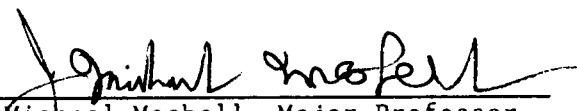
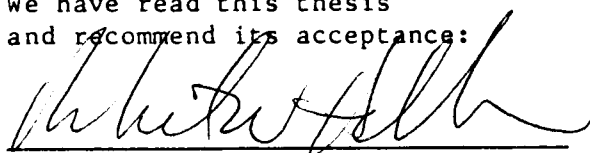


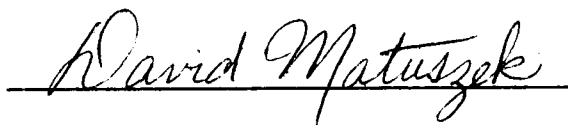
To the Graduate Council:

I am submitting herewith a thesis written by John R. Stambaugh, Sr., entitled "The Apple FELIX Pseudo-Machine." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

  
\_\_\_\_\_  
J. Michael Moshell, Major Professor

We have read this thesis  
and recommend its acceptance:

  
\_\_\_\_\_

  
\_\_\_\_\_

Accepted for the Council:

  
\_\_\_\_\_  
Vice Chancellor  
Graduate Studies and Research

THE APPLE FELIX PSEUDO-MACHINE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

John R. Stambaugh, Sr.

June 1981

**3051868**

## ACKNOWLEDGEMENTS

The author wishes to thank his advisor on this thesis, Dr. Michael Moshell. Because of Dr. Moshell's innovative research in microcomputer-based animation graphics, the author was given the opportunity to work in this fascinating field.

Sincere appreciation is due to Dr. R. W. Heller and Dr. David Matuszek for their contributions and service on the committee.

Special gratitude is extended to the author's family, especially his wife, for their encouragement and support.

## ABSTRACT

A cartoon animation "pseudo-machine" language for the Apple II computer is described. It is designed to process numbers and character strings, but the major emphasis is given to commands that manipulate visual objects on a color television screen. This language is designated Apple FELIX. It is patterned after, and shares the same set of core instructions with, the FELIX/8080 language. Thus, it is one manifestation of what is intended to be a general graphics language for a variety of microcomputers. Since extensive documentation already exists on the core instruction set, the emphasis in this report is on the additional instructions introduced in Apple FELIX and the peculiarities of this version of the language as dictated by the environment of the Apple host computer.

## TABLE OF CONTENTS

| CHAPTER                                            | PAGE |
|----------------------------------------------------|------|
| 1. INTRODUCTION . . . . .                          | 1    |
| 2. GENERAL DESIGN OF FELIX . . . . .               | 14   |
| 2.1 Objects, Buffers, and the SHOW Operation       | 15   |
| 2.2 FELIX Parameter Types                          | 19   |
| 2.3 The FELIX Instruction Set                      | 21   |
| 2.4 Text Output                                    | 27   |
| 2.5 The WRITE and ARITH FELIX Operations           | 29   |
| 3. IMPLEMENTATION OF APPLE FELIX . . . . .         | 32   |
| 3.1 Memory Map of the Display Area                 | 32   |
| 3.2 The FIND Algorithm                             | 34   |
| 3.3 The SIN/COS Table                              | 36   |
| 3.4 2-Byte Sine and Cosine Values                  | 38   |
| 3.5 Details of the Parameter Formats               | 40   |
| 3.6 The SWEET16 Pseudo-Machine                     | 42   |
| 3.7 SWEET16 and the FELIX Registers                | 45   |
| 3.8 The Details of Text Output                     | 48   |
| 3.9 The Routines SLOW and MOVTEXT                  | 50   |
| 3.10 The Role of SHOW in Text Output               | 53   |
| 3.11 Alternative Mode of Operation for PRINT       | 55   |
| 3.12 The Apple FELIX Trace                         | 58   |
| 3.13 GETSTR and BLNKRD                             | 58   |
| 3.14 The Permanent Objects and the WRITE Operation | 60   |
| 3.15 The 6502 Routines of CIPHER                   | 65   |
| 3.16 The FELIX Program FELIX-ARITH                 | 72   |
| 3.17 The SOUND and NOTES Operations                | 76   |
| 4. FUTURE DEVELOPMENT . . . . .                    | 81   |
| REFERENCES . . . . .                               | 83   |
| APPENDIX . . . . .                                 | 86   |
| VITA . . . . .                                     | 95   |

## CHAPTER 1

### INTRODUCTION

The field of computer graphics has greatly expanded with the introduction of inexpensive microcomputers which support color graphics. Software systems of various types have been developed to support graphics on small computers. These include the FELIX, RASCAL, ANIMA, PILOT, Zgrass, and Turtlegraphics systems, to be discussed below.

This thesis is an outgrowth of the FELIX project at The University of Tennessee. Before FELIX/8080 and Apple FELIX are discussed, some of the work done by others will be considered. Two of these, Zgrass and RASCAL, will be discussed in some detail.

Lucido (14) describes the ANIMA graphics language developed under the auspices of the Computer Graphics Research Group at Ohio State University. The Lucido article also contains bibliographical references to the ESP3 and DIAL graphics-oriented programming languages. ANIMA is described as a development language, rather than an end-user language, but user languages and packages based on ANIMA are promised. Dr. Moshell has a video cassette which shows impressive examples of graphics produced by what is called the ANIMA II system.

There is a graphics capability built into the Apple PILOT system. PILOT is a high-level language particularly suited to the needs of courseware developers. Apple PILOT is designed so that high-resolution color graphics can be included in lessons, text may be written on the graphics screen, and the "pictures" can be stored on diskette. However, no animation is supported.

Another graphics package from the Apple people is called Turtlegraphics (15). Its name derives from the "turtles" devised by S. Papert and his co-workers at MIT. To make computer graphics less intimidating to children, Papert invented the idea of a "turtle" who could walk a given distance and turn through a specified angle dragging a pen along. Turtlegraphics uses Apple's high-resolution graphics.

Zgrass is a graphics-oriented programming language and operating system written in assembly language for the Z80 microprocessor by Nola Donato, Jay Fenton, and Thomas DeFanti (16). Zgrass started out as GRASS (Graphics Symbiosis System), a language designed to bring the power of computer graphics within the grasp of artists and educators at Ohio State. (Dr. DeFanti, the author of the source article for this information, is now associated with the University of Illinois at Chicago Circle.) GRASS depends on \$120,000 of equipment centered around a PDP-11/45 computer and a Vector General 3DR Display system.

Zgrass has been under development for ten years. The first version of Zgrass had a resolution of 160 by 102 pixels, using two bits per pixel. The present version has a resolution of 320 by 204 pixels, again using two bits per pixel. There are 256 colors available, with a maximum of four colors on the screen at any one time.

DeFanti and his co-workers are developing a system which they hope can sell in the neighborhood of \$3000. The current configuration includes: a Z80 processor with 16K of EPROM and 48K bytes of programmable memory; custom-graphics integrated circuits and floating-point hardware; dual UARTS for connection to larger computers, printers, etc.; a color monitor; an alphanumeric terminal (which the user provides);

and provision for floppy disks, tablet, and other I/O devices. Eight such units have been put together and tied into the Bell Laboratory-developed UNIX operating system since January 1980.

As the title of DeFanti's article might suggest, he has devoted considerable thought to the problem of control structures--what he defines as "the program-flow manipulation features of the language that you use to beat your computer into submission." He concludes that there should be a variety of control structures; that they should function in a context of "parallelism;" and that there should be a flexible priority system.

These conclusions are made manifest in the Zgrass system. For instance, parallelism is exhibited by the fact that several different programs, or macros, as DeFanti calls them, can run at the same time by interleaving execution of the macro instructions. DeFanti calls these pre-written programs "background macros."

While the background macros are executing, the keyboard is active, and any command that the user types has priority over the background macros.

There is a third level of operation that has priority over keyboard input: the user may specify programs to run as the result of a clock interrupt. The frequency of execution can range from one to 32,768 sixtieths of a second. Thus, Zgrass has three effective levels of priority with parallelism at two of the three levels.

The component parts of the total RASCAL system that will be discussed, in turn, are RASCAL, INTERPAS, MAKER, PAINTER, KEYJOY, and LORES. RASCAL and INTERPAS are the major components of the system.

The developers of the RASCAL system are Dr. Moshell, Dr. Charles E. Hughes, and G. W. Amann. This development followed Dr. Moshell's work on the FELIX/8080 system.

RASCAL is a dialect of Pascal designed for cartoon animation using the high-resolution capabilities of the Apple II computer (13). A special Library Unit adds a variety of commands which cause figures to be moved, rotated, rescaled, recolored, and turned on and off. There are disk I/O commands that move background scenes from disk into the screen buffers; figures can be loaded from disk into memory. The command SAVESCENE causes the current scene to be stored on disk with the associated name.

The action verbs of FELIX and RASCAL are quite similar in many instances. Since the FELIX verbs are discussed in detail in subsequent pages of this report, that material shall not be duplicated here. There are differences, of course. FELIX has no facility, for instance, for changing the size of figures. The angle orientations of a figure in FELIX are changed by multiples of 15 degrees, not by degrees, as in RASCAL. In the present version of FELIX, individual figures cannot be turned on and off; an ONLIST instruction simply specifies all the figures or objects to be seen upon the next execution of the SHOW verb. On the other hand, two instructions found in FELIX, but not in RASCAL, are the PATH and DIST instructions. PATH allows an object to define a path on the screen for other objects. Each execution of PATH with a target object moves it one more step down the "path." Even "path objects" can be moved along a path. The DIST function returns the distance between two specified objects. It is very useful in action games.

In RASCAL, there are three scene control commands: VIEWPORT, FILLSCREEN, and PAINT. VIEWPORT (low-x, high-x, low-y, high-y) establishes a "window" with x and y limits specified by the arguments. FILLSCREEN (screen-color) fills the current viewport window with the specified color. PAINT (figure) causes the figure to be added to the scene. This can be used to place multiple copies of a figure in a scene. In the way of functions, RASCAL has five in number. These serve to tell the program the location, scale, and angle orientation of a figure.

An interesting aspect of RASCAL is that certain "special effects" are possible due to the fact that there are two full-screen images, foreground and background. As a means of "repairing" the foreground image, when a figure moves across the screen the background is copied into the foreground at the old image position. If, then, the background has some interesting scene and the foreground is blank, moving an object across the screen has the effect of "revealing" the scene at programmable rates, directions, and locations.

INTERPAS is a specialized tool for the teaching of computer programming. Its chief attribute is that it contains an interpreter for RASCAL; but it also contains many other special features to enhance its value in an educational environment.

A customized screen editor is included which has the following features: the ability to delete files and list directories from within the edit-mode; disk space compaction to circumvent lack-of-space error from thwarting a file save from the editor; and direct access to the interpreter, so that code can be immediately tried out as it is written. A filer program is included, which is similar to the UCSD Filer but customized for student use.

The interpreter supports IMMEDIATE-mode execution of commands. An initialization program can be run to "set up" a given graphics environment and then "jump out" to the immediate mode to allow the student to experiment with individual commands. INTERPAS also includes a converter program that converts INTERPAS source code into compilable UCSD source code, allowing debugged programs to run with the full speed of the UCSD system.

The INTERPAS interpreter itself has many features to aid the student in developing and debugging a program. Program execution may occur with a program listing. If this option is chosen, the user may also request single-step or delayed execution with the amount of delay specified by the user. The user's program can contain TRACEON and TRACEOFF procedure calls. Assignment statements in the program between these two calls have the results of their execution shown on the text part of the screen.

INTERPAS has a "one-line error" correction capability. The interpreter executes one Pascal source statement at a time, and if there is an error, an error message is produced immediately following a listing of the current statement. A user may either return to the editor or correct the offending statement. If the correction option is exercised, the program is immediately re-executed, and the current workspace will be updated for use by the editor.

MAKER is a graphics editor that is used to build figures for the RASCAL animation system. Although work is done on a single figure, several figures can be on the screen and in memory concurrently, and the user can switch back and forth among them. The figure presently being built or changed is designated the "work figure."

The size and angular orientation of the work figure can be changed by using the game paddles. Its position on the screen is controlled by a keypad. The figures are composed of lines, and the line presently being added to the figure can have its color set from the keyboard.

