stored on disk in the Logo file with the specified name. Example:

```
SAVETEXT "LETTER
```

READTEXT :FILE Complementary to SAVETEXT. Reads a Logo file into the editor.

SHOWFILE :FILE Prints the contents of the file on the screen.

PRINTFILE :FILE Prints the contents of the file on the printer. The printer is assumed to be in the slot specified by the variable PRINTER. If you don't set it yourself (by typing MAKE "PRINTER 7, for example) it will remain slot 1.

PRINTTEXT Prints the contents of the editor on the printer.

DPRINT

This file contains procedures for printing arbitrary text into disk files. These procedures are described in section 7.3.

OPEN :FILE Takes a file name as input. The input to DPRINT (see below) will be printed into the Logo file with this name.

CLOSE Closes the open file. All output will be written to the file.

DPRINT :ITEM Will cause the item to be printed into the file.

OPEN.FOR.APPEND :FILE
Used instead of OPEN. Will cause everything printed with DPRINT to be appended to the existing file rather than writing over it.

The files created by these procedures can be printed and read into the editor by the procedures in the file TEXTEDIT. If you use READ on such a file, Logo will attempt to execute the text in the file as Logo commands.

ARCS

This file has procedures for drawing arcs and circles of specific radii. For your convenience, the procedures for drawing arcs of specific radii are reproduced in this file.

RARC :RADIUS :DEGREES, LARC :RADIUS :DEGREES
These procedures each take two inputs and draw an arc of the specified radius and covering the number of

degrees indicated.

RCIRCLE :RADIUS, LCIRCLE :RADIUS
These procedures each take one input, the radius, and draw a circle of that radius.

These procedures are also discussed in the book Logo for the Apple II.

TCL

Turtle Control Language.

This file is for owners of the Terrapin Turtle (TM). It contains the following procedures for using the Turtle:

SETUP — Type SETUP after loading the file. It initializes the Turtle Interface and various system parameters.

HELP — This procedure describes the procedures available for using the Turtle.

TFD :DIST, TBK :DIST, TLT :ANGLE, TRT :ANGLE
These procedures correspond to FORWARD, BACK, LEFT, and RIGHT, but control the floor turtle instead of the screen turtle.

EYESON, EYESOFF
These procedures turn off and on the LEDs attached to the Turtle.

HORNLO — Makes the Turtle sound its horn at a low pitch.

HORNHI — Similar, but with a higher pitch.

HORNOFF — Turns the horn off.

TPU — Raises the Turtle's pen so it won't draw. This is its initial state.

TPD — Lowers the pen.

FTOUCH? — Outputs TRUE if the Turtle's dome is touching anything on the front; otherwise it outputs FALSE.

LTOUCH? — Left touch. RTOUCH?

Right touch. BTOUCH?
Back touch.

FTOUCHONLY? Whereas FTOUCH? will output TRUE if the Turtle is touching something in front, or in front and on the left, or in front and on the right, FTOUCHONLY? will output TRUE if the Turtle is touching something only in front. Use this procedure as a model for similar procedures for other directions.

ANYTOUCH? Outputs TRUE if the Turtle is touching anything.

NOTOUCH? Outputs TRUE if the Turtle is touching nothing.

TOUCH Outputs the state of the touch sensors as a number. This routine is used internally in the above touch-testing routines. The Logo variables :FBIT, :LBIT, :BBIT, and :RBIT are names for the numbers this number is composed of. (See the Terrapin - Apple Interface manual for clarification.)

.TCMD :COMMAND :ARGUMENT
.TCMD is the lowest-level procedure for controlling the Turtle. It sends the command and argument to the Turtle. You can use it to make the Turtle do things that the above procedures don't support. See the Terrapin - Apple Interface manual for the details of controlling the Turtle via this command.

4.1.1. Demonstration Programs

These files are various Logo demonstration programs.

Rocket

The Logo files ROCKET, ROCKET.AUX and the file ROCKET.SHAPES are an illustration of a typical use of user-defined turtle shapes. READ "ROCKET and type ROCKET. See page 56 for a description of the shape editor.

Animal

This program attempts to augment its knowledge about the animal kingdom by playing a game in which it tries to guess the animal you are thinking of. It asks various questions, such as "Does it have wings?" You answer with "Yes" or "No."

Type READ "ANIMAL and ANIMAL. When you are finished playing, you can type SAVE "ANIMAL, and the next time you play, it will know the animals you taught it. The animal game on the Utilities Disk already knows several animals. To make it start out fresh, run the procedure INITIALIZE.KNOWLEDGE.

Animal.Inspector

The procedures in this file are for examining the ANIMAL knowledge base. The ANIMAL game described in the book Logo for the Apple II keeps its information about animals in the variable KNOWLEDGE. This file contains the procedure INSPECT.KNOWLEDGE, which prints out the ANIMAL program's "knowledge" about animals in an easily readable form. This procedure is intended as a learning aid to be used with the discussion of the ANIMAL program mentioned above.

In its use of recursion, it is similar to tree-drawing programs, since it actually follows the tree of the Animal program's knowledge as it prints it out. Look at the procedures in this file as an example of recursive programming.

Instant

This collection of procedures makes the Logo system easy to use even for very young children. After you READ "INSTANT and type INSTANT, you can use single-character commands to manipulate the turtle and define procedures. Each character is acted upon immediately. Typing F, for example, makes the turtle move forward a small amount, leaving a trail. R makes it turn to the right. Repeating a sequence of F's and R's will draw a square.

The INSTANT system allows you to store the commands you have typed as a procedure. When you type N, it will define a procedure to draw the picture currently on the screen. If you draw a square using R and F, and

name the result SQUARE (using the N command), the INSTANT system will define a procedure SQUARE with calls to FORWARD and RIGHT in it. Typing P to INSTANT will have it ask you the name of a procedure to run (picture to show), and run that procedure. Here is a table of INSTANT commands:

?	Help.
D	Clears the screen.
F	Go forward.
L	Turn left.
N	Names a new picture.
P	Asks for the name of a picture to show.
R	Turn right.
U	Undo the last command.

The INSTANT program is an example of how easy it is to create "languages" with simple Logo programs. It also serves as an example of Logo programming style, and of the use of RUN and DEFINE. You can easily modify INSTANT to provide more complex commands.

Dynatrack

The DYNATRACK program implements something called a "dynamic turtle" -- one which moves around with time. This particular program simulates a ride around a frictionless racetrack. Type DYNATRACK to start. K causes the turtle to accelerate in the direction it was pointing. L and R turn the turtle to the left and to the right.

FID

FID is a file utility program written in Logo. It allows you to catalog the disk, rename files, and delete files, all with single-keystroke commands. Each file command asks for the name of a file, and appends to it the current "file extension." The "." command in FID allows you to change the file extension. Useful file extensions for Logo are "LOGO," "PICT," "SHAPES," and "BIN." In the event of a disk-file error, restart the program

by running the procedure FID.

Music

There are three music files: MUSIC.LOGO, containing Logo procedures to play music; MUSIC.SRC.LOGO, containing the assembly language program for playing notes; and MUSIC.BIN, the file which the music demo procedures BLOAD. To run the music programs, READ "MUSIC and type SETUP. For a short demonstration, run the FRERE procedure. See page 72 for documentation of the music system. The Tutorial also has an explanation and examples of its use.

Inspi

The picture file INSPI was generated by running the following procedure four times with the turtle pointing at different angles, and with different pen colors:

```
TO INSPI :DISTANCE :ANGLE :INCREMENT
 FD :DISTANCE
 RT :ANGLE
 INSPI :DISTANCE :ANGLE + :INCREMENT  :INCREMENT
END
```

You can display the picture by typing READPICT "INSPI.

TET

The procedures in this file are an example of how simple Logo programs using recursion can draw complex, interesting figures. The TET procedure takes two inputs, SIZE and LEVELS. Try TET 100 1 to see what the first "level" of the drawing looks like. Then try it with 2, 3, and higher levels.