Some important procedures in MAKER are BKGRID, DIR, FILL, COMBIN, and RECOLOR. For precision in figure construction, BKGRID draws a grid into the background of the screen. The lines occur every 20 pixels. DIR gives a directory of the figures currently in memory. It lists each figure's name and shows that figure for two seconds. FILL gives a new background color as specified by the user. A figure may, of course, be multicolored; RECOLOR allows the user to change its first color. COMBIN merges two figures by adding one to the end of another. The combined figure retains the name of the first figure. MAKER adds a line of color NONE from the end of the first figure to the beginning of the second. Of course, commands exist for disk I/O operations.

PAINTER is a program that allows the user to paint pictures with the Apple Computer. These pictures can then be used as background scenes in RASCAL cartoon animations. A cursor is used as a paintbrush. The size and rotation of the brush are controlled by the game paddles, and the movement of the brush is controlled by a keypad. Keyboard facility is provided for the user to control the speed of movement of the brush. Keystrokes also set the color of the brush. The brush actually paints only when the button of Paddle 0 is held down. The button of Paddle 1 serves as an eraser by painting in the background color. Pictures can be saved on disk and loaded from disk.

Three other library UNITS offered by the RASCAL development team are KEYJOY, FREEJOY, and LORES. KEYJOY is used with RASCAL to allow the

keyboard of the Apple computer to simulate two dual-axis joysticks. Also, numerical, character, and "color" input functions can be updated according to the most recent keystroke. FREEJOY performs the same functions, except for the color input function, for use without RASCAL.

The use of LORES with a Pascal program provides it with low-resolution graphics capabilities. The user can switch between graphics and text mode, fill the screen with any one of 16 colors, and plot points by specifying color and coordinate arguments. LORES has a function that returns the color of any pixel, and it has a MOD function that, unlike UCSD's, works correctly for negative numbers. LORES and FREEJOY can be used together to build interactive programs.

Present plans call for making RASCAL and INTERPAS available to the public in the spring or summer of 1981. The RASCAL system is presently in use in many high schools as part of the NSF High School Computer Science Curriculum Project. This is a project at The University of Tennessee under the direction of Dr. Moshell.

Shortly after microcomputers with graphics capability appeared on the market, Professor Michael Moshell and a team of students at The University of Tennessee developed a microcomputer-based animation programming system which they called FELIX (Moshell, 4, 5, 6). Specifically, it is called FELIX/8080 since it has been implemented on an IMSAI 8080 microcomputer. The balance of the hardware consists of a "TV DAZZLER" display, a terminal, a dual floppy disk, and two joysticks.

FELIX is designed for use with microcomputers having low-resolution color television graphics displays. The TV DAZZLER has a 64 x 64 pixel screen. The Apple FELIX version, which is the subject of this report,

has a resolution of 40 x 48 in the full-graphics mode of operation and a resolution of 40 x 40 in the mixed-text and graphics mode. In this latter mode, provision is made for four lines of text at the bottom of the screen. The origin (0,0) in both FELIX systems is located in the lower left-hand corner of the screen. Thus, in the mixed mode of Apple FELIX, graphics rows 0 to 7 are sacrificed to text output.

The FELIX/8080 software package is available to the public from the CP/M User's Group. This software package includes the assembler for the FELIX language written (in 8080 assembly language) by Lee Wittenberg. A listing of this assembler can be found in Reference 6 of this report. Presently, there is not an assembler for Apple FELIX; writing such an assembler is a top priority item in the future development of Apple FELIX. Of course, an efficient solution of this problem for the owner of a Z-80 Softcard by Microsoft would be to customize Wittenberg's assembler for use on an Apple computer.

In an early statement concerning the future plans for FELIX, Moshell (4, p. 2) said the following:

Our vision of the future of FELIX includes a variety of experimental higher-level language compilers, screen editor programs, etc., all executable on any machine for which the FELIX pseudo-machine interpreter has been written. It also includes a variety of pseudo-machine realizations, even on the same hardware, all optimizing some feature of operations. Their common ground, their basic compatibility, will be that they all execute the FELIX core of instructions, on the FELIX-defined data structures.

Thus, one of the prime motivations for writing FELIX as a low-level, assembly-type language was the plan to develop transportable high-level graphics languages.

The need for such developmental work became much less important when UCSD Pascal became available for the widely-used Apple computer. Dr. Moshell, joined at this time by Dr. Charles E. Hughes, saw great potential for a graphics system in this new combination of software and hardware. Subsequently, Moshell and Hughes, together with G. W. Amann, developed the RASCAL system.

Despite the diminished priority of the high-level language development, the goal of transportability for FELIX is still being pursued, and this is certainly most easily attained in its assembly language form.

Students who feel a need to know "how things work" will never feel very "comfortable" with computers if they are only familiar with a high-level language. FELIX, with its vivid graphics effects, its error messages, and its trace mode of operation, should prove to be very useful in teaching assembly language. Since it is very easy to add new instructions to FELIX by writing a subroutine in the assembly language of the host computer, graphics experimentation and language learning can take place at this even more basic level.

Moshell (4, p. 1) describes FELIX/8080 as follows:

The system consists of an interpreter program for the FELIX "pseudo-machine" language, together with an assembler, editors, and "object doodler" (screen image editor), and other utility programs to create, store, and execute FELIX programs describing simple cartoons.

The purpose of the project which is the subject of this thesis has been to write an Apple II version of the interpreter program for the FELIX language. The central routine of the interpreter is the EXEC routine. Using its table of "op-routine" addresses, EXEC fetches and executes

the instructions of the FELIX language program. There is an op-routine for each FELIX instruction.

In some cases, an op-routine, when called by EXEC, in turn calls one of the "base routines" to do much of the actual work. This is always true, for instance, if the instruction involves parameter passing; in this case, the base routines INPAR and/or OUTPAR are called.

The hardware configuration for Apple FELIX is centered around a 48K Apple II computer with Integer BASIC and Utility Subroutines in ROM. The Utility Subroutines include the SWEET16 interpreter and the Floating-Point routines. In Apple FELIX, SWEET16 is heavily relied upon to implement the FELIX 16-bit registers. The Floating-Point routines are used in the op-routine for NOTES, the latest Apple FELIX instruction. A single-drive floppy disk, a joystick, and a SONY color television complete the hardware of the system. S-C ASSEMBLER II from S-C SOFTWARE was the assembler-editor used to produce the source programs and machine executable code.

It should be noted that there is no CRT screen or printer included in the hardware for text output. If text output, or echoed input, is required when working with graphics, the Apple is put into a mixed-text and graphics mode. In this mode, the bottom part of the screen is devoted to four lines of text, while graphics continues to be shown on the remainder of the screen. It turned out that meeting the challenges and the opportunities that this situation created consumed a substantial portion of the total time devoted to this project.

While on the subject of text I/O, it might be pointed out that one of the biases while working on this project was toward increasing

user interaction with FELIX programs. Also, there is the Apple FELIX instruction GETSTR, whereby a FELIX program can accept an entire character string. There is the case of the ARITH instruction which accepts arithmetic problems from the user. In the same vein, certain instructions have been provided which are intended to facilitate writing games in the FELIX language. Examples of the latter are KOUNT and PRINTK, instructions which increment and print, respectively, a four-digit decimal number. Such a number can be used to "keep score" in a FELIX language game.

The resolution of Apple FELIX is very low indeed. In the full-graphics mode, the screen is a 40 x 48-pixel screen, and in the mixed mode it is cut down to 40 x 40. Under these constraints, only simple, unsophisticated figures can be put on the screen. This is an oversimplification, but Apple FELIX, with its large pixels and bold colors, rather reminds one of a modern version of children's alphabet blocks. In any case, perhaps the most appropriate users of FELIX programs will be the "little people." Spurred on by these ruminations, in addition to the 16 variable objects provided for in FELIX/8080, Apple FELIX has been supplied with an extensive set of permanent objects; they are always read into memory with the FELIX system.

There is a figure (object) to represent each digit, each letter of the alphabet, and each of the following characters: '!', '\$', '%', '&', '\*', '+', ',', '-', '.', '/', '=', and '?'. Also, there are figures for the "less than" and the "greater than" symbols. The more easily recognized symbol for division (a horizontal line with a period above and below) is also represented by a figure. These figures are built on

a 5 x 7-pixel matrix; their style or "pattern" was copied from the characters produced by a Centronics printer.

Loading and execution of an Apple FELIX program is performed, after booting the Disk Operating System, by running a short BASIC program that consists of a series of BLOAD instructions that load into memory the FELIX system, the FELIX program, objects, and perhaps a background or two. After the BLOAD instructions, a CALL instruction to location 10464 (\$28E0) is executed.

As part of the DOS booting operation, a user written BASIC program--called the "greeting" program by the Apple people--is executed. One can make this greeting program the driver for a particular FELIX program. In this case, assuming the disk card resides in slot #6, for example, a "6 Control-P" command to the Apple Monitor will have the FELIX program up and running.

The focus of this report will be on those aspects of FELIX which are peculiar to the Apple version. Although it is not intended to duplicate the material in References 4 and 5, information about the general design of the FELIX system is given in Chapter 2, along with a discussion of the FELIX instruction set. Chapter 3 gives the details of specific Apple FELIX routines and data structures.

## CHAPTER 2

### GENERAL DESIGN OF FELIX

This chapter is begun with a description of the standard FELIX objects. Next, the memory display areas associated with the low-resolution graphics mode are identified. Because the manipulations of these objects and graphics buffers by the FELIX operation SHOW are pivotal in the overall design of the FELIX system, an outline of the operation of the SHOW routine is given.

FELIX works with 5-, 8-, and 16-bit numbers by way of 8 different "data" or "parameter" types. In Section 2.2, these parameter types are identified, and two very important base routines--INPAR and OUTPAR--are introduced that are the data handlers for FELIX.

In Section 2.3, a brief description of the "core" of the FELIX instruction set (i.e., the collection of those instructions that are implemented in both FELIX/8080 and Apple FELIX) is given. This is followed by a discussion of the additional Apple FELIX instructions.

It was mentioned in Chapter 1 that developing techniques to handle test output turned out to be one of the principal challenges of this project. Section 2.4 is devoted to a discussion of the Apple FELIX design for text output.

Chapter 2 concludes with a section devoted to two FELIX instructions--WRITE and ARITH--which are quite different from the others. As explained later, WRITE and ARITH are in the nature of "stand alone" projects rather than integral parts of the FELIX instruction set.

## 2.1 Objects, Buffers, and the SHOW Operation

We use the word "object" to signify a combination of pixels on the screen or a set of numbers which can be "decoded" to produce the pixel combination. The set of numbers consists mainly of a sequence of coordinate pairs.

As in FELIX/8080, Apple FELIX reserves the 4K memory block starting at address \$1000 for storage of up to 16 objects. These objects are designated by the hexadecimal numbers \$00 to \$0F. Each one, whether it is a 1-pixel object or a 124-pixel object, is allotted 256 bytes for its storage. Thus, if any FELIX program has an Object #3 associated with it, for instance, that object will be stored in the 256-byte area starting at address \$1300. Objects bearing the same number need have nothing in common other than their number designation and their storage area. This is in contrast to the permanent Apple FELIX objects discussed later in this report; in this latter set of objects, Object #\$43, for instance, is always the letter "C."

The first four bytes of every object contain the same type of information. The first byte contains some number between \$00 and \$0F designating the color of the object. FELIX has the ability to rotate an object about its "center of gravity." The unit of rotation is a 15-degree angle. Thus, 24 units of rotation constitute a complete revolution. The second byte of an object always contains an orientation number between zero and 23. The average of the x coordinates of the left-most and right-most pixels of an object is the x coordinate of the "center of gravity" of the object. The y coordinate is calculated using the lowest and highest pixel. The third and fourth bytes of an object always

contain the x and y coordinates, respectively, of the center of gravity of the object.

The sequence of coordinate pairs mentioned above follows these first four bytes; there is one coordinate pair for each pixel of the object. Since the x coordinate precedes the y coordinate in storage, x coordinates are stored in bytes with even-number addresses. If the number \$FF appears in one of these even bytes, it signals the presence of a special code in the subsequent (y coordinate) byte. At present, FELIX utilizes two types of special code: any number between \$00 and \$0F indicates a color change--subsequent pixels will be this new color--and the number \$80 indicates the end of the object. Finally, byte 255 of each object is reserved for use by the PATH operation to record the object's path position. Byte 254 is, thus, also available for some future special use.

The two areas of memory that contain display information in the low-resolution graphics mode are designated Page 1 and Page 2 in the Apple literature. Page 1 is made the active page simply by addressing location \$C054 with a 6502 instruction. Page 2 is "switched on" by addressing location \$C055. Page 1 and Page 2 are adjacent 1K blocks of memory; Page 1 starts at location \$400 and Page 2 at \$800.

There is another 1K buffer that is used in FELIX to aid in graphics output. It is not recognized by the Apple system as a graphics buffer and, consequently, must be moved onto Page 1 or Page 2 to be effective. This buffer is called BAKBUF, and in Apple FELIX it occupies the block of memory from location \$C00 to \$FFF.

Since the switches for Page 1 and Page 2 have already been mentioned, the locations of the other mode selection switches of the Apple system might be given at this point. The switches are given in pairs; as is true of the Page 1, Page 2 pair, when one member of a pair is "on," the other member is "off." The switches for (1) text or graphics, (2) low-res or high-res, and (3) full-screen or mixed-mode, are, respectively, (1) \$C051 or \$C050, (2) \$C056 or \$C057, and (3) \$C052 or \$C053.

In Apple FELIX the graphics switch (\$C050) and the low-res switch (\$C056) are always "on." On the other hand, both Page 1 and Page 2 and both the full-screen graphics and the mixed text and graphics modes are, of course, used. In the mixed mode, the bottom eight rows of pixels (rows 0-7 for FELIX) are given over to four text lines.

The terms SHOBUFF and ONBUFF are used in the nomenclature of the FELIX system. SHOBUFF is that 1K block of memory whose contents are presently controlling the screen output. ONBUFF is that block of memory being prepared for SHOWing. Apple Page 1 and Page 2 alternate at each execution of the FELIX operation SHOW between being ONBUFF and SHOBUFF.

Location \$100 is the starting address of a table called ONLIST. ONLIST contains, as its first entry, the number of objects that will be painted on ONBUFF when SHOW is next executed. Following the first entry are the actual object numbers themselves. ONLIST is supplied these values by the FELIX operation LISTOP. LISTOP is the only operation whose opcode is followed by a variable number of parameters--one for the "count" and one for each object.

Now that ONLIST has been introduced, the operation of SHOW can be described. SHOW checks ONLIST to see if any objects are listed there.

If so, and if rotation is not indicated, it paints these objects on ONBUF by utilizing calls to the routine PUT. PUT is a base routine that paints a pixel on ONBUF if passed a coordinate pair and a color number. Whenever rotation is indicated (by the presence of a nonzero orientation number), SHOW calls ROTATE to paint the object. Once the objects are painted on ONBUF, SHOW reverses the roles of Page 1 and Page 2 as ONBUF and SHOBUFF, and the new scene appears on the screen. Before returning, SHOW moves BAKBUF onto the new ONBUF. This is the first step in the preparation of the next scene.

The GET base routine is a complement to the PUT base routine. GET is supplied a coordinate pair and returns the color of the designated pixel in ONBUF. There are also PUT (opcode \$20) and GET (opcode \$21) FELIX operations. Both of these operations use three parameters: the x and y coordinates and the color parameter. The PUT and GET op-routines, of course, call the PUT and GET base routines, respectively. These latter two routines, in turn, call the base routine FIND, which is the central routine of both the "put" and "get" processes. FIND transforms a coordinate pair into the appropriate address in ONBUF. The intricacies of the memory map of the graphics buffers, Page 1 and Page 2, are explained in Section 3.1. Then, in Section 3.2, the algorithm used in the FIND routine is described.

The ROTATE routine uses stored sine and cosine values to calculate the rotated pixel positions. The sine/cosine table is described in Section 3.3. In order to alleviate the severity of round-off error in the ROTATE routine, 2-byte sine and cosine values were utilized. These values are discussed in Section 3.4.

## 2.2 FELIX Parameter Types

Many FELIX operations involve the use of one or more parameters. In a FELIX program, the parameter immediately follows the opcode of the associated operation. FELIX has eight basic parameter types, numbered 0-7. Depending on the type, a parameter requires one, two, or three bytes. In each case, however, the type is always indicated in the three most significant bits of the first parameter byte. This byte is designated the "param-header."

Brief descriptions of the parameter types follow. There is a 5-bit constant (Type 0). Type 1 and Type 5 are 8-bit constants, respectively. Type 2 and Type 6 are the respective 8-bit and 16-bit variable types. Type 3 indicates an in-port or an out-port. Although only a few of these are actually used in Apple FELIX, potentially there can be 32 in-ports and 32 out-ports. As examples, two very important in-ports are the X and Y A/D converters of the joystick.

Parameter Type 7 indicates the contents of one of eight 16-bit general-purpose registers. FELIX/8080 also has eight system registers that are included under Type 7, but these are not implemented in Apple FELIX. The Type 4 parameter indicates the contents of an array indexed by one of the eight general-purpose registers. Actually, there is a ninth parameter type which is used with the seven FELIX jump operations and the CALLOP operation. In this case, the opcode is followed by a 2-byte address, which is accessed directly by the op-routine without using a call to the routine INPAR, discussed below.

There are two base routines whose job it is to decode the format of the eight basic parameter types and then to get data from, or move data

into, the parameter in question. INPAR is called when data from a parameter is needed; OUTPAR puts data into a parameter. With the exception of the LISTOP op-routine, which is passed the information in the first parameter, it is the job of the op-routine to know how many parameters follow its associated FELIX instruction and, thus, how many times it must call INPAR and/or OUTPAR. In particular, it must know which parameters require an INPAR call and which parameters require an OUTPAR call.

INPAR always returns 8-bit data in the Y register and 16-bit data in the X register and the Y register, with the low byte in the Y register. INPAR sets the 6502 carry to signal 16-bit incoming data and then resets it otherwise. OUTPAR receives the value to be put into a variable or port from these same registers using the same format. When INPAR or OUTPAR is called, the FELIX program counter, FPC, is always pointing at the param-header of the next parameter to be processed. Before returning, INPAR and OUTPAR update the FPC so that it is pointing at the next param-header or at the opcode of the next FELIX instruction.

The logic involved in the processing of the Type 7 and Type 4 parameters--the registers and the register indexed arrays--is quite complex. In FELIX/8080, the appropriate routine was written in the PL/M language. In Chapter 3, a description of how SWEET16, a pseudo-machine created by Stephen Wozniak and resident in Apple II firmware, is used in conjunction with the 6502 assembly language to write the necessary code is provided. This description is contained in Sections 3.6 and 3.7. As a background for this material, the description of the parameter types is completed with a discussion of their formats in Section 3.5.

### 2.3 The FELIX Instruction Set

The SHOW, LISTOP, PUT, and GET FELIX instructions were dealt with in Section 2.1. In this section, the remainder of the instruction set is introduced. More information concerning the instruction set can be found in Moshell (4, pp. 19-22).

For the purpose of discussion, the instructions are placed into one of four categories: (1) graphics, (2) number, (3) control, and (4) miscellaneous. The graphics instructions are discussed first. Where appropriate, parameters are included with the instruction.

Execution of POSIT P,Q,R gives object P the center of gravity Q,R and recalculates the other coordinates accordingly. Unlike the rotation process, the new coordinate values replace the original ones in memory storage. FLY P,Q,R is similar to POSIT, except that Q and R are added to all x and y coordinates, respectively. Q and R may be negative numbers.

In the instruction PATH P,Q, Q is called the "path object." Each time PATH is executed, the next coordinate pair of Q is used as a center of gravity for object P, and P is POSITIONed by the PATH op-routine, which calls POSIT using POSCAL, the 6502 entry point of POSIT. Special code in Q can recolor (\$0-\$F) P or "freeze" it (\$82) at a given position. If the end of Q is reached, P starts at the beginning of Q when PATH P,Q is next executed. If, subsequent to an execution of PATH, the FELIX zero flag is set and byte 255 of P equals 4, then the end of path Q has been reached; if Z = 1 and byte 255 is greater than 4, P is in a "freeze" position. Finally, it should be mentioned that it is possible to apply the POSIT, FLY, and even a PATH instruction to a path object.

Since the zero flag has been mentioned, it should be pointed out at this time that the 8-bit Apple FELIX Status Word is located at \$5094. Bits 7, 6, 3, and 1 are the sign, zero, overflow, and carry flags, respectively.

SETANG P,Q and SPIN P,Q bear the same relationship to each other with respect to rotation that POST and FLY do with respect to translation; SETANG gives object P orientation number Q, and SPIN increments (or decrements) the orientation number by Q. These operations are performed Modulo 24.

RCLROP P,Q simply gives object P the color Q. If the object uses more than one color, only the first color (at byte 0) is changed. The instruction BKGDOP P paints the object P on BAKBUF. When WHERE P,Q,R is executed, the center of gravity coordinates of object P are placed in the variables Q and R. Execution of the instruction OBJR P,Q,R reads byte Q of object P and returns the value in variable R. OBJW P,Q,R writes value R to byte Q of object P.

The instruction DIST P,Q,R returns the "distance" between the objects P and Q in the variable R. The definition of this non-Euclidean distance is detailed in Moshell (4, p. 20). The final graphics instruction of the FELIX core instruction set is NEWBCK P. This instruction paints BAKBUF the color P. This clears BAKBUF of any objects that may have been painted on it.

In the graphics category, the additional Apple FELIX instructions are GRFMDE, MXTMDE, PTHSET P,Q, and MOVMEM P,Q. Two flags, XWRAP and YWRAP, have also been added to the POSIT op-routine. GRFMDE switches the screen to full-screen graphics from the mixed-text and graphics mode.

MXTMDE has the opposite effect. These two routines are discussed in Sections 2.4 and Section 3.8.

Execution of PTHSET P,Q places the value Q into byte 255 (or its equivalent--see Section 3.14) of object P. This instruction is commonly used to reset the path position of an object ( $Q = 0$ ), but it could also be used to manipulate the path position for other purposes--adding 2 to the contents of byte 255 could "unfreeze" an object.

MOVMEM P,Q moves a 1K block of memory with starting address P to the location starting at address Q. This instruction is used to move "fancy" backgrounds stored beyond address \$7000 onto BAKBUF.

Apple FELIX has the wrap-around feature for all objects that move off the screen; but sometimes it is desirable to remove the object from the screen or take other action if an object is about to be wrapped. The flag ZWRAP (\$508E) or YWRAP (\$508F) is set as soon as any pixel of an object is wrapped horizontally or vertically, respectively, during execution of a POSIT or FLY instruction.

The FELIX core instructions that have been grouped together as falling into the "number" category are: MOVE P,Q, COMP P,Q, COMPU P,Q, ADD P,Q,R, SUB P,Q,R, PUSH P, POP P, and RSCALE R,Q. As in FELIX/8080, MOVE, COMP, and COMPU affect the sign and zero flags; ADD and SUB affect these flags and also the overflow and carry flags.

COMP is a comparison of signed numbers and COMPU of unsigned numbers. SUB P,Q,E places the difference P minus Q into the variable R. The instruction PUSH P places the 16-bit value P on the FELIX stack which is located on page \$21. The high byte precedes the low byte on the stack. Location \$75 of page zero is used as the FELIX stack pointer.

After the execution of RSCALE P,Q, input from the Type 3 input, port P will be shifted Q places to the right (with sign-bit extension) before returning to the user. This is division by the Q(th) power of 2; thus, only values of Q in the 0-7 range are meaningful. This process allows for adjustment of the ranges for the joystick and other inputs.

In Apple FELIX, the variable COUNT, located at \$509E-\$509F, is reserved for a 4-digit decimal number. The main purpose of COUNT is to provide game-scoring capabilities for Apple FELIX programs. Execution of the instruction PRINTK places the contents of COUNT on the screen.

Apple FELIX has three number operations that involve COUNT: SETK P, KOUNT P, and SUBK P. SETK P causes the 16-bit (decimal) number P to be placed in COUNT. KOUNT P adds the decimal number P (0-99) to COUNT, and SUBK P subtracts P from it.

Another Apple FELIX number instruction is RND. Execution of RND produces a somewhat randomly selected permutation of the numbers 1-6. These numbers are available to a FELIX program at locations \$656E to \$6573. More details are given about RND and the base routine RANDOM in Section 3.15.