Chapter 5
Changing the Turtle Shape

As an example of the kind of facility that can be added to Logo through judicious use of .EXAMINE and .DEPOSIT, we consider the problem of doing "animation" in Logo. Logo for the Apple II is not designed for producing animation effects.[1] The best screen motion that you can obtain is by moving the turtle. If you want to make a circle move across the screen, it will be very slow to repeatedly draw and erase the circle, moving the position of the circle little by little. One thing you can do, however, is to change the shape of the turtle itself, so that the turtle looks like a circle. Then you can make a circle move across the screen by simply moving the turtle.

The Logo turtle is drawn using the Apple "shape" mechanism, that allows specification of shapes by tables of two- and three-bit vectors as described in the Applesoft Programming Reference Manual. You can design your own shapes for Logo to move around on the screen in place of the turtle. To set up your own shape table, deposit the location of the first element of the table in the address USHAPE. (See section 6.5 for explanation of Logo addresses.) The size of the turtle or of the created shape can be changed with the one-byte size code contained in address SSIZE. The default value, 1, is best for the regular turtle; however, values of 2 or greater often make user-defined shapes more visible. Unlike the general Apple shape mechanism, the Logo interface to shapes allows user-defined shapes to be displayed only at at 0, 90, 180 and 270 degree headings. The heading at which the shape is displayed is determined by the quadrant in which the "turtle" is facing and can be changed by turning the "turtle" with the usual LEFT and RIGHT commands.

The format of shape tables is as described in the Applesoft Reference Manual, except that the header information in the shape table should be omitted. Begin each shape table directly with the vectors. Terminate it

---

[1] This is in contrast to the Logo implementation for the Texas Instruments 99/4, which takes advantage of special animation hardware provided by the computer.

normally.

You can construct a shape table by hand and use .DEPOSIT to store it in the Logo area reserved for user code, and then set USHAPE and SSIZE. Note that you can make more than one user-defined shape and switch between them by changing USHAPE.

The Logo Shape editor

Constructing shape tables is a tedious process. One of the programs contained on the Logo disk is a "shape editor." This is a Logo program that enables you to design a shape by drawing it directly on the screen. It then automatically assembles the shape into a shape table. The shape editor was written by Henry Minsky. Note that the shape editor is itself a collection of Logo procedures that work by using .EXAMINE and .DEPOSIT according to the scheme outlined above. You can read in the procedures and use them as a guide to writing similar functions.

To use the shape editor, read in the file SHAPE.EDIT and type SETUP. The file contains a real-time shape editor and functions for changing the currently displayed turtle shape and its size. To begin designing your own shape, give the MAKESHAPE command. This takes one input that is to be the name of the shape you will design, for example,

```
MAKESHAPE "BOX
```

Typing another MAKESHAPE command will cause a new shape to be defined. You cannot edit previously-defined shapes. If you wish to erase all shapes and start over, type SETUP again. The following commands (similar to the editor commands) are available for constructing shapes.

U	Penup
D	Pendown
CTRL-P	Move up (and draw a vertical line if the pen is down).
CTRL-N	Move down (and draw a vertical line if the pen is down).
arrow keys	Move in the direction of the arrow (and draw a horizontal line if the pen is down).

CTRL-C	Exit the shape editor and define the shape.
CTRL-G	Exit without permanently defining the shape. (You can set the turtle to the shape as defined so far, but the next time you define a new shape, this one will be lost.) Use this command to abort definition of a shape if you wish to start over.
ESC	Delete the previous few commands.[2]
1...9	Typing a number causes the size of the shape to change. Typing 3 is equivalent to executing SIZE 3. It is helpful to switch between the size you want and a larger size, which is easier to see, while designing a shape.

Once you've defined a turtle shape (BOX, in this example), you can make the turtle assume that shape by typing SETSHAPE :BOX.

You can change the size in which shapes are shown by using the SIZE procedure. SIZE 1 is the default.

The SETSHAPE takes one input and changes the turtle to have that shape. It first hides the turtle, and then shows it. If you want to restore the turtle to its original triangular shape, type SETSHAPE 0.

The internal procedure which SETSHAPE calls is .SHAPE. It, too, takes the shape as input, but it doesn't hide the turtle first. This is useful because Logo draws and erases the turtle by drawing the turtle shape in "reverse" mode(i.e., pencolor 6). This implies that if you set the turtle to some shape, then set the turtle to the a shape with no lines it in, then hide the turtle and move it, the original turtle image will remain on the screen, because "hiding" the empty shape effectively erases nothing. .SHAPE 1 will give the turtle the "null" shape. The following procedures use this method. The first will stamp a shape that the user specifies. The second procedure determines the current shape and stamps that. (It uses USHAPE which is an address that can be read from the file ADDRESSES

---

[2]What actually happens is that the previous byte in the shape table is deleted, so that the previous one, two or three segments are deleted.

on the utilities disk.) The last procedure stamps the shape at random places on the screen.

```
TO STAMP :SHAPE
 .SHAPE 1
 HIDETURTLE
 .SHAPE :SHAPE
 SHOWTURTLE
END

TO SS
 STAMP (.EXAMINE :USHAPE)+256*.EXAMINE :USHAPE+1
END

TO STAMPRANDOM
 SS
 SETXY (120 - RANDOM 240)(110 - RANDOM 220)
 STAMPRANDOM
END
```

Saving Shapes

To save on disk all the shapes you defined, use the SAVESHAPES procedure. It takes the name of the file as input. Be sure to use a quote, just as you would with SAVE.

The following information is very important: Start in a fresh Logo workspace with no procedures defined. Then read in the shape editor and define a few shapes. SAVESHAPES "PARTS will create two files. The first, PARTS.SHAPES, will contain the shape table (the actual appearance of the shape); the second, PARTS.AUX.LOGO, will have the names of the shapes and the following procedures: SETSHAPE, .SHAPE, SIZE, INITSHAPES, and any procedures you have defined. Unless you are writing low-level procedures for manipulating shapes themselves, you probably don't want to include any extra procedures in this file.

The ".AUX" file contains a procedure called INITSHAPES; it automatically loads and sets up the defined shapes. For a procedure to use shapes you saved in a file called BLOCKS, your procedure should include the following:

```
READ "BLOCKS.AUX
INITSHAPES
```

For easier maintenance of your program, you should keep the program that actually makes use of the shapes in a file separate from the shape files. That way, you can make changes to the shapes and keep them in the shape files, and make changes to the procedures that use shapes and keep them in a different file. You don't have to worry about accidently erasing the procedures or names that the shape system uses. To accomplish this, you should not define procedures to use shapes while you're still using the shape editor. If you accidently do, you should edit the resulting ".AUX" file and separate the procedures into two different files.

A Sample Session

```
?GOODBYE
?READ "SHAPE.EDIT
?MAKESHAPE "BLOCK
Define a shape here
?MAKESHAPE "TIRE
Define another shape here
?SAVESHAPES "BLOCKS
```

At this point, Logo will ask you to place your files disk in the disk drive. This disk should be the one you want to use to store the shapes and the program to use them. After Logo saves the shapes, it will ask you to put the disk containing the shape editor back into the drive. You should then place the Utilities Disk (or a copy of it) into the drive and press return. Logo will pause for a while as it reads the shape editor back into memory.