The instructions NUMCKD P, NUMCKH P, and LETCHK P do not manipulate numbers, but they do check them for ASCII code ranges. If P contains the ASCII code for a decimal digit, NUMCKD P resets the FELIX carry flag; otherwise, it sets the carry flag. NUMCKH P and LETCHK P provide the same service for hexadecimal digits and letters, respectively. The purpose of these instructions is to reduce, to some extent, the burden of writing validity checks of user input to interactive FELIX programs.

The 11 core instructions in the control category are: JUMP, JPL, JMI, JNZ, JZ, JNC, JC, CALL, RET, HALT, AND DLAYOP P. All the jump instructions and the CALL instruction are followed by a "Type 9" 16-bit address. The operation of the jump instructions depends upon the FELIX flags in the manner indicated by their mnemonic forms. The subroutine instructions CALL and RET utilize the FELIX stack. The HALT instruction ends a FELIX program and returns control to the Apple Monitor. Execution of DLAYOP P causes the EXEC routine to delay P milliseconds each time it fetches and executes a FELIX instruction. The only additional control instruction added by Apple FELIX is SLOWOP P. SLOWOP causes a delay only when it is executed. The delay can range from a fraction of a second for P equal to 1 up to approximately 40 seconds for P equal to 255.

The only core FELIX instruction not yet mentioned is PRNTOP P. Execution of this instruction prints--on the bottom of the screen--a character string that is stored at location P.

There are many special print routines embedded in the Apple Monitor. Some of these which do not involve argument passing can be accessed by the Apple version of PRNTOP P. This is done by putting a fake address in P. The fake address is actually a pointer to, or a selector of, one of these Monitor routines.

The Apple FELIX operation PRINT2 P,Q can call those Monitor routines called by PRNTOP, as well as three others that are passed on argument by way of the parameter P and the 6502 accumulator. Q is a 1-byte parameter since the fake address pointer or selector only involves the high byte of the "address." Detailed explanations of these operations are

given in Section 3.11. The general design of text output is discussed in the next section.

Three Apple FELIX instructions that often complement each other's operation are: NOTOP, TRCOP, and NTRCOP. NOTOP is the "no operation" operation. TRCOP sets TRCFLG (\$5097), and NTRCOP resets it. These three operations are discussed in Section 3.12. The last minor Apple FELIX operation to mention is BELL P. This simply calls the Monitor BELL routine P times.

The GETSTR instruction accepts a string of keyboard input, echoes it on the screen, stores it in INBUF (\$200), and puts its length in the location LENGTH (\$509D). Section 3.13 discusses GETSTR and its base routine, BLNKRD.

As has already been pointed out, WRITE and ARITH are very special FELIX operations and are discussed extensively throughout the remainder of this report, beginning with Section 2.5.

SOUND P,Q is an Apple FELIX instruction which produces noises. For example, SOUND produces the gunfire in a FELIX program called SPELL and SHOOT. The parameters P and Q contain addresses of a "note duration" table and a "note pitch" table, respectively.

NOTES P,Q is a considerably more sophisticated instruction than SOUND. NOTES plays the tune that is stored in a table, whose address is contained in parameter P. Parameter Q contains a tempo argument. Section 3.17 is devoted to a discussion of the instructions SOUND and NOTES.

## 2.4 Text Output

In Apple FELIX, text output is to the bottom four lines of the screen using the mixed-text and lo-res graphics mode. As Dougherty (9, p. 27) points out, the "preferred method" of text output to the screen is by loading the character to be output into the accumulator and calling the Monitor routine COUT (\$FDED). Using this method, carriage return, line feed, and scrolling within the four-line window are all supported.

In the Apple FELIX environment, a major problem with this approach soon became apparent. Text output via the Monitor is always to Page 1. If Page 2 is the active page, the text output is not visible. The inability to write to Page 2 was not the only problem. As matters stood, every call of the operation SHOW wiped out the text on the screen; that is, extant text expired at the introduction of each new graphics frame. This was considered to be an undesirable feature.

It was eventually concluded that the ideal situation would be one in which text output and graphics output were independent of each other. Thus, the appearance of the text should be independent of the page configuration, and its longevity should be independent of calls to the SHOW routine. On the other hand, the integrity of a full graphics display should be maintained after a temporary interruption from mixed-mode text output.

In the main, Apple FELIX has achieved these goals. The process is quite complex, and there is a slight time penalty paid in the routine SHOW. Of course, the time consumed by the code that actually produces the text is of no great concern since, at such moments, the process

usually has to be slowed down even more to allow the user time to read the text.

At this point, a general description of the process and routines involved when text output is called for by a FELIX program is given. The principal routines involved in text output are PRINT, MXTMDE, CLRTXT, and GRFMDE. Supportive routines are SLOW, SHOW, and MOVTEXT. The operations PRNTOP and SLOWOP call PRINT and SLOW, respectively. Two flags are used: TXTFLG and PG1FLG. There are three buffers specifically introduced to handle text output: TXTBUF, SAVBUF, and BCKSAV.

The capsule description of the process is as follows. When PRNTOP is executed, it passes the address of the string to PRINT, and PRINT takes control at this point. PRINT immediately tests TXTFLG. If TXTFLG is zero, indicating the full-graphics mode of operation, PRINT calls the routine pair MXTMDE-CLRTXT. (CLRTXT is a subroutine involved with Monitor calls and is always called by MXTMDE.) MXTMDE initializes the mixed mode of operation and returns to PRINT. PRINT proceeds to print the string and returns to the EXEC routine. When it is desired to return to the full-graphics mode of operation, the FELIX operation GRFMDE is executed.

SHOW monitors the whole process; it checks flags and updates buffers when necessary. In most cases, the FELIX operation SLOWOP (with a rate parameter) is executed to provide reading time for the user. The routine MOVTEXT has the job of moving information in and out of the buffers. "MOVTEXT" is somewhat of a misnomer, since it also moves pixel information. The details concerning the process of text output and the routines involved are given in Sections 3.8 through 3.13 in Chapter 3.

## 2.5 The WRITE and ARITH FELIX Operations

WRITE and ARITH are sometimes designated SLATE and CIPHER, respectively. WRITE and ARITH are FELIX operations in one important respect: they have FELIX opcodes. Unlike the other FELIX operations which are single instructions in the FELIX language, WRITE and ARITH are substantial programs that perform a variety of functions. Perhaps one way to characterize them is to say that, given a greater abundance of FELIX instructions, they might have been written in the FELIX language. Thus, one purpose of programs written in this format is to serve as a sort of "pilot plant" environment to test out new functions and concepts.

Notwithstanding the developmental nature of ARITH and WRITE, their output is (it is assumed) of some interest. Also, at present, they count themselves in the company of a small number of programs that demonstrate the features of the FELIX system. For these reasons, it has been desired to keep WRITE and ARITH in the FELIX family. As a future project, some of their principal functions might be separated out and given their own FELIX opcodes. ARITH would be a particularly likely candidate for such a procedure.

The routine WRITE served as the motivation for the incorporation of some new elements in Apple FELIX. These are permanent objects that are in addition to the original 16 objects of FELIX/8080. Aside from their permanence--they are loaded into the Apple's memory as part of the FELIX system--they differ from the first 16 objects in that their lengths, as stored data, are not fixed at 256 bytes. Each object is only as long as is necessary to record its pixel positions.

At present, the objects included in Apple FELIX are the letters of the alphabet, the ten digits, and, in fact, all but eight of the characters on the Apple keyboard. These "character objects" are formed on a 5 x 7 pixel matrix and are actually patterned after the character set of a Centronics printer. Although it is not an element in the Apple character set, the division symbol is included as one of the objects.

The WRITE operation functions as follows. Using the cursor as a prompt, WRITE accepts a character string from the keyboard and echoes it on the text part of a mixed-text and graphics screen. The graphics section of the screen is divided into five equal horizontal strips that serve as five lines for the graphics character objects. Each graphics line can accommodate six character objects. After the carriage return is entered, the character string, in the form of the character objects, is put on the graphics screen. The character objects are placed on the bottom graphics line, with scrolling of the lines as necessary. WRITE recognizes "Control-D" as the proper keyboard input for the division symbol.

In addition to serving as a stimulus for incorporating the character objects into Apple FELIX, the WRITE routine also first drew attention to an inability in Apple FELIX to put a cursor on Page 2. As detailed in Chapter 3, this problem was worked in the routine WRITE; with the ARITH routine, a more direct approach was taken, as detailed below.

The ARITH routine accepts an arithmetic problem involving positive numbers from the keyboard. This can be a problem in addition, subtraction, multiplication, or division. ARITH calculates the answer and

stores it for subsequent processing. At this point, the partner of ARITH, designated FELIX-ARITH, takes over. FELIX-ARITH is the first substantial Apple FELIX program written. Its first job is to present the problem on the screen using the character objects mentioned above. After a response from the keyboard, it puts the answer on the screen. With a second response from the user, ARITH is returned to for the next problem. The user can terminate the process with an input of "Control-E" to ARITH.

ARITH is a project of considerable magnitude. In addition to the FELIX program FELIX-ARITH, ARITH code is found in three source files: ARITH, ARITH2, and DECADD. Two Apple FELIX operations have already resulted from the development of ARITH. They are the RND operation and the GETSTR operation. The noteworthy features of GETSTR is its ability--by way of the base routine BLNKRD--to prompt for with a cursor and accept user input on Page 2.

Section 3.14 is devoted to the WRITE operation and the permanent objects. Section 3.15 discusses the 6502 routines of ARITH. The operation of FELIX-ARITH is explained in Section 3.16, and a listing of FELIX-ARITH is given in the Appendix.

## CHAPTER 3

### IMPLEMENTATION OF APPLE FELIX

The first two sections of this chapter deal with the memory map of the Apple display area and the algorithm used to "decode" it. In Sections 3.3 and 3.4, the storage and use of the 2-byte sine and cosine values are discussed.

With particular emphasis on Types 4 and 7, the description of the parameters is completed with an explanation of their formats in Section 3.5. Then, in the following two sections, the role of SWEET16 in implementing the registers is discussed.

Sections 3.8 to 3.13 are devoted to all those routines involved in any way with text output in the Apple FELIX system. The routines of the WRITE and ARITH operations are explained in Sections 3.14 to 3.16. The chapter concludes with a discussion of the SOUND and NOTES operations in Section 3.17.

#### 3.1 Memory Map of the Display Area

In this section, the memory map for the lo-res graphics mode is described in detail, in terms of Page 1 addresses. The description could serve just as well for Page 2 if 400 "hex" were added to every address mentioned. When the full graphics mode is operative, the screen consists of 1920 pixels in an array 40 pixels wide and 48 pixels high. For the purposes of this description, the origin (0,0) is regarded as being in the upper left-hand corner of the screen. (The origin for FELIX is in the lower left-hand corner.)

A byte in the display area controls two pixels. If  $j$  is an even number, and the low nybble of a certain byte controls the pixel with coordinates  $(i,j)$ , then the high nybble controls the pixel with coordinates  $(i,j+1)$ . Thus, memory location 1024 (\$400), the first byte of Page 1, controls screen location  $(0,0)$  and  $(0,1)$ .

Within a pixel row, consecutive memory locations control consecutive pixels. Designate such a group of 40 consecutive memory locations a "memory row." In addition, call the two pixel rows controlled by one memory row a "double pixel row."

In order to understand the memory map, the screen may be thought of as being trisected. Thus, there are 16 pixel rows in each third of the screen corresponding to eight memory rows. The memory byte that controls the final two pixels in a double pixel row is not consecutive with the byte that controls the first two pixels in the next double pixel row; this is what gives rise to the trisection.

Specifically, the pattern is as follows. The first memory row (1024-1063) controls pixel rows 0 and 1. The second memory row controls pixel rows 16 and 17. These two rows constitute the first double pixel row of the second third of the screen. The first double pixel rows of the last third of the screen (pixel rows 32 and 33) are controlled by the third memory row. At this point, there is an anomaly: the next eight bytes are not used for display purposes. If the eight unused bytes at the end are included, exactly 128 bytes have been used for one traverse down the screen.

The fourth memory row, starting at location 1152, controls the second double pixel row (pixel rows 2 and 3) in the top third of the

screen, and another 128-byte cycle has begun. A total of eight such cycles completes the memory mapping of the lo-res display area.

As a result of this mapping, the following relationships hold:

(1) pixels in even-numbered rows are controlled by low nybbles of display area bytes; high nybbles control pixels in odd rows, (2) within a given third of the screen, addresses of controlling bytes for first pixels of two adjacent double pixel rows differ by 128, and (3) the address of the byte that controls the first pixel in the last third of the screen is 40 greater than the address of its counterpart in the second third, which, in turn, is 40 greater than the address of its counterpart in the top third of the screen. These are the relationships used in writing the routine FIND which transforms a coordinate location into a memory location.