There should now be two new files on the files disk that you used: BLOCKS.AUX.LOGO and BLOCKS.SHAPES. Type GOODBYE to erase everything and start over. To regain the shape you created, type

```
READ "BLOCKS.AUX
INITSHAPES
SETSHAPE :BLOCK
```

You now have two options: you may write procedures to use these shapes. and then save everything. That way, all you have to do to read in the shapes is execute INITSHAPES; however, if you change the shapes in the file BLOCKS, you must erase all the NAMES and read in the new

BLOCKS.AUX file. The alternative is to start with a fresh workspace by typing GOODBYE, write the procedures that use the shapes, and include the READ and INITSHAPES commands in the procedures so the shapes are read in when the program starts running. When testing the program, it will bring the shapes into the procedures workspace. Therefore, when you are satisfied, use ERASE to erase the following; NAMES, SETSHAPE, .SHAPE, and INITSHAPES. Then save the file. The following paragraph briefly describes a program that uses this technique of separating the files.

The Utilities Disk Example: Rocket

On the Utilities Disk, there is a demonstration program called ROCKET. Type READ "ROCKET and execute the procedure ROCKET. The ROCKET procedure READs the ROCKET.AUX file (described above), and calls the procedure INITSHAPES. The INITSHAPES procedure automatically sets up the shapes.

It will make it easier to use the shape procedures if you follow the conventions outlined here. Type PO ROCKET to see how it works. To run it again without loading the file, type SHOW.

Chapter 6
Assembly Language Interfaces to Logo

The Logo system for the Apple has been designed to be both powerful and easy to use. Writing and executing programs in the Logo language using the primitives listed in the previous chapter should be sufficient for most purposes. However, there are situations in which it is desirable to extend the capabilities of the system by getting direct access to machine language.

Warning: This chapter will only be useful/intelligible to people who are familiar with assembly language programming on the Apple.

The Logo system has various "hooks" built in to it that enable users to directly access memory locations in the Apple and to interface assembly language routines to Logo programs. The Logo Utilities Diskette includes a 6502 machine language assembler that aids in doing this. Another hook built in to Logo allows you to create simple animation effects by supplying a new shape to be displayed in place of the Logo turtle. Another hook allows you to modify the behavior of the Logo editor so that it can be used as a regular text editor rather than as a procedure editor and to access disk files in non-standard ways.

6.1. .EXAMINE and .DEPOSIT

These two commands are essentially the usual Apple PEEK and POKE routines.[1] .EXAMINE takes an address as an input and returns (as a number) the byte stored in that address. .DEPOSIT takes two inputs, an address and a numeric value, and deposits the value in the byte specified by the address. These commands are useful for communicating with special-purpose I/O devices, especially in cases where the facility supplied by OUTDEV is insufficient. Needless to say, .DEPOSITing into

---

[1] One difference is that the addresses should always be specified as positive integers. Apple PEEK and POKE require addresses above 32K to be given as negative numbers.

arbitrary memory locations can cause Logo to crash or do other unfriendly things. Note that the addresses used with these commands are ordinary Logo numbers, which are expressed in base 10, even though it is customary to think of Apple addresses as written in hexadecimal notation. For many purposes it would be useful to write a conversion routine that converts from hexadecimal to base ten.[2] That way, you could type, for example

```
.EXAMINE HEX "9E
```

rather than

```
.EXAMINE 158
```

When Logo is running, monitor ROM locations are not available. Instead, these locations correspond to parts of the Logo system stored on the 16K memory card. The actual contents of the ROM are shadowed by this memory. Calling .EXAMINE (or .DEPOSIT) with an address which corresponds to the "Language Card" will cause unpredictable effects. The 16K memory card occupies locations $C080-$C08F (49280-49295).

6.2. Writing Your Own Machine-Language Routines

You can interface your own machine-language routines to Logo by using the .CALL primitive. .CALL takes two inputs: the first is the address of the routine, and the second is an integer input that the routine may examine. The routine may output an integer or output nothing. The .CALL primitive always requires two inputs, regardless of whether the user routine chooses to examine the second one.

.CALL transfers control to the address specified by its first input. Naturally, before doing this, you should assemble an appropriate routine and store it at the address. The available memory for user machine code begins at $99A0 and extends to $9AA0. You can do the assembly by hand and store the routine using .DEPOSIT, but you will find it much more

---

[2] Throughout this chapter, we use the convention of specifying hexadecimal numbers as prefixed by a dollar sign, e.g., $9E is 158 decimal.

convenient to make use of the Logo assembler described in section 6.3.

When your routine begins executing, the page zero locations NARG1 and NARG1 + 1 contains the first input to .CALL -- which is just the address of the routine itself. NARG1 + 2 and NARG1 + 3 are guaranteed to contain zero at the time the routine is called. The routine may use locations NARG1 through ANSN4 + 3 as temporary storage locations, without worrying about restoring them before returning. These storage locations are volatile; that is, Logo may change these locations between successive calls to your routine. Locations USERPZ through $FF are not used by Logo and so can be used by your routines as non-volatile, page-zero storage.

NARG2 through NARG2 + 3 contain the second input to .CALL, stored as a four-byte fixnum in two's complement form. Thus .CALL ($ "99A0) 3 would result in the following values in memory:

```
NARG2   NARG2+1   NARG2+2   NARG2+3
3       0         0         0

NARG1   NARG1+1   NARG1+2   NARG1+3
$A0     $99       0         0
```

Substituting -1 for 3 would make NARG2 through NARG2 + 3 contain $FF.

To output an integer, store the integer to be returned (using the above format) in the four locations with NARG2 through NARG2 + 3, and jump to location OTPFX2. If the number is stored in some other set of 4 consecutive page-zero variables (such as NARG1), load Y with the address and jump to OTPFIX.

To output the Logo word "TRUE, jump to OTPTRU; similarly, jumping to OTPFLS will cause your routine to output "FALSE. To output no value, simply end the routine with an RTS instruction.

Here is an example which reads the state of the cassette port (by addressing the cassette input location $C060) and returns "TRUE or "FALSE, depending on whether there is sound available. The code here is written in standard 6502 assembler format. To use it you will have to assemble it by hand and deposit the instructions in memory (but see section 6.3 below).

```
        ORG $99A0
CIN     EQU $C060
OTPTRU  EQU <see Addresses.Logo>
OTPFLS  EQU <see Addresses.Logo>
LISTEN: LDA CIN
        BMI ON          ;See Apple II Reference Manual,  p. 78.
        JMP OTPFLS
ON:     JMP OTPTRU
        END
```

Now you can set the Logo variable LISTEN to the address of the label LISTEN and execute this new "primitive" by typing

```
.CALL :LISTEN 0
```

(Note that an input is needed, even though it is ignored.) This will work just like a normal primitive or procedure --

```
PRINT .CALL :LISTEN 0
```

will print TRUE or FALSE.

When a machine language routine has determined some error condition that would make it inappropriate to return to the Logo procedure that called it, it can and jump to PPTTP, which effectively executes the Logo TOPLEVEL primitive.

6.3. The Logo Assembler

The Logo assembler is a 6502 assembler that is written in the Logo language. This program was designed and written by Leigh Klotz. The assembler is stored on the Logo utilities disk in the file ASSEMBLER.[3] To use the assembler, simply read this file into Logo as you would any normal Logo file and then run a procedure called SETUP:

```
READ "ASSEMBLER
SETUP
```

To assemble a routine, you write the routine in the format of a Logo procedure, using the Logo editor. For example, the tape cassette

---

[3]The ASSEMBLER program in turn reads data stored on the Logo disk in auxiliary files AMODES and OPCODES.

example on page 62 would be written as the procedure:

```
TO CASSETTE.CODE
 [MAKE "CIN $ "C060]
 [MAKE "OTPFLS <see Addresses.Logo>]
 [MAKE "OTPTRU <see Addresses.Logo>]
 LISTEN: LDA CIN
         BMI ON
         JMP OTPFLS
ON:      JMP OTPTRU
END
```

Notice that there are differences in syntax between the input accepted by the Logo assembler and the standard 6502 assembler. The syntax of code for the assembler is explained in section 6.3.2.

Once you have defined the procedure you now assemble it by typing

```
ASSEMBLE "CASSETTE.CODE
```

ASSEMBLE will now assemble the instructions and place them in the default location ($99A0). Also, any labels in the code (such as LISTEN, above) will now be defined as Logo symbols. So now you can call the routine by

```
.CALL :LISTEN 0
```

6.3.1. Using the Assembler to Write I/O Routines

While it is possible to use the .EXAMINE and .DEPOSIT primitives to operate most peripheral devices, machine language routines are required for others. If the peripheral device is one which has a built in "driver," then you can use the OUTDEV primitive. OUTDEV takes as input a slot number 1 through 7 as an input and directs the Logo character input or output routines to the device at the specified slot.

Some devices, however, may require special routines to handle input and output. If you specify OUTDEV with an input greater than 8, the input will be interpreted as the address of a routine in memory that should be called in place of Logo's regular character input or output routine.

Many peripherals use a technique called "handshaking" to assure that the computer does not try to send data to them (or read data from them) too fast. The following program will interface Logo to such a device. We

assume that STATUS is the memory-mapped I/O address on the peripheral card indicating the status of the device. In this case, bit 7 is high if the device is ready to receive a character. DATA is the address where bytes to be sent should be stored.

Once you have assembled this routine, you may access the peripheral by executing OUTDEV :TYOWAIT.

```
TO CODE
 [MAKE "STATUS <address>]
 [MAKE "DATA <address>]
 TYOWAIT: LDX STATUS
          BPL TYOWAIT
          STA DATA
          RTS
END
```

A character output routine like this, which is meant to be called via OUTDEV, should expect that the A register will contain the byte to be output.

Here is another example output routine. This one causes all ! characters to be printed as spaces:

```
TO IOCODE
XCLOUT:   CMP # "!
          BNE OUTCHAR
          LDA # 32
OUTCHAR:  JMP COUT
END
```

6.3.2. Syntax of Input to the Assembler

In order to take advantage of some aspects of the Logo language, the Logo assembler uses a format slightly different from most assemblers. Each assembly-language program is stored as a Logo procedure, although this procedure cannot be executed directly. The following paragraphs concisely describe the Logo assembler format; a study of the examples provided will better explain how to write assembly language programs to interface with Logo.

Labels within the program are indicated by a postfixed colon. References to page-zero memory locations that are not indirect-indexed (LDA (FOO ,X)) or indexed-indirect (LDA (FOO) ,Y) must have an

exclamation point before the label or expression that is on page zero. (If you forget the exclamation point, the instruction will be coded as absolute references, and will occupy one more byte.) There must be a space following every ! (indicating page-zero reference) or # (indicating immediate mode), and after every label or reference to a label. The operand of an instruction may be a word (a reference to a label), a number, a list, or a single-letter word beginning with a quote. If the latter case, the operand is the ASCII value of the letter.

Anything inside a list is evaluated as a regular Logo expression. If the list is the first thing on the line, it is not allowed to output a value, and is evaluated for "side-effect" (label assignment) only. If it is an operand (follows the name of an instruction), it is expected to output something. Thus, arithmetic expressions such as :FOO + 3, where FOO is a label or regular Logo symbol, may be used provided they are enclosed in square brackets. Of course, references to the values of labels inside square brackets must have dots (:) before them, and spaces have their normal significance. All labels are Logo variables. DOT is a Logo variable whose value is the current location being assembled.

The HI8 and LO8 procedures, which return respectively the high and low eight bits of a number, are also useful inside lists. Use them like this:

```
LDA # [LO8 :SOURCE]
STA ! DEST
LDA # [HI8 :SOURCE]
STA ! [:DEST+1]
```

The $ procedure takes as input a word that is a hexadecimal number and outputs the number that it represents. Thus, hex numbers may be included in programs by placing a call to the $ inside a list. Use the MAKE primitive to assign values to labels.

If you use octal or binary numbers a lot, you might want to change the value of the Logo word $BASE. This is the base used by the $ procedure. Changing it to 2 gives you binary, and so on. You can do this within the source for an assembly language program with [MAKE "$BASE 2].

You can assemble arbitrary bytes into code by placing the number on the line with nothing (except perhaps a label) preceding it.

Here is a simple program in normal assembler syntax:

```
ORG     EQU $99A0
NARG2   EQU $9E
BELL    EQU $1C40
PASS:   LDA NARG2
        CMP #$04
        BEQ YES
        RTS
YES:    JMP BELL
END
```

And in the Logo assembler syntax:

```
TO CODE
PASS: LDA ! NARG2        ;! means page zero. Note space after !.
      CMP # [$ "04]
      BEQ YES
      RTS
YES:  JMP BELL
END
```

To assemble this program, load in the assembler and type SETUP and READ "ADDRESSES. The following will assemble the above program, with a default origin of $99A0.[4]

```
ASSEMBLE "CODE
```

This will generate a listing file on the screen and deposit the code in memory. The labels are available as Logo symbols for use with .CALL, .DEPOSIT, and .EXAMINE. To invoke the above routine, type

```
.CALL :PASS 4
```

to beep the bell, and .CALL :PASS <anything but 4> to do nothing. This is sort of a secret "password" program.

If you try to assemble long programs, you may run out of memory. One way to get more memory is to load in only those instructions that your program uses. In a fresh Logo, read in the OPCODES file from the utilities disk and erase the instructions (using ERNAME "BIT, for example) that you don't plan to use. Then, rewrite this as the new OPCODES file.[5]

---

[4] To assemble at some other start address, assign the value to the Logo variable ORG.

[5] Of course, you should not do this on the original Logo disk. Save copies of the original assembler files on an ordinary Logo file disk and run the assembler using these copies.

6.3.3. Saving Assembled Routines on Disk

With the DOS primitive, you can save the actual machine code that the assembler generates. The following will save your assembled routines in a file called ROUTINES.

```
DOS [BSAVE ROUTINES.BIN,A$99A0,L$100]
```

To load the routines into Logo, type

```
DOS [BLOAD ROUTINES.BIN]
```

The BIN is short for BINARY, and might help you remember that the file is a saved machine-language file. Keep in mind that in addition to saving the actual machine code, you should save the Logo variables that define the addresses used by .CALL. One way to do this is to type EDIT NAMES, then exit the editor with CTRL-G and execute ERASE ALL. Re-enter the editor and edit the definitions to include only the ones you still want. Then exit the editor with CTRL-C. Save the file by typing SAVE "ROUTINES. Then, to reload your routine, type READ "ROUTINES and DOS [BLOAD ROUTINES].

6.4. Example: Generating music

This section presents an example of an assembly language extension to Logo. Although this version of Logo has no primitives for playing music, you can use the .CALL feature to interface a machine-language routine to produce pitches with the Apple II speaker. The speaker produces a narrow pulse each time the location to which it is mapped, $C030 (49200), is referenced. Try repeatedly reading this location from Logo using .EXAMINE.[6]

In order to play pitches, a program has to examine this location many times each second. The number of clicks produced per second is called the *frequency*. To make the pitches sound equally spaced, the ratio of

---

[6]Due to the way the speaker is interfaced, depositing in the location has no effect. Additionally, the speaker generate clicks only on every other reference. This brings the pitch down one whole octave, but does not affect the intervals of pitches played.

successive frequencies must be constant; that is, the frequencies must be in geometric progression. In Western music, which has twelve pitches to the octave, this constant must be such that the frequency doubles after twelve pitches; thus, the ratio of successive pitches in the scale is the twelfth root of 2, or approximately 1.05946.

Closely related to the concept of frequency is that of *period*. The period of a pitch is simply the reciprocal of its frequency. Given the period, it is possible to play the corresponding pitch by repeatedly generating a narrow pulse (click), and then waiting for the period to expire. The program which does that will have to be written in machine language, since it must run *very* quickly.