### 3.2 The FIND Algorithm

FIND is the routine that receives an x,y pair of coordinates as input and then calculates the display memory in ONBUF corresponding to this pair. It outputs this address to the calling routine--either PUT or GET. Specifically, it stores this address in the pointer ADDR,ADDR+1. FIND also returns a 1 in the flag, ODDEVN, if the pixel row is odd-numbered, and a 0 if the row is even-numbered. ODDEVN flags PUT (or GET) that it should write to (or read) the high nybble if it is set, or the low nybble if reset.

The algorithm that converts a coordinate pair into a display memory location is now given. Each step is followed by a parenthetical remark justifying or explaining that step.

- (i) Put the address of ONBUF into ADDR,ADDR+1. (ADDR, ADDR+1 contain the address of the byte associated with the first pixel in the top third of the screen.)
- (ii) Subtract the FELIX y coordinate from 47. (The FELIX origin is located at the lower left-hand corner of the screen. The user has converted to Apple coordinates.)
- (iii) If the y coordinate is greater than 31, add 80 to ADDR,ADDR+1 and go to step (v). (If y is greater than 31, the user is in the bottom third of the screen. The byte associated with its first pixel has an address 80 larger than the address found in step (i)).
- (iv) If y is greater than 15, add 40 to ADDR,ADDR+1. (If the condition holds, the user is in the middle third of the screen, and its "base byte" is 40 beyond the byte of step (i)).
- (v) "And" the y value with \$0F. (The result is a modulo 16 value giving the number of pixel rows into the applicable third of the screen.)
- (vi) Divide the result of step (v) by two. (There are two pixel rows for each "row" of memory locations.)
- (vii) Multiply the result of step (vi) by 128 and add the product to ADDR,ADDR+1. (Within any given

third of the screen, there is a 128-byte difference between the first bytes of adjacent "memory rows.")

(viii) Add the value of the x coordinate to ADDR, ADDR+1.

(Pixels adjacent in the same row correspond to adjacent memory locations.)

(ix) Return to caller.

### 3.3 The SIN/COS Table

Since, in the ROTATE routine, the object is rotated about its own center of gravity, the coordinates of the center of gravity can be momentarily designated by (XORG, YORG). If (X,Y) mean the coordinates of a pixel of the object with respect to the point (XORG, YORG) taken as the origin, then the transformation for the new X coordinate, after rotation, with respect to (0,0) as the origin, is given by:

$$X' = XORG + X\cos A + Y\sin A;$$

and the corresponding transformation for Y is:

$$Y' = YORG + Y\cos A - X\sin A.$$

Since sine and cosine function routines are not available, and since four multiplications per pixel would, in any case, require an excessive amount of time, the values  $X\cos A$ ,  $X\sin A$ ,  $Y\cos A$ , and  $Y\sin A$  are retrieved directly from the SIN/COS table. The variable Y can take any integer value between 0 and 47, inclusive. The variable A stands for any one of the 23 strictly non-negative multiples of 15 degrees between 0 and 360. Thus, taking both the sine and cosine functions into account,

if each possible value were an entry in the table, the table would have  $48 \times 23 \times 2$ , or 2208, entries. Since, as detailed in another part of this report, a 2-byte entry was used, a table constructed in this way would consume more than 4K bytes. Knowing that various sine and cosine values are shared by various angles, it was decided to investigate the possibility of condensing the table. It turned out that the cosine half of the table can be eliminated, and even allowing for separate entries for the negative values, only 13 of the original 23 entries are needed in each row of the table. Thus, the table has been reduced to nearly 1/4 of its original size. Typically, one entry will serve for three others. For example,

$$\sin 30^\circ = \sin 150^\circ = \cos 60^\circ = \cos 300^\circ.$$

There are 48 rows corresponding to the 48 values of Z which range from 0 to 47, inclusive. Each row has the following non-negative entries: Zsin0, Zsin15, Zsin30, Zsin45, Zsin60, Zsin75, and Zsin90. The negative entries are: Zsin195, Zsin210, Zsin225, Zsin240, Zsin255, and Zsin270.

With the help of two auxiliary tables, SINIX and COSIX, the code required by ROTATE and TRIG to access the proper values is short and straightforward. Upon entry into ROTATE, the orientation number (minus one) is used as an index into both SINIX and COSIX to get two more indices into the main table--one for the sine value and one for the cosine value. These are stored in SINPTR and COSPTR, respectively, for use later by the routine TRIG. For example, if the orientation number is eight, corresponding to a rotation of 120 degrees, a four is found as the seventh entry in SINIX, and an eight is found as the seventh

entry in COSIX. This corresponds to the facts that  $\sin 120 = \sin 60$  and  $\cos 120 = \sin 210$ , and the  $Z\sin 60$  and  $Z\sin 210$  values are in the fourth and eighth positions in each row, respectively.

Before leaving this topic, it should be pointed out that if there were code to form the negatives of values in the table, we could get by with only the seven non-negative entries in each row. With the arrangement described above, the only change that would be necessary is to set bit 7 of the SINIX or COSIX values if a negative quantity were called for. Thus, returning to the previous example of a rotation of 120 degrees, the value found in COSIX would be 82 in hexadecimal because  $\cos 120 = -\sin 30$  and the  $Z\sin 30$  value is the second (counting from 0) entry into each row. Thus, the 2 of 82 would point into the table, and the 8 of 82 would remind the user later to form the negative of the value.

### 3.4 2-Byte Sine and Cosine Values

In the routine ROTATE, the four expressions (or their additive inverses),  $X\cos A$ ,  $X\sin A$ ,  $Y\cos A$ , and  $Y\sin A$ , are obtained directly from the SIN/COS table. The expression  $X\cos A + Y\sin A$  appears in the transformation giving the new X coordinate for a pixel after rotation;  $Y\cos A - X\sin A$  appears in the Y transformation.

If the table entries are 1-byte values that are the result of rounding off the actual values to integers before insertion into the table, it is clear that adding or subtracting two such values together will occasionally yield an incorrect result. In a test performed on 150 summations of randomly generated values, the results were incorrect in approximately 15% of the cases. Of course, some of these were borderline cases in which no integer value would be a very good approximation.

While on the subject of image distortion under rotation, it might be mentioned that, undoubtedly, the biggest obstacle to image integrity is the fact that, as Wadsworth points out (12, p. 35), the Apple 10-res pixel has a "horizontal side to vertical side" ratio of three to two. This is vividly apparent in a clock demonstration programmed by the author. It is rather disconcerting to watch the minute hand grow 50% in length at the same time it is shrinking to one-half its previous width as it progresses from the 12 to the 3:00 position.

It is, nonetheless, true that early round-off can, and does, give values that are markedly incorrect. For example, if  $A = 120^\circ$ ,  $X = -11$ , and  $Y = 11$ , then calculation of  $X\cos A + Y\sin A$ , using 1-byte table values, yields a result of 16. Carried to two decimal places, the correct value is 15.03. Because of such considerations, it was decided to use 2-byte values in the SIN/COS table, with the low byte containing the fractional part of the value.

In the BASIC program that calculated and stored the values in the SIN/COS table, the integer part was POKed into the high byte. Then the decimal fraction was multiplied by 256, and the integer part of this result was POKed into the low byte. If the value was negative, the above procedure was performed on its absolute value, and then the two's complement was formed and inserted into the table. The above procedure was simple and straightforward. Also, it was not anticipated that the code for the necessary additions, subtractions, and complementations in ROTATE would be unduly expanded.

The important unresolved question, as it was foreseen, involved the amount of "testing" code necessary in ROTATE to utilize the information

contained in the fractional part of the result once all operations had been performed. Dozens of different cases were visualized involving such factors as the type of operations, the signs of the operands, the relative size of their absolute values, etc., that might have to be tested out before evaluating the fractional part of the result. As one conversant with the intricacies of two's complement arithmetic might have anticipated, it was found, after extensive experimentation, that the problem resolved itself into one very simple rule: if, after all operations are performed, the fractional byte of the result has a value of 128 or greater, then increment the integer byte of the result.

Returning to the previous example, the numbers  $-11 \cos 120$  and  $11 \sin 120$ , using the 2-byte formulation, would be 5(high), 128(low) and 9(high), 136(low), respectively. Their sum would be 15(high), 9(low); clearly indicating a round-off of 15 and not the previously arrived at value of 16.

### 3.5 Details of the Parameter Formats

Before reading this section on the parameter formats, the reader may wish to review Section 2.2, which describes the eight parameter types.

Types 1, 2, 5, and 6 are the parameters with the simplest param-headers. These param-headers (the first byte of the parameter) contain only the type number in bits 5 to 7; bits 0 to 4 are not used. The second byte of Type 1 contains the 8-bit constant value. The second and third bytes of the Type 5 parameter contain the 16-bit constant. The second and third bytes of the Type 2 and 6 parameters contain a 16-bit address. The 16-bit addresses and the 16-bit constants are in low-high order.

The remaining parameter types are 1-byte parameters. Of these, the ones with the simplest formats are the Type 0 and the Type 3 parameters. Type 0 contains the 5-bit constant in bits 0 to 4. Thus, the constant for a Type 0 parameter is restricted to a value between 0 and 31. Bits 0 to 4 of the Type 3 parameter contain the appropriate port number.

The param-headers of Types 4 and 7 have three fields. In addition to the type number field in bits 5 to 7, there is an index field in bits 3 and 4 and a register number field in bits 0 to 2. The function of the four index numbers, 0, 1, 2, and 3, needs to be explained. Index number 3 specifies the system registers that are not implemented in Apple FELIX; when it is encountered, an error message is printed, and register processing terminates.

With the one exception explained below, the index numbers 0, 1, and 2 have the same effect on Type 4 parameters and Type 7 parameters. They all affect the incrementing or decrementing of the registers before or after data is moved from or to the applicable register or array. If the index number is zero, the register is not incremented or decremented. The register value is incremented after the data transfer takes place if the index number is one. The register value is decremented prior to the data transfer if the index number is two. The exception alluded to above is that in processing a Type 7 parameter with an index number of two and with an OUTPAR call, there is no point in decrementing the value in the register and then writing to the register; and, indeed, the decrement is not applied to the register in this case.

There is one other distinction recognized by INPAR and OUTPAR in processing Type 4 and Type 7 parameters: the processing of registers

0-3 is different from the processing of registers 4-7. When appropriate--as detailed above--the lowered-numbered registers are incremented or decremented by one; registers 4-7 are changed by two. Also, the arrays indexed by registers 0-3 contain byte length data, and the arrays indexed by registers 4-7 contain words. Accordingly, INPAR and OUTPAR move bytes in the former case and words in the latter case.

### 3.6 The SWEET16 Pseudo-Machine

Stephen Wozniak has implemented a pseudo-machine, SWEET16, resident in the Apple II ROM memory, which is based around 16 16-bit registers. SWEET16 is not resident in the ROM of the Apple II Plus computer. If Apple FELIX is run on the Apple II Plus computer, SWEET16, as well as the Floating Point routines, which are used by the NOTES op-routine, must be supplied to the system. These routines are available, for instance, on Apple Computer's Integer BASIC Firmware Card.

The registers, called R0 to R15, are actually implemented as the first 32 memory locations of Page 0. Some of these registers are pre-empted by SWEET16 itself. R0 is the SWEET16 accumulator; R15 is the program counter; R14 is a status register; R13 holds compare instruction results. If SWEET16 subroutines are used, R12 serves as the subroutine return stack pointer. All other SWEET16 registers are available to the user, that is, to the writer of 6502 code on the Apple II. In the present case, the SWEET16 registers R1 to R8 correspond to FELIX registers 0 to 7, respectively.

At this point, only as much will be said about the operation of the SWEET16 machine as is necessary to enable the reader to understand

how it is used in conjunction with INPAR and OUTPAR to implement the FELIX registers. More details can be found in the two references, Wozniak (10) and Vile (11).

The SWEET16 instructions fall into two broad categories: the register instructions and the nonregister instructions. Except for the SET instruction, the register instructions require only one byte. The high nybble of the byte contains the opcode of the instruction, and the low nybble specifies the register. There are 15 register instructions with opcodes ranging from 1 to F. As an example, the opcode for the LOAD instruction is 3; thus, 38 signifies that the SWEET16 accumulator, R0, should be loaded with the contents of R8. The 13 nonregister instructions have opcodes ranging from 00 to 0C. A few of these are 1-byte instructions, but most are 6502 style relative branches and require two bytes.