To make Logo play music, we need to write some procedures. Let's say there should be a procedure called "PLAY," and that it should take two inputs: a list of pitches and a corresponding list of durations. The pitches should be numbers specifying the number of chromatic steps above or below a center pitch. The durations should be lengths of time for the note to sound, with 1 being the shortest, 2 being twice as long, and so on.

```
TO PLAY :PITCHES :DURS
  IF :PITCHES=[] STOP
  PLAY.NOTE (FIRST :PITCHES) (FIRST :DURS)
  PLAY (BF :PITCHES) (BF :DURS)
END
```

Even though we're not exactly sure how notes will be played, we can assume that PLAY.NOTE actually plays a note (given the pitch number and duration) because that's what we're going to write it to do.

Since the notes are played by a machine-language program that requires the period of the pitch, we must find some way of associating the periods of various pitches with their representation in the PLAY procedure. A table would be one good way of doing this. For each note, there is an entry in the table that contains the period. We'll construct our table as Logo words, and have the periods as the things associated with the names. We'll choose some arbitrary name for this table, and then have the individual notes be represented by words that begin with the table name and have the number of the pitch at the end of the name. So, if we call the table " # " the period for note number 3 is in the Logo variable called " # 3." We'll assign a special value to mean rest, and use the Logo

variable #R to store this value, so that "R" will cause a rest in the PLAY procedure.

Additionally, it would be useful to be able to specify notes above or below the center octave in some convenient notation. We have chosen postfixed plus and minus signs to indicate different octaves. In inputs to the PLAY procedure 4 means the fourth pitch above the center tone. 4+ means the same pitch an octave above it, and 4- the pitch one octave below. (You can worry about an appropriate notation for extending the range to more than these three octaves if you wish.)

The following MAKE.PITCHES procedure associates each pitch with the proper period. It takes as input the number of the highest octave and the period of the highest pitch in that octave.

```
TO MAKE.PITCHES :PERIOD
  MAKE.OCTAVE 11 "+ :PERIOD
  MAKE.OCTAVE 11 "  :PERIOD * 2
  MAKE.OCTAVE 11 "- :PERIOD * 4
  MAKE "#R 16384
END

TO MAKE.OCTAVE :PITCH :OCTIND :PERIOD
  IF :PITCH=0 STOP
  MAKE (WORD "# :PITCH :OCTIND) :PERIOD
  MAKE.OCTAVE :PITCH-1 :OCTIND :PERIOD * 1.0596
END
```

This is about as far as we can proceed in Logo before we know the specifics of the implementation of the note-generating routine. This machine-language routine should sound a note with a specific period for a certain length of time. As mentioned before, the way to generate a tone on the Apple speaker is to cause a click, wait for the period to expire, and keep doing this until the note is supposed to be over.

The heart of our machine language routine will be a subroutine called CLICK (listed below). This routine is called repeatedly with the (16 bit) period in locations PER.H and PER.L. It copies them to another place so that they will still be valid next time around CLICK is called.

One way to cause notes to have a certain duration would be to call the CLICK routine a certain number of times. Calling it twice that many times would result in a note twice as long. That is fine if only one period (pitch)

is used. Unfortunately, the CLICK routine by its very nature takes a different amount of time for different periods (i.e., different pitches); therefore, the routine that plays notes must convert the actual duration to the number of clicks to generate.

If we pick a base pitch, and scale the durations of all other pitches from that one, our problems will be solved. The number of times to call the CLICK routine for a given duration and period is :DURATION * (:BASE.PERIOD / :PERIOD). This scaling is called *normalization*. It is much easier to do the required multiplication and division in Logo than in machine-language, so we'll calculate this scaling factor in Logo. The machine-language routine will take this number as an input, and call the CLICK routine that many times.

Here is the entire machine-language program for playing notes:

```
TO MCODE
[MAKE "SPKR $ "C030]
[MAKE "DUR.L :NARG2]
[MAKE "DUR.H :NARG2+1]
[MAKE "PER.L :NARG2+2]
[MAKE "PER.H :NARG2+3]
[MAKE "COUNT.L :USERPZ]
[MAKE "COUNT.H :USERPZ+1]
[(PRINT [STARTING ADDRESS:] :ORG)]
TONE:   LDA ! DUR.L
        ORA ! DUR.H
        BEQ EXIT                ;A duration of 0 means no note.
        LDA ! DUR.L
        SEC
        SBC # 1
        STA ! DUR.L
        LDA ! DUR.H
        SBC # 0
        STA ! DUR.H
        JSR CLICK
        JMP TONE
EXIT:   RTS
CLICK:  LDA ! PER.L
        STA ! COUNT.L
        LDA ! PER.H
        STA ! COUNT.H
        BIT ! COUNT.H
        BVS PDLOOP
        LDA SPKR            ;Click
PDLOOP: LDA ! COUNT.L
        ORA ! COUNT.H
        BEQ EXIT
        LDA ! COUNT.L
        SEC
        SBC # 1
        STA ! COUNT.L
        LDA ! COUNT.H
        SBC # 0             ;propagate carry
        STA ! COUNT.H
        JMP PDLOOP
[(PRINT "LENGTH: :DOT-:ORG "BYTES. )]
END
```

The loops that make up the body of the CLICK and TONE routines both have an interesting property: every iteration takes the same amount of time as every other. Some methods of writing these loops would have them run faster or slower when DUR.H (or PER.H) was 0. Those methods

would cause durations of 400 clicks not to be twice as long as durations of 200.

Note that in the CLICK routine there are some instructions that are outside the loop, and are executed once for each period of the tone. The time they take has an affect on the pitches produced. It is just like adding a small amount to every period. To counteract this, we subtract a small amount from each period. This factor is the FUDGE constant.

Some method of interfacing the machine-language routine to the Logo procedures is needed. The .CALL primitive is provided for just this purpose, but allows passing only one input. What we need is a way to pass both the period (PER.H and PER.L) and the duration (DUR.H and DUR.L). A careful look will show that this adds up to 32 bits, which is the number of bits .CALL can pass to the machine-language programs it calls. If we arrange memory locations so that DUR.L/DUR.H are the low two bytes of the input to .CALL, and PER.L/PER.H the high two, the following procedure will give the two inputs to machine-language routines:

```
TO .CALL.2 :ADDR :INPUT1 :INPUT2
  .CALL :ADDR (ROUND :INPUT2) + (ROUND :INPUT1)*65536
END
```

Note the use of the ROUND primitive. Were it not called, non-integer periods would cause the result of the multiplication not to be a multiple of 65536, interfering with the duration. The PLAY.NOTE procedure is a combination of this procedure and the normalization step mentioned before (The three inputs to .CALL.2 are put on separate lines below, to make them more readable):

```
TO PLAY.NOTE :PERIOD :DURATION
  MAKE "PERIOD THING (WORD "# :PERIOD)
  .CALL.2 :TONE
          :PERIOD - :FUDGE
          (:DURATION * :BASE.PERIOD / :PERIOD)
END
```

The Music demonstration program

There are two music-related files on the Utilities Disk. One is a Logo file called MUSIC, and the other is a file of saved machine-language routines called MUSIC.BIN. To try out the music demo, type READ "MUSIC and SETUP. All the procedures shown here are included.

6.5. Useful Memory Addresses

This section contains brief descriptions of addresses in the Logo program that serve as "hooks" for modifying Logo with .EXAMINE and .DEPOSIT and for interfacing assembly language programs to Logo as described in section 6.2. The actual values of the addresses are contained in a file called ADDRESSES that is included on the Logo utilities disk. Beware that the actual values of these addresses may change with new releases of Logo. Executing READ "ADDRESSES in Logo will define the addresses as normal Logo variables whose values are integers. (Comments in the file also give the values of the addresses as in hexadecimal notation.) When using an address, it should be preceded with the character : as in .EXAMINE :EPOINT.

Page zero locations:

EPOINT — Location of the current character in the edit buffer. Used by the editor and the EDOUT routine.

ENDBUF — The address of the last character in the edit buffer, plus one. The disk saving routines (see SAVMOD) save from $2000 to the address in this location. See the example on page 81.

SAVMOD — If the contents of this location is 0, READ and SAVE work normally. If it is non-zero, SAVE saves whatever is in the edit buffer (which can be text other than procedures and names) and READ restores the edit buffer from disk, but doesn't evaluate it. See section 7.1.

BKTFLG — If this location contains 1, then Logo attempts to print out objects in a manner such that they can be read back in. This is useful when you are printing to the EDOUT device. See section 7.3. All lists will be printed with brackets around them; none will be printed in "sentence" form. Funny-pnames will be printed with their funny quotes. PO NAMES will print out Logo variables and their values with MAKE "VARIABLE 3

instead of "VARIABLE is 3.

NOINTP — Controls the action of the special "interrupt" characters CTRL-F, CTRL-G, CTRL-S, CTRL-T, and CTRL-W. If the location contains zero (the default), these keys have their normal action in DRAW and NODRAW mode. If it contains 1, these characters have no special meaning, and will be recognized by READCHARACTER. CTRL-Z and CTRL-SHIFT-M are still enabled. To disable them also, deposit 255 in NOINTP.

CH, CV — These locations contain the current cursor location, in columns and rows, respectively. .EXAMINE :CH outputs the current horizontal cursor position. See the CURSOR primitive, page 36

OTPDEV — Contains the address of the routine currently being used for character output.

INPDEV — Like OTPDEV, but for character input.

USHAPE — Pointer to user-defined turtle shape. See section 5

SSIZE — Shape size for turtle or user shapes. Default = 1.

INVFLG — Determines whether characters will be white-on-black (default, contents = 255), black-on-white (contents = 0), or flashing (contents = 64).

NARG2 — Second input to .CALL. 4 bytes. See section 6.2.

NARG1 — First user-available temporary location. All memory from here to ANSN4 + 3 is available for user routines. See section 6.2.

ANSN4 — Last user-available temporary location. This and the next three bytes are free.

USERPZ — First user-available permanent page zero location. From this location to $FF is not used by Logo.

HIMEM — (.EXAMINE :HIMEM) + 256*(.EXAMINE :HIMEM + 1) outputs the highest address available for user machine-language programs. It is set by the

MAXFILES 1 DOS command to $9AA5. See VZZZZZ.

Other useful addresses

OTPFX2	Jumping to this address will cause the .CALL to output the integer stored in NARG2 through NARG2 + 3. See section 6.2.
OTPFIX	Like OTPFX2, but return to Logo with value the integer stored in the four successive bytes starting with the page-zero location pointed to by the Y register.
OTPTRU	Jump to this routine to output "TRUE.
OTPFLS	Output "FALSE.
GETRM1	Loading from or storing to this location twice, e.g., LDA GETRM1, LDA GETRM1, enables Logo locations in extended memory and disables Monitor ROM.
KILRAM	Referencing this location enables the Apple monitor ROM. It is enabled during .CALL execution unless explicitly disabled.
PPTTP	An alternate exit for user machine-language routines. Jumping to this address runs the Logo primitive TOPLEVEL (page 35). It is useful for to return to Logo in this manner when some error condition has occurred, making it inappropriate to continue executing .CALLing procedure.
COUT	Logo's normal screen character-output routine. Prints the character in A on the screen.
EDOUT	Routine to place the character in the A register in the edit buffer. Deposits A in location pointed to by EPOINT and increments EPOINT. Returns without doing anything if EPOINT is greater than $3FFF. Can be used with OUTDEV to cause Logo to place text directly in the edit buffer. See the example on page 81.

PNTBEG	Routine to reset EPOINT to beginning of the edit buffer. Use before outputting to the buffer (with OUTDEV :EDOUT) the first time. See the example on page 81.
ENDPNT	Routine to set ENDBUF to EPOINT. Use when finished printing to the buffer. See PNTBEG, EPOINT, ENDBUF.
BELL	Routine to beep the bell. Use PRINT1 CHAR 7 to beep from Logo.
HOME	Homes the cursor and clears the screen.
CLREOP	Clear from cursor position to end of screen.
SCROLL	Scroll.
CLREOL	Clear to end of line.
FILLEN	Contains length of last file loaded.
FILBEG	Start address of last file loaded.
VZZZZZ	(.EXAMINE :VZZZZZ) + 256*.EXAMINE :VZZZZZ + 1 outputs the lowest address available for user machine-language programs. Although this address may be below $99A0, you should not place code in the intermediate region, since future releases of Logo may use that area of memory.

Chapter 7
Miscellaneous Information

7.1. Using the Logo System as a Text Editor

The Logo system is set up to read and save files that contain Logo procedures. By modifying the system, one can use Logo to read and save text files (and thus be able to use the Logo editor purely as a text editor), to use the disk for temporary storage, and to design "self starting" Logo programs.

Normally, to enter the editor you type TO followed by the name of a procedure to be created or edited. To work with text in Logo, it is necessary for the edit buffer in memory to be empty. Entering the empty edit buffer is achieved by typing TO followed by RETURN. At this point, text is written and edited in the same way that a procedure is written and edited. To save the text it is necessary to exit the editor. Instead of typing CTRL-C, which will define a procedure, CTRL-G is typed. This exits the editor without making any changes to it. (Note: For this reason, the text will be saved and printed exactly as it appears. It is not reformatted as procedures are.) However, if you then edit a procedure, enter graphics mode, etc., the text will be lost. Therefore, it is necessary to SAVE the buffer as a file on a disk. As with other files, any name can be used, such as SAVE "MYFILE. However, SAVE needs to be used differently than usual as described below. To make this easier, there is a TEXTEDIT file on the Utilities Disk which incorporates these concepts. For an explanation of TEXTEDIT, see the section on the Utility Disk.

Important: The SAVE primitive normally saves the names and procedures in the workspace by placing them in the edit buffer, and then saving the buffer on disk. If you want to write arbitrary text on disk to be loaded back into the edit buffer without being evaluated, you can set the "save mode" flag; it controls the action of READ and SAVE. Normally the memory location SAVMOD contains zero. When you deposit any non-zero number, SAVE will save the previous contents of the edit buffer (rather than saving workspace), and READ will not evaluate the edit buffer after

reading it in from disk. Nothing in Logo but GOODBYE resets this flag. The actual address of the flag may be found in the file ADDRESSES on the Utilities Disk, which can be accessed by typing READ "ADDRESSES. This should be done before creating the text, or it will be lost. After CTRL-G is typed and before the file is saved, you should type

```
.DEPOSIT :SAVMOD 1
```

If you are going to use the Logo procedure editor as a text editor for an entire session, you might want to type this in at the beginning. If you should want to read or save some procedures (or names), just type

```
.DEPOSIT :SAVMOD 0
```

and things will be back to normal.

To get the file back to work on at a later date, READ "ADDRESSES from the utilities disk, type .DEPOSIT :SAVMOD 1, and READ "MYFILE from your own disk. Then type ED followed by a RETURN. Do not type TO after reading the file or you will start in a new empty buffer and have to read in the file again.

7.1.1. Printing Files

Were there no means to print files, the Logo screen editor would be useless for editing text. The following procedures will print the contents of the edit buffer to the peripheral in slot SLOT. Before using it, you will need to make ENDBUF have the value listed in the Logo ADDRESSES file.

```
TO HARDCOPY
  OUTDEV :SLOT
  PRINTMEM 8192 256*(.EXAMINE :ENDBUF+1)+.EXAMINE :ENDBUF
  OUTDEV 0
END

TO PRINTMEM :FROM :TO
  IF :FROM > :TO STOP
  PRINT1 CHAR .EXAMINE :FROM
  PRINTMEM :FROM+1 :TO
END
```

Immediately after READing in a file, typing HARDCOPY will print the contents of the file. If you have been using the Logo screen editor as a

text editor, typing HARDCOPY after typing the CTRL-G editor command will print the contents of the edit buffer.