Table 1 lists the SWEET16 instructions that are used in Apple FELIX. The registers are used as data elements or as pointers to data in memory. In the "Arguments" column of the table, the symbol @Rn signifies the latter usage. In the "Effect" section of the table, parentheses around the register symbol signify this usage. We might note that there is disagreement between Table 1 and information given in Reference 11 (Vile) for the instructions LD @Rn, ST @Rn, LDD @Rn, STP @Rn, and POPD @Rn.

Before explaining how the SWEET16 machine is utilized by a 6502 language program, the following observation is made. In the 6502 language, as part of the execution of a JSR (Jump to Subroutine) instruction, the address of the last byte of the JSR instruction is placed on the

TABLE 1  
SWEET16 OPERATIONS USED IN APPLE FELIX

| Opcode | Operation |             | Effect                                       |
|--------|-----------|-------------|----------------------------------------------|
| 1n     | SET       | Rn Constant | Rn = Constant                                |
| 2n     | LD        | Rn          | R0 = Rn                                      |
| 3n     | ST        | Rn          | Rn = R0                                      |
| 4n     | LD        | @Rn         | ROH = 0; ROL = (Rn); Rn = Rn+1               |
| 5n     | ST        | @Rn         | (Rn) = ROL; Rn = Rn+1                        |
| 6n     | LDD       | @Rn         | ROL = (Rn); Rn = Rn+1; ROH = (Rn); Rn = Rn+1 |
| 7n     | STD       | @Rn         | (Rn) = ROL; Rn = Rn+1; (Rn) = ROH; Rn = Rn+1 |
| 8n     | POP       | @Rn         | Rn = Rn-1; ROL = (Rn); ROH = 0               |
| 9n     | STP       | @Rn         | Rn = Rn-1; (Rn) = ROL                        |
| Cn     | POPD      | @Rn         | Rn = Rn-1; ROH = (Rn); Rn = Rn-1; ROL = (Rn) |
| Dn     | CPR       | Rn          | R13 = R0-Rn; R14H = 13*2+Carry               |
| En     | INR       | Rn          | Rn = Rn+1                                    |
| Fn     | DCR       | Rn          | Rn = Rn-1                                    |
| 00     | RTN       |             | Return to 6502 mode                          |
| 01     | BR        | Address     | Branch always                                |
| 06     | BZ        | Address     | Branch if zero                               |
| 07     | BNZ       | Address     | Branch if not zero                           |

6502 stack. If, on occasion, this address is referred to as the "return address," it is with the understanding that it must be increased by one in order to give it the exact return address.

In a 6502 language program, whenever its services are required, the SWEET16 machine is "turned on" by the execution of a JSR SWEET16 instruction. After saving the 6502 registers, SWEET16 then pops the return address of the JSR SWEET16 call from the 6502 stack. This address is stored in R15, the SWEET16 program counter. Thus, the PC is pointing at the code immediately following the JSR SWEET16 call in the 6502 program. This code is written in the SWEET16 language. With the aid of the indirect addressing capability of R15, SWEET16 proceeds to fetch and execute this code.

When a SWEET16 return instruction, RTN, is encountered, the 6502 registers are first restored, and then SWEET16 executes the 6502 indirect jump instruction, JMP (R15). Since, as the SWEET16 PC, R15 has been continuously updated, the result is that the next instruction executed is the first 6502 instruction following the RTN. The SWEET16 machine has been turned off.

### 3.7 SWEET16 and the FELIX Registers

All of the SWEET16 code used to implement the FELIX registers and the register-indexed arrays can be found in the base routine REGS starting at location \$5076. REGS processes the Type 4 and Type 7 parameters for both INPAR and OUTPAR. On their calls to REGS, which are actually jump instructions, both INPAR and OUTPAR enter REGS at the top.

INPAR and OUTPAR pass REGS the information it needs by way of the variables SAVE5 (\$509C) and TYPE (\$509B). SAVE5 contains bits 0-4 of the present param-header. Thus, in the case of Type 4 or Type 7 parameters, SAVE5 contains the register number in bits 0-2 and the index number in bits 3-4. TYPE, of course, informs REGS whether the parameter type is 4 or 7; but it also informs REGS whether this is a call from INPAR or OUTPAR. Specifically, TYPE contains one of the four numbers 04, 07, 84, or 87. INPAR is responsible for the first two, and OUTPAR is responsible for the last two.

If the call to REGS is from OUTPAR--in addition to the information in SAVE5 and TYPE--the data to be stored must be passed to REGS. At the very top of OUTPAR, a JSR SAVE instruction is executed in anticipation of a Type 4 or Type 7 parameter. SAVE is the Monitor routine that saves the register contents, and OUTPAR receives the data to be stored in the X and Y registers. Once REGS has determined that it is being called by OUTPAR, it calls the Monitor RESTORE routine to retrieve the X and Y register contents; it stores them in the variable VALUE,VALUE+1 in low-high order.

The routine REGS is divided into six sections: the beginning of REGS, ERRRTN, ITYPE4, ITYPE7, OTYPE4, and OTYPE7. The beginning of REGS supplies the index number to IDXNO, the register number to REGNO, and sets REGFLG if the register number is four or larger. It jumps to ERRRTN if the index number is three; otherwise, it jumps to one of the "type" sections. Each of these "type" sections processes its appropriate parameter type, the first two for INPAR and the last two for OUTPAR.

ITYPE4, ITYPE7, OTYPE4, and OTYPE7 each have their own segment of SWEET16 code. When one of these sections is entered, the user could

be working with any one of the eight Apple FELIX registers. On the other hand, in a SWEET16 register instruction, the register number must be specified. As a consequence of this situation, each of the type sections of REGS begins with a code modification segment. As originally written, every register instruction uses the dummy value seven as the register number. When a type section is entered, every appearance of seven as a register number is changed to the value stored in REGNO.

The mechanics of the code modification are quite simple. For each type of register instruction that appears in the SWEET16 routine in question and with its opcode in the high nybble of the accumulator, an ORA REGNO instruction is executed. The resulting correct code is then stored in each line--all with labels, of course--in the SWEET16 routine in which that instruction appears.

The following information may be of assistance in reading REGS code. The SWEET16 code of the ITYPE4 and the OTYPE4 sections is, in both cases, divided into two main parts: a part to process byte data and a part to process word data. Each of these parts, in turn, processes the index information. The ITYPE7 SWEET16 code, on the other hand, is divided into three main parts according to the value of the index number. Within the last two of these, processing is determined by the value of REGFLG. Because the decrement is not applied in the OTYPE7 section, its SWEET16 code is not so markedly subdivided.

### 3.8 The Details of Text Output

In Chapter 2, a very general description was given of the method by which text is written to the screen in Apple FELIX. In this section, a detailed explanation of this process is begun.

This first section concerns those routines which might be designated the "main characters" in the production of text output: PRINT, MXTMDE, CLRTXT, and GRFMDE. The support routines SLOW and MOVTEXT are discussed in the next section. SHOW will be honored with a section of its own. PRINT has a secondary method of text output which is described in Section 3.11. The Apple FELIX trace mode is explained in Section 3.12. Section 3.13 is devoted to the GETSTR and BLNKRD routines.

The discussion is begun with the PRINT routine. For the moment, assume that the mode of operation is mixed-text and graphics when PRINT is called by the FELIX operation PRNTOP. This fact is flagged to PRINT by TXTFLG being set. In this case, PRINT bypasses the call to MXTMDE and proceeds directly to printing the string. The first action taken by PRINT is to set PG1FLG (Page 1 flag). All text output to the screen is done through the Monitor routine COUT, and COUT always places its output on Page 1. Thus, whichever page is active, any text output will change the contents of Page 1. PG1FLG flags this condition to SHOW, which, as shall be seen subsequently, takes appropriate action.

At this point, PRNTOP, through INPAR, has already picked up from the FELIX program the address of the character string and stored it in the zero page pointer P RTPTR, P RTPTR+1. PRINT checks P RTPTR+1 for a value of \$F0 or larger. Such a value is, of course, a dummy address,

but it is the method used to indicate a call to the secondary mode of operation of PRINT. In this case, it is assumed the address is a valid one.

PRINT proceeds to output the characters through COUT, all the while looking for line feed, carriage return, or "stop" code. When the printing is complete, the bottom four lines of Page 1 are copied onto TXTBUF by a call to MOVTEXT. The idea behind TXTBUF is that it should always contain an updated version of the text presently on the screen. Before returning, PRINT checks whether Page 1 is ONBUF (and, thus, Page 2 is active or SHOBUF). If Page 2 is active, by another call to MOVTEXT, the bottom four lines of Page 1 are copied to Page 2. In either case, the new text--perhaps along with previous text--is on the screen.

The routines MXTMDE and CLRTXT are now discussed. Recall that if, on entry into PRINT, TXTFLG equals zero, indicating the full-graphics mode of operation, then PRINT immediately calls MXTMDE. MXTMDE can be called by 6502 programs, and it is also a FELIX operation with opcode \$29. MXTMDE is always called to effect the transition from full-graphics to mixed-mode operation.

MXTMDE sets TXTFLG and also throws the mixed-mode switch (location \$C053). It then enters the section of code labeled ERASE. At this point, CLRTXT is called. CLRTXT calls a Monitor routine starting at location \$FC5A that clears the bottom four lines of Page 1. It then calls another Monitor routine at \$FB51 which implements a vertical tab to line 23--the bottom line. The top text line is set at line 20, after which CLRTXT returns. MXTMDE sets PG1FLG and then, by a call to MOVTEXT, it clears TXTBUF with a Page 1 to TXTBUF move. Finally, if Page 2 is

the active page, the cleared portion of Page 1 is moved to Page 2, again by a call to MOVTEXT.

MXTMDE can also be called when TXTFLG is already set. In this case, it immediately jumps to the ERASE section; in other words, MXTMDE can be called simply to "clear the slate."

The routine GRFMDE is the complement of the routine MXTMDE. Just as the mixed mode of operation is always initialized by a call to MXTMDE, so it is always terminated by a call to GRFMDE. Like MXTMDE, BRFMDE can be implemented as a FELIX instruction (opcode = \$28), or it can be called by 6502 routines. The operation of GRFMDE is quite simple. It hits the full-graphics switch (\$C052), resets TXTFLG, and, by a call to MOVTEXT, moves SAVBUF to the active page. SAVBUF is a buffer, prepared by SHOW, containing the bottom eight pixel rows of the active page.

### 3.9 The Routines SLOW and MOVTEXT

The routine SLOW is often called in conjunction with text output to allow the user "reading time." SLOW can be called by other base routines by passing a rate parameter to it in the accumulator, or it can be called by the FELIX operation SLOWOP. In the latter case, the code of the routine SLOWOP calls INPAR and passes the rate to SLOW, again in the accumulator.

There are two other routines that are always involved in any call of SLOW. They are the FELIX base routine DELAY and the Monitor routine WAIT (\$FCAB). In the explanation of the operation of the routine SLOW, it would be best, at this point, to go to the low end of the hierarchy of routines, that is, to WAIT, and then work back up to SLOW.

WAIT gives variable time delays depending on the value passed to it in the accumulator. The delay ranges from approximately 30 microseconds, with an accumulator value of one, to one-sixth of a second, with an accumulator value of 255. According to Dougherty (9, p. 42), if WAIT is called with a value of 17, then a delay of approximately one millisecond is achieved. The one-millisecond delay is noteworthy because both FELIX/8080 and Apple FELIX have an operation, DLAYOP P, which causes a delay of P milliseconds. Thus, if the accumulator contains 17 (\$11), the delay of P milliseconds can be achieved by calling WAIT in a loop that is executed P times. This is precisely what is done in the routine DELAY. It might be pointed out that DELAY is not called by the FELIX routine DLAYOP. DLAYOP simply puts the value P into the location RATE. EXEC is the routine that calls DELAY. It does so, as long as RATE has a nonzero value, each time that it executes a new FELIX instruction.