This method is not the most efficient one for printing listings of programs. See page 17.

7.2. Self-starting files

You can use SAVMOD to append arbitrary text to the end of procedures and names to be saved on disk. This is useful if you have a program that you want to start automatically every time a certain file is loaded in. For example, suppose you want a procedure called SETUP to be run automatically every time you read in the file called GAME. This can be accomplished by arranging things so that the command SETUP is executed automatically each time the GAME file is read in.

To do this, define all the procedures needed for GAME. Type EDIT ALL to get the entire workspace into the edit buffer. Then, go to the end of the buffer (using CTRL-F) and insert commands you want executed directly (SETUP, for example). Then, type CTRL-G to exit the editor and then type

```
.DEPOSIT :SAVMOD 1
SAVE "GAME
.DEPOSIT :SAVMOD 0
```

Now, whenever the GAME file is loaded, the procedures will be defined and the SETUP instruction that you appended to the end of the edit buffer will be executed.

7.3. Printing to Disk Files

Another use of SAVMOD is to enable you to write Logo programs that create Apple DOS Binary files. Since this implementation of Logo uses binary files to store procedures, you can write text to a file and have Logo read it in as a program later. Normally, the DEFINE primitive (31) is better for defining procedures; however, certain applications call for printing the text of procedures into a file. For example, a procedure that would save a list of procedures in a given file could use the following protocol for writing to the disk file. (The PSAVE program on the utilities disk uses this method.)

Since this implementation of Logo doesn't support arbitrary reading from disk, these procedures are useful only for a particular set of applications, and are printed here mostly for information purposes.

The following Logo procedures supply a DPRINT facility for "printing" to disk files. To use it, execute OPEN with the name of the file as input. Then, use the DPRINT command to print to the file buffer. To close the file, type CLOSE. This will save things on the disk in the file that is currently "open." You cannot use the editor or graphics while a file is open, nor can you print more than 8192 characters. Any extra characters will be ignored.

```
TO OPEN :FILE
 MAKE "OPEN.FILE :FILE      ;SAVMOD, PNTBEG, EDOUT, EPOINT,
 .CALL :PNTBEG 0            ;ENDBUF, and ENDPNT are found
END                         ;in the ADDRESSES file.

TO DPRINT :THING
 OUTDEV :EDOUT
 PRINT :THING
 OUTDEV 0
END

TO CLOSE
 .CALL :ENDPNT 0                    ;updates end-of-buffer pointer.
 .DEPOSIT :SAVMOD 1
 SAVE :OPEN.FILE
 ERNAME "OPEN.FILE
 .DEPOSIT :SAVMOD 0
END

TO OPEN.FOR.APPEND :FILE
 MAKE "OPEN.FILE :FILE
 .DEPOSIT :SAVMOD 1
 READ :FILE
 .DEPOSIT :SAVMOD 0
 .DEPOSIT :EPOINT .EXAMINE :ENDBUF
 .DEPOSIT :EPOINT+1 .EXAMINE :ENDBUF+1
END
```

For assembly-language programmers: The location ENDBUF is Logo's end-of-edit-buffer pointer. The edit buffer begins at location $2000, and extends to $3FFF. PNTBEG merely sets ENDBUF to $2000. EDOUT is a routine that takes a character in A and places it at the location pointed to by EPOINT, and increments EPOINT. The disk saving routine saves from $2000 to ENDBUF, so CLOSE has to update ENDBUF from EPOINT by calling the ENDPNT routine.

If you are writing out a file that Logo should be able to read back in and execute or interpret as data, then you probably want brackets to be printed around toplevel lists. That is to say, Logo normally behaves like this:

```
PRINT [1 2 [A B] 3]
1 2 [A B] 3
```

but you probably would rather have it behave like this:

```
PRINT [1 2 [A B] 3]
[1 2 [A B] 3]
```

Deposit 1 in location BKTFLG to obtain this feature, and restore it to 0 to get back the default behavior.[1] The .DEPOSITs should be done at the beginning and end of the DPRINT routine, so as not to interfere with other Logo printing operations.

Additionally, BKTFLG controls the action of PO NAMES. If you execute a PO NAMES while BKTFLG contains 1, then the names will be printed out in this form:

```
MAKE "NUM 259
```

7.4. Various System Parameters

This section contains various esoteric information about Logo and about this specific implementation. It is certainly not necessary to know what is presented here in order to use Logo; these topics are covered for the curious.

The Graphics Screen

When pointing straight up, the turtle can go 121 steps before wrapping around to the bottom of the screen. It can go 120 steps downward before wrapping around to the top. It can go 140 steps when pointing the the left, and 140 when going to the right. If you change the aspect ratio (see the .ASPECT primitive, page 40), then the allowable vertical range will change, but the horizontal range will remain the same.

---

[1] Besides using this feature in writing disk files, you may also find it convenient to have Logo print top-level brackets when you are dealing with list processing applications. That way, a list consisting of a single word will not be printed in the same way as the word itself.

Numbers

The smallest number on which Logo can perform operations is 1N38, and the largest is 9.9999E38. The largest positive number which is not "floating point" is 2147483647, and the largest negative is -214783647.

ASCII Values

There is a correspondence between the characters available in the Logo character set and the numbers 0-255. The ASCII primitive, if given a word of one letter, outputs the number associated with that letter. The CHAR primitive is the inverse, returning a single-letter word. The character represented by 0 (often called "null") is special in Logo: it represents the empty word. Just as SENTENCE ignores empty lists as input, WORD ignores the empty word. It is impossible to make a word which contains the empty word, unless that word is itself the empty word.

The READCHARACTER primitive, abbreviated RC, reads a key from the keyboard and outputs a single-letter word. There are certain "interrupt' keys that will never be output by RC. These are CTRL-F, CTRL-S, CTRL-T, CTRL-SHIFT-M, CTRL-W, CTRL-Z, and CTRL-G. The functions these keys provide are available whenever Logo is in draw or nodraw mode. The following table shows the ASCII values of selected keys. To find out the ASCII value of any key, type PRINT ASCII RC, and type the key.

Key	ASCII Value
ESC	27
LEFT ARROW	8
RIGHT ARROW	21
CTRL-SHIFT-P	95
CTRL-SHIFT-N	30

Sometimes it is useful to be able to disallow CTRL-G, or to use some interrupt character for purposes other than the function to which it is assigned. For these cases, Logo provides a hook for turning off the special meanings of all the above mentioned interrupt characters, except for CTRL-Z and CTRL-SHIFT-M. .DEPOSITing 1 in location NOINTP disables

interrupt characters.[2] Deposit 0 to re-enable them. See page 74 for a discussion of special memory locations.

When interrupt characters have been disabled, the READCHARACTER primitive will output on any key pressed (except of course, CTRL-Z and CTRL-SHIFT-M). A typical use of this feature is a system like the INSTANT program included on the Logo Utilities Disk. The program could disable interrupt characters and assign its own meanings to the characters normally reserved for special immediate actions in Logo.

Another occasion where disabling interrupts is useful is in procedures which do things which must be undone before returning to toplevel. If the user presses CTRL-G during the execution of a procedure which temporarily changes OUTDEV to some other device, all output from then on[3] would be directed to the alternate device. The following procedure, which uses the NOINTP feature, can be executed without fear of causing "STOPPED!" or "PAUSE" messages to be sent to the device.

```
TO TCMD :CMD :ARG
.DEPOSIT :NOINTP 255
OUTDEV 7          ;device in slot 7.
(PRINT :CMD :ARG)
OUTDEV 0
.DEPOSIT :NOINTP 0
END
```

Line length

Lines typed in to Logo in the line editor may not be more than 256 characters long. Additionally, the list which is input to RUN and REPEAT, and each sub-list in the second input to DEFINE must abide by this restriction.