The routine DELAY is the one which attention must be centered on in order to complete the explanation of the operation of SLOW. Since code modification is involved and DELAY is a short routine, it is listed below:

```

1730 * -----
1740 * *      DELAY      *
1750 * -----
1760 *
1770 * WITH (A) = #$11, ONE CALL TO WAIT
1780 * CAUSES ABOUT 1-MILLISECOND DELAY.
1790 * "RATE" HOLDS NUMBER OF CALLS.
1800 RATE .EQ $5095
51B5- AE 95 50 1810 DELAY LDX RATE      # OF CALLS
51B8- A9 11    1820 DLAYLP LDA #$11
51BA- 20 A8 FC 1830          JSR $FCA8  MONITOR "WAIT"
51BD- CA      1840          DEX
51BE- D0 F8    1850          BNE DLAYLP
51C0- 60      1860          RTS

```

SLOW proceeds in the following manner. It stores the value passed to it in the accumulator into memory location \$51B9. As the code above indicates, this code modification has the effect of passing this value onto WAIT to be used by it as a rate parameter. Next, the value in RATE is saved, and \$FF is put in its place. Thus, the loop in DELAY will be executed 255 times. SLOW then calls DELAY. After DELAY returns, SLOW restores the value \$11 at location \$51B9; it restores the original RATE value and then returns.

In summary, DELAY, as written, causes delays in the millisecond range--from one to 255 milliseconds. This variation is dependent on the variation of the value in the location RATE. This delay is implemented throughout a FELIX program, that is, every time EXEC executes a new FELIX instruction. SLOW, on the other hand, can cause delays ranging from a fraction of a second up to approximately 40 seconds, depending on the value--not in RATE, which is constant at 255--but on the code-modified value stored at location \$51B9. This delay is implemented only at the time SLOW is called.

MOVTEXT is called with one of the numbers zero through nine in the X register. Each one of these numbers signifies a different move of four rows--each row made up to 40 consecutive memory locations. It is stressed that the memory locations are not consecutive across row boundaries. The scrambled mapping of the Apple display area is responsible for this fact; it is also the main reason that MOVTEXT is table-driven.

Each move requires eight different addresses: one address for each of the four rows of the "TO" area and one address for each of the four rows of the "FROM" area. All of these addresses are stored in the

table ADRTAB. The addresses for the first move (X=0) occupy the first 16 bytes of ADRTAB, and each successive move occupies the next successive 16 bytes of ADRTAB with its addresses.

There is another table in MOVTEXT: a "pointer table," PTRTAB. For each of the ten moves, there is a 1-byte entry giving the proper pointer or index value into ADRTAB for that move. The pointer into PTRTAB, in turn, is the value in the X register on entry into MOVTEXT.

The two lines of code in MOVTEXT that perform the actual moves are given below:

```

2390 FROM   LDA $FFFF,Y   DUMMY ADDRESS
2400 TO     STA $FFFF, Y   DUMMY ADDRESS

```

MOVTEXT first modifies the code at lines 2390 and 2400 to make the dummy addresses legitimate, getting the legitimate addresses from ADRTAB. The Y register is loaded with 39, and the code above is executed 40 times with, of course, a decrement of Y after each execution. This moves one row. The process is repeated three more times, and the move is complete.

### 3.10 The Role of SHOW in Text Output

For convenience, the description of the operation of SHOW as it pertains to the production of graphics output is repeated below. The code involved with text output has been inserted into this original form of SHOW at strategic points; it expands but does not basically alter the original algorithm.

SHOW checks ONLIST (location \$100) to see if any objects are listed there. If so, and if rotation is not indicated, it paints these objects on ONBUF by utilizing calls to the routine PUT. Whenever rotation is indicated for an object, SHOW calls ROTATE to paint that object. Once the objects are painted on ONBUF, SHOW reverses the roles of Pages 1 and 2 as regards SHOBUFF and ONBUF, and the new scene appears on the screen. Before returning, SHOW moves BAKBUF onto the new ONBUF. This is the first step in the preparation of the next scene.

The details of the text processing code in SHOW are now presented. At the very top of SHOW, PG1FLG and ONBUF are tested. If PG1FLG is set, and if Page 1 is ONBUF, then the buffer BCKSAV is moved to Page 1. Under these conditions, Page 1 has been altered because of text output intended for Page 2. This graphics buffer move restores the integrity of Page 1 as ONBUF. It is noted at this point that all the buffer moves now being described involve the last eight rows of pixels, and in the case of a graphics buffer move, or the bottom four rows of text, in the case of a text buffer move.

Whichever page is ONBUF, after the objects have been painted on it, there is a graphics buffer move from ONBUF to SAVBUF. At this point, ONBUF contains only graphics. ONBUF is about to become the active page. If, subsequent to this execution of SHOW, the FELIX operation GRFMDE is executed, SAVBUF will be moved onto the active page, and the full-graphics scene that is presently on ONBUF will appear on the screen.

Just prior to the swap between ONBUF and SHOBUFF (the active page), TXTFLG is tested. If TXTFLG is set, then the mixed-text and graphics mode of operation is in effect, and text is presently on the screen.

TXTBUF, which contains a copy of this text, is moved to ONBUF, and only then is the swap made. Thus, the SHOW operation is transparent to the user as far as text output is concerned.

After the swap and after the BAKBUF to ONBUF move, PG1FLG is reset. Page 1, whether it is SHOBUF or ONBUF, is--at this precise moment--unblemished by its duties as the intermediary between Page 2 and the print routines. After the swap, the page that is presently ONBUF will be ONBUF when SHOW is next executed. Therefore, ONBUF is again tested, and, if it is Page 1, a BAKBUF to BCKSAV move is made in anticipation that PG1FLG might be set on the next entry into SHOW.

If Page 2 is ONBUF, the move is not made, and SHOW returns. Otherwise, SHOW continues with a "post swap" test of TXTFLG. If it is set, a TXTBUF to Page 1 move is executed, and PG1FLG is set. This is done in anticipation of a subsequent print operation that adds to the present text output and does not first clear the text area. In this case, Page 1, even if it is ONBUF, must have a copy of the text presently on the screen.

### 3.11 Alternative Mode of Operation for PRINT

The role of the PRINT routine in printing a stored character string to the screen has been described in Section 3.8. Recall that the address where the string is stored is--prior to the call to PRINT--placed in the zero page pointer P RTPTR, P RTPTR+1 by the calling routine. If the high byte of that address is a number from \$F0 to \$FB, then PRINT is being called, not to output a stored line of text, but to call one of a dozen Monitor print routines. For instance, if \$F5 is passed to PRINT

in PRTPTR+1, then it will call the PRNTAZ Monitor routine. This routine prints a 2-byte hexadecimal number contained in the accumulator and the X register. The 12 routines that PRINT can call are listed in the table MONTAB in the Print source program.

Aside from the convenience of having these routines accessible in one location in Apple FELIX, the primary reason for including them in the PRINT routine is that they can, in this way, be routinely incorporated into the rather elaborate text output procedure that has been described in the previous sections. Thus, as with the stored string output, the operation of these routines is independent of the Page 1 and Page 2 switching of graphics output.

The 12 Monitor print routines can be called by other 6502 routines. Four of the routines do not require parameter passing; these four are: CROUT, PRBLNK, BELL, and PRERR. Because they do not require parameters, these four can be called by the FELIX operator PRNTOP by passing the appropriate value to PRTPTR+1 and any "don't care" value to PRTPTR. These four routines perform the following functions: CROUT prints a carriage return and line feed through COUT; PRBLNK prints three blanks; BELL sounds the bell once; and PRERR writes "ERR" to the screen and sounds the bell. The other eight routines require parameters to be passed to PRINT.

The FELIX operation PRINT2 was specifically written in order to have an operation that could pass a parameter in the accumulator to PRINT and then call one of the three routines COUT, PRHEX, or PRBYTE. COUT prints the character whose ASCII code is in the accumulator; PRHEX prints the hexadecimal value contained in the low nybble of the accumulator; and PRBYTE prints the contents of the accumulator as a 2-digit

number. PRINT2 can also call the aforementioned routines CROUT, PRBLNK, BELL, and PRERR simply by passing the proper value to PRTPTR+1 and any value to the accumulator--it will simply be ignored.

The other five Monitor print operations, since they involve passing X-register or Y-register values to PRINT, can only be called by other 6502 routines and not directly by FELIX operations. These five routines are PRNTYX, PRNTAX, PRNTX, PRBL2, and PRBL3. The first three print hexadecimal numbers; PRBL2 prints (X) blanks; and PRBL3 prints the character in A followed by (X) - 1 blanks.

When a 6502 routine calls one of the Monitor print routines by way of a call to PRINT, it must first store the correct value in PRTPTR+1. This, of course, would be some value between \$F0 and \$FB. If one or more register values are to be passed, it loads the appropriate register or registers and calls the Monitor routine SAVE, a routine that saves register values. PRINT is then called.

When PRINT, by testing PRTPTR+1, determines that one of the Monitor routines is being asked for, the first action it takes is to use the number in PRTPTR+1 as an index into the table MONTAB--a table whose entries are the addresses of the 12 Monitor routines. This is done by masking out the high nybble and doubling the low nybble of the value in PRTPTR+1. The address taken from MONTAB is used to modify the code of the instruction "JSR DUMMY" in order to aim it at the proper routine.

Just before PRINT executes the JSR instruction, it calls the Monitor routine RESTORE which retrieves the register values previously saved by the Monitor routine SAVE. After the Monitor routine prints and returns, PRINT moves the new version of the text on Page 1 to TXTBUF, and, if Page 2 is active, to Page 2; PRINT then returns.

### 3.12 The Apple FELIX Trace

The Apple FELIX trace operation prints the memory location and opcode of the present FELIX instruction. Also, an arrow is printed between the memory location and the opcode. The TRACE routine is called by the EXEC routine each time that it executes an instruction if the trace flag, TRCFLG, is set. As with all other text output, the TRACE routine performs all of its printing by way of the PRINT routine. In this way, the trace information is automatically printed on the active page.

There are three FELIX operations used in conjunction with the TRACE routine: TRCOP, NTRCOP, and NOTOP. TRCOP (opcode = \$2A) simply sets TRCFLG. NTRCOP (opcode = \$2B) resets TRCFLG, and NOTOP (opcode = \$2C) is the Apple FELIX "no operation" operation. When writing long FELIX programs, a NOTOP instruction may be included at the beginning of each major section of code and each subroutine. Then, when debugging the program, the TRCOP or NTRCOP instruction can be put in these reserved locations as the need arises. This simple procedure has been very effective in debugging programs.

### 3.13 GETSTR and BLNKRD

Two routines that should be discussed in the general context of text I/O are the op-routine GETSTR and the base routine BLNKRD (blink-read). BLNKRD gets a string of keyboard input and echoes it on the screen and into the storage area INBUF, starting at location \$200. It also stores the length of the string in the variable LENGTH, which is located at memory location \$509D. The FELIX operation GETSTR has an

opcode of \$32. The purpose of this instruction is to call BLNKRD--by way of the op-routine GETSTR--from a FELIX program. When GETSTR is called by EXEC, it first checks TXTFLG. If TXTFLG is zero, GETSTR calls MXTMDE to initiate the mixed-text and graphics mode, and then it calls BLNKRD; if TXTFLG is set, GETSTR immediately calls BLNKRD. As well as being called by the FELIX operation GETSTR, BLNKRD can also be called by other base routines.

BLNKRD derives its name from the fact that it can place a cursor on an active Page 2. For proper operation, the cursor must be restricted to the 40 columns of the bottom line. Attempts to run the cursor off either end of this line can cause erratic behavior. Actually, this limitation is not very severe; user response to a program is usually no more than a few characters in length.

The cursor is placed on Page 2 in the following manner. Prior to the call to the Monitor keyboard input routine, RDKEY, a test is made to determine if Page 2 is active. If it is, the "cursor producing" code is executed. This code is patterned after the Monitor code that produces a cursor on Page 1. Using the base address \$BD0 assures us that the cursor will be placed on the bottom line of Page 2. The Monitor horizontal displacement variable, CH, is initially given the value zero. This is done as a precaution against the possibility that the immediately preceding text output was not terminated with a carriage return. For optimal operation, that is, to prevent overwriting of previous text, such a carriage return should be provided. After the RDKEY call, the text for an active Page 2 is again made, and, if it is active, the cursor is turned off by restoring the character in its nonblinking form

to its place on the screen. Storage of the new character in INBUF and printing it on the screen by way of a COUT call to PRINT is the same whether the active page is Page 1 or Page 2.

A nice feature about restricting this input to one line is that the index into INBUF is taken to be the same as the horizontal displacement value contained in the Monitor variable CH. CH is incremented and decremented by the Monitor routine. Thus, incrementing and decrementing of the index for storage into INBUF does not have to be done by BLNKRD. This is especially convenient when, as is frequently the case, backspacing is involved. Of course, this means that even if Page 1 is the active page, input should be restricted to one line for proper INBUF storage and string length determination.

### 3.14 The Permanent Objects and the WRITE Operation

The addresses of the permanent objects are stored in an object table which bears the name OBJTAB. OBJTAB occupies a memory block of two pages starting at memory location \$4500; therefore, there is space for the addresses of 256 objects. To simplify the code of routines that access the objects, the addresses of the first 16 objects--the standard FELIX objects--are also stored in OBJTAB. Since these objects are numbered \$0 to \$F, their addresses occupy the first 32 bytes of the table.

The object number of a character set object is the same as its ASCII code. The position of its address in the table can be found by multiplying its ASCII code by two. Thus, if objects are summoned by way of the keyboard, as they are in the routine WRITE, or by way of ASCII table storage, as they are in the routine FELIX-ARITH, access to

their addresses in OBJTAB is "built-in." In these circumstances, the writer of FELIX programs does not have to remember the object numbers. When it is necessary to know the object numbers, they can conveniently be obtained from an ASCII code table.

Whenever a FELIX routine needs to work with an object, it stores the object number in the accumulator and calls the subroutine GETADR. GETADR doubles the accumulator value, uses the doubled value as an index into OBJTAB, and moves the object address into the zero page pointer OBJECT. The 2-byte location OBJECT (\$7D.7E) is reserved exclusively as a pointer into the objects. The Apple keyboard routines use a "bit 7 set" version of the ASCII code. This yields the wrong object number, but the ASL instruction which accomplishes the doubling mentioned above shifts this faulty information out of the accumulator.

Each time the FELIX operation PATH is executed, it moves a specified object to the next position in a path which is defined--by a listing of successive centers of gravity--in a path object. Before returning to EXEC, PATH uses byte 255 of the moved object to store the present position. This feature of FELIX/8080 is retained in Apple FELIX for the original 16 objects, but another provision had to be made for the shorter permanent objects. A one-page block of memory, designated TW055 and starting at location \$6000, is reserved for this purpose. The index into TW055 for a given object is equal to its object number. This means, of course, that the first 16 bytes of TW055 are not used. Before storing or retrieving path position data, PATH always checks the object number to determine the proper storage location. As part of its initialization procedure, the routine EXEC clears the TW055 area.

The remainder of this section is devoted to a discussion of the WRITE routine. In this routine, except when BAKBUF is used for clearing purposes, Page 1 is the only page or screen buffer used. Throughout the execution of WRITE, Page 1 is both the active page (SHOBUF) and ONBUF. Consequently, the solution to the problem of the lack of a cursor on Page 2 is simply not to use Page 2.

Again, except for clearing purposes when the subroutine MOV1K is used, the only part of SHOW used by WRITE is the subroutine SHOW1. SHOW1 is passed an object number in the X register; it obtains the address of the object by a call to GETADR and, using calls to PUT, paints the object on ONBUF. Since, in this case, ONBUF and the active page are one and the same, no buffer transfers are necessary.

The other FELIX routines used by WRITE are CLEAR, BKGND, GETADR, MXTMDE, and POSCAL. GETADR and MXTMDE have been discussed in other parts of this report. CLEAR is simply a link between calling base routines and the NEWBCK FELIX operation routine. CLEAR is supplied the color in the accumulator, and then it simply jumps to the third line of the NEWBCK routine, thereby bypassing the INPAR call. NEWBCK, of course, paints BAKBUF with the color--black, in this case--that is passed to it.

BKGND, not to be confused with the FELIX op-routine BKGDOP, is a base routine that moves BAKBUF onto ONBUF. BKGND, after supplying the two zero page pointers with the appropriate addresses, calls the subroutine MOV1K. As its name implies, MOV1K moves 1024 bytes of data to and from memory locations indicated by the pointers. CLEAR and BKGND are called to clear BAKBUF and Page 1 only during the initialization portion of WRITE.

The FELIX operation POSIT positions an object in a new location in the sense that it recalculates and then stores in memory its new pixel locations. To get the data it needs for the calculations, the POSIT routine uses three INPAR calls. With these calls, it gets the object number and the X and Y coordinates of the new center of gravity of the object. POSCAL is not actually a routine; it is the entry point for 6502 routines that call POSIT. The routines pass the needed values in the three registers and enter POSIT at POSCAL which is the point immediately following the INPAR calls.

Since WRITE always puts the next part of the character string on the bottom graphics line, the Y coordinate passed to POSIT remains fixed at \$C. The first X coordinate value is four, and succeeding X values are obtained by adding six to the previous value. When X equals \$28, we have "run off the screen," and a call to the SCROLL subroutine is executed.

WRITE gets help from the Apple Monitor in two ways. It uses the Monitor routine NXTCHAR to get the next character string from the keyboard, echo it on the screen, and store it in the buffer INBUF. Secondly, it gets a random number from memory location \$4E when it is selecting a color for the first character in a string. The value in location \$4E is random in the sense that while waiting for keyboard input, the Monitor continuously increments the 2-byte location at \$4E-4F. The color of a character other than the first depends on the ASCII code of the previous character. This code, or the value from location \$4E, is ANDed with the number three. The resulting number is used as an index into the table, COLOR, which contains the code numbers of the four colors used.

WRITE performs a validity check on each object number it receives. Invalid numbers are ignored. The five-line LMTCHK routine checks if the number is between \$2A and \$5B, inclusive. All but five of the valid object numbers are within this range. If the number is not in this range, it is checked against these five numbers in the code immediately below the LMTCHK routine. Otherwise, a branch is taken to the line labeled TEST 3 to compare the number with the three numbers that lie within the LMTCHK range, but are not, at present, valid object numbers.

Before closing this section, some information concerning the 2-byte entries in the table NUMS is given. First, recall that the graphics lines are four text lines deep. This is fortunate, in that in so far as their addressing pattern is concerned, the 24 text lines of Page 1 can be thought of as being divided into six groups of four lines each. That is, given the address of the first byte of the first line in a group, succeeding first byte addresses for succeeding lines within the group can be obtained by adding \$80 to the previous address.

The NUMS entries \$428, \$628, and \$450 are the first byte addresses for the text lines eight, 12, and 16, respectively. \$600 is the first byte address for text line four; FRMPTR gets this address during the initialization of SCROLL. At the same time, TOPTR is started off with \$400, the first byte address for text line zero. Subsequently, after each four-line move, FRMPTR gets its first byte addresses from NUMS, and TOPTR takes the old FRMPTR values, after undoing the three additions of \$80 with a subtraction of \$180.

The last entry in NUMS, \$C00, is the address corresponding to the first position in BAKBUF. Since BAKBUF had previously been cleared to

black, it is a convenient source to use to clear the bottom graphics line.

### 3.15 The 6502 Routines of CIPHER

The flag that is lowered at the beginning of the op-routine ARITH is the communications link between ARITH or ARITH2 (the continuation of ARITH) and the FELIX program FELIX-ARITH. It is lowered at this point in case the user responds to the BLNKRD call a few lines hence with an S, indicating the termination of CIPHER processing. In the absence of an S, ARITH2 will eventually supply FLAG with the value \$FF, which is a signal to FELIX-ARITH to produce graphics output.

An acceptable user input of a problem consists of a number followed by the operation symbol followed by the second number. The numbers cannot exceed five digits in length. The input string is rejected if it contains any spaces, and new input is requested. This "no spaces" restriction is not overly burdensome for the user, and it greatly simplifies the validity checks necessary for proper problem input. For the ASCII storage of the operands into NUM1 and NUM2, start at the beginning of INBUF at location \$200 to pick up the first digit of the first number. For the BCD storage into NUM1LH and NUM2LH, start with the last digit of the second number and work toward location \$200.

The two locations NUM1LH and NUM2LH are central ones for the CIPHER routines. When entered, the four arithmetic routines DECADD, DECSUB, DECMUL, and DEC DIV find the two operands in these locations. DECADD and DECSUB return the answer in NUM1LH. ARITH, after calling DECMUL, moves the product to NUM1LH. ARITH2, after calling DEC DIV, moves the quotient to NUM1LH and the remainder to NUM2LH.

After storage of the numbers and their lengths (in COUNT1 and COUNT2), branch off to the appropriate sections of code as dictated by the operation symbol whose code has previously been stored in the variable OPER. Those sections of code which call DECADD, DECSUB, and DECMUL are all part of the routine ARITH. The code that calls DECDIV starts off the routine ARITH2. If a call to DECSUB results in a negative answer, an error message is printed, and new input is requested. Since the graphics screen cannot accommodate a product of more than six digits, verify that the sum of COUNT1 and COUNT2 is less than seven before calling DECMUL.

After the return from DECADD, DECSUB, or DECMUL, jump to the location SHOPRP in the source file ARITH2. SHOPRP is a mnemonic for SHOW PREPARATION. At this location, those tasks that follow the arithmetic calculation but precede transfer of control to FELIX-ARITH are performed.

Preceding the SHOPRP section of code in ARITH2 is the initialization code necessary for a DECDIV call, as well as the call itself. Both the dividend in NUM1LH and the divisor in NUM2LH must be normalized before the call. When two values are to be normalized, as in the present case, the subroutine NRMCAL is entered at the top. If it is wished only to normalize the answer in NUM1LH, as is done in subsequent code, NRMCAL is called at the entry point NCALLB. In any case, NRMCAL, in turn, calls NORM, which normalizes the 3-byte area on Page \$65 pointed to by the variable ROLPTR. The final link in the chain of subroutines is the routine ROTATL in the source file DECADD. NORM calls ROTATL to do the actual bit shifting.

As a momentary digression, it is desirable to point out that except for the six-line routine PRTMSG, the source file DECADD is the collection of all routines from the 6502 COOKBOOK by Robert Findley that are used in Apple FELIX. Also, the routine NUMCHK in the ARITH source file is inspired by, and is a correction of, the LMTCHK routine in the same book.

NORM knows that its job is done when the most significant non-zero digit of the number appears in the high-order nybble of the third byte of the 3-byte storage area. The variable SIGCNT is initialized to five and is decremented each time that a shift of the digits results in yet another value of zero in the key nybble. Thus, SIGCNT records the number of significant digits in the operand. A value of zero also causes an exit from NORM. If zero appears in SIG2 (by way of SIGCNT) after the divisor has been normalized, abort the process and ask for new input.

NORM moves digits within the 3-byte area in the same pattern that ROTATL moves bits in this area. This is true because NORM always calls ROTATL in groups of four calls each. After clearing the carry, ROTATL executes the ROL instruction on each of the three bytes, starting with the low-address byte. Bit 0 of the first byte gets the zero from the carry. Since the carry is only cleared once, bit 7 of the first byte is moved into bit 0 of the second byte, and bit 7 of the second byte is moved into bit 0 of the third byte. Bit 7 of the third byte is lost. The net result of these rotations to the left is to shift the number to the right (that is, into locations with higher addresses).

The DECDIV routine performs division by subtracting the divisor from the dividend until a borrow occurs. At this point, the divisor is

added back to the dividend. Each time a non-borrow subtraction occurs, the quotient is incremented. As written, DECDIV is a floating-point routine in which six-digit operands are assumed; the process mentioned above is cycled through six times to yield a six-digit quotient.

Between each series of subtractions, the new value of the dividend is rotated one digit to the right, using a 4-byte area. This puts the most significant digit into the fourth "extension" byte and a zero in the low nybble of the first byte. This serves the same purpose as "bringing down a zero" from an extended dividend in the conventional division algorithm. Also at this time, the quotient is rotated so that each new digit of the quotient is in the least significant position.

Just prior to the DECDIV call, the first line of its code is modified so that it executes the subtraction cycle one more time than the difference between the number of digits in the dividend and the number of digits in the divisor. This has the effect of giving a whole number quotient and remainder. Also at this point, the remainder has as many digits--possibly including leading zeros--as the divisor has significant digits. This fact is used to produce a remainder of the proper magnitude.

The quotient and remainder are normalized after the DECDIV call. This updates SIG1 and SIG2 and positions the numbers so that it is easy to pick up the digits when converting to ASCII and store the numbers in the variables ANSWER and REMAIN.

The DECDIV routine is not called if the divisor