Lines typed in in the screen editor (as with TO *procedurename*) may be of any length, as long as it fits in the edit buffer. Similarly, lines read in from disk files may be of any length.

---

[2] .DEPOSITING 255 in the location will disable all interrupt characters; be careful.

[3] Until another OUTDEV or CTRL-SHIFT-M.

The edit buffer is 8192 characters long.

Storage in Logo

Logo stores procedures much more efficiently than most other languages. Each procedure is stored as a list of lines.[4] The lines are lists of other lists and words. Each word takes up the same amount of space every time it is used, no matter how many characters it has. Thus, there is almost no penalty for using long, descriptive procedure and variable names.

When Logo runs out of storage space, it enters a process called garbage collection. This simply means that Logo is finding out what parts of memory are not being used, and makes a big list of all of them. Then, when Logo needs to use a memory location, it takes it off of this list.

Since Logo can't do anything else (like run your procedures) when it is garbage collecting, the process can interfere with certain programs where real-time response is important. If this becomes annoying, place calls to the .GCOLL primitive at natural pauses in the program.

7.5. Memory Organization Chart

This chart describes how the Logo system uses the available address space in the Apple II.

---

[4] Actually, this implementation of Logo usually stores procedures as arrays of arrays, since that method takes half as much space; however, when there isn't enough contiguous memory, Logo uses the list-of-lists method. It is possible for the curious to tell how procedures are stored: If each line is indented one space, the procedure is stored in the array form. If not, it is stored in the rarer list form. This information is completely arcane.

```
Nil:        $0000 - $0003: $   3 bytes The empty list.
Misc.:      $0004 - $07FF: $ 7FC bytes Buffers and impure.
Stacks:     $0800 - $1BF5: $13F6 bytes Stacks (PDL, VDPL)
Vectors:    $1BFC - $1BFF: $   A bytes Re-entry addresses
Othercode:  $1C00 - $1FFF: $ 400 bytes I/O subroutines
Buffer:     $2000 - $3FFF: $2000 bytes Editor/graphics buffer
Logo:       $4000 - $999F: $599F bytes (23K bytes) Logo code
User:       $99A0 - $9AA5: $ 105 bytes User Machine Code
DOS:        $9AA6 - $BFFF: $255F bytes DOS code, buffers
I/O:        $C000 - $CFFF: $1000 bytes Mapped I/O addresses
Nodearray:  $D000 - $F65F: $2660 bytes (2456. nodes) Nodespace
Typearray:  $F660 - $FFF7: $ 998 bytes (2456. bytes) Type-codes
Ghostmem:   $D000 - $DFFF: $1000 bytes Static storage
Unused:     $FFF8 - $FFF9: $   2 bytes
Intrpts:    $FFFA - $FFFF: $   6 bytes Interrupt vectors
```

Index

* 28

+ 28

– 28

.ASPECT 40
.BPT 40
.CALL 40, 60
.CONTENTS 41
.DEPOSIT 41, 59
.EXAMINE 41, 59
.GCOLL 41
.NODES 41

/ 28

; 41

< 28

= 29

> 28

Addresses, useful 73
ALLOF 33
Animal 50
ANYOF 33
Arcs 47
Arrow keys 4, 15
ASCII primitive 36
ASCII values, table of selected 83
ATAN 28

BACK 25
BACKGROUND 20, 25
Beep 76

Bell 76
BF 29
BG 20, 25
BK 25
BL 29
Brackets 4
Bugs 6
BUTFIRST 29
BUTLAST 29

CATALOG 22, 38
Catalog, Utilites 43
CHAR 36, 83
Characters, interrupt 83
Circles 47
CLEARINPUT 36
CLEARSCREEN 25
CLEARTEXT 36
CO 39
Color 20
Commands, editing 14
CONTINUE 39
Control characters 4, 11
Coordinates, graphics 82
COS 28
Cosine 28
CS 25
CTRL key 4
Ctrl-A 15
Ctrl-B 15
Ctrl-C 10, 15
Ctrl-D 15
Ctrl-E 15
Ctrl-F 11, 12, 15, 25, 83
Ctrl-G 10, 12, 15, 83
Ctrl-K 15
Ctrl-L 15
Ctrl-N 15
Ctrl-O 15
Ctrl-P 16
Ctrl-S 11, 12, 27, 83
Ctrl-shift-M 12, 16, 36, 83
Ctrl-shift-P 12
Ctrl-T 11, 12, 27, 83

Ctrl-W 12, 83
Ctrl-Z 12, 83
Cursor 9, 36
CURSOR program 46

DEFINE 31
Demonstration files 43
Diskette, Utilities 43
DOS 38
DPRINT 47
DRAW 25
Draw mode 10

ED 32
Edit 12, 31
Edit mode 9
Editing Commands 14
ELSE 33
Empty list 29
Empty word 29
END 32
ER 32
ERASE 32
ERASEFILE 22, 38
ERASEPICT 23, 38
ERNAME 32
ESC key 4, 15

FD 25
Files 21
FIRST 30
FORWARD 25
FPUT 30
FULLSCREEN 11, 25
Fullscreen mode 11

GO 34
GOODBYE 34
Grappler 18

Hain, Stephen 1
Hardcopy 17
HEADING 25
HIDETURTLE 25

HOME 25
HT 25

IDS Color Printer 19
IF 34
IFF 34
IFFALSE 34
IFT 34
IFTRUE 34
INTEGER 28
Interrupt Characters 83

Keys, editing 14
Klotz, Leigh 1

LAST 30
LEFT 26
LIST 30
LIST? 30
Listings 17
Logo, starting 5
LPUT 30
LT 26

MAKE 33
Memory, addresses of interesting locations in 73
Minsky, Henry 54
Modes 9
Music 67

ND 10, 26
NODRAW 10, 26
Nodraw mode 9
NOT 34
NOTRACE 39
NOWRAP 26
NUMBER? 28

OP 35
OPCODES 66
OUTDEV 16, 36, 63
OUTPUT 34

PADDLE 36

PADDLEBUTTON 36
PAUSE 39
PC 20, 26
PD 26
PENCOLOR 20, 26
PENDOWN 26
PENUP 26
Pictures, printing 18
Pictures, saving on disk 23
PO 39
POTS 39
PR 37
Primitives, descriptions of 25
PRINT 37
PRINT1 37
Printers 17
Printing 17
PRINTOUT 38
Procedures,Printing 17
PU 26

QUOTIENT 28

RANDOM 28
RANDOMIZE 29
RC 37, 83
RC? 37
READ 22, 39
READCHARACTER 37, 83
READPICT 23, 39
REMAINDER 29
REPEAT 35
REPEAT key 4
REQUEST 37
RESET key 5
Restarting Logo 40
RIGHT 26
ROUND 29
RQ 37
RT 26
RUN 35

SAVE 22, 39
SAVEPICT 23, 39

SE 31
SENTENCE 30
SETH 26
SETHEADING 26
SETSHAPE, in shape editor 55
SETX 26
SETXY 26
SETY 26
Shapes 53
Shapes, editing 54
SHIFT key 3
SHOWTURTLE 26
SIN 29
Sine 29
SIZE, in shape editor 55
Sobalvarro, Patrick 1
SPLITSCREEN 11, 27
Splitscreen mode 10
SQRT 29
ST 26
STOP 35
System bugs 6

TEST 34
TEXT 32
TEXTEDIT 46
TEXTSCREEN 27
THEN 34
THING 33
THING? 33
Tintinabulation 76
TO 32
TOPLEVEL 35
TOWARDS 27
TRACE 39
TS 27
Turtle, floor 48
TURTLESTATE 27

Utilities Disk 43

WORD 31
WORD? 31
WRAP 27

XCOR 27

YCOR 27

Terrapin, Inc.
222 Third Street
Cambridge, Massachusetts 02142
(617) 492-8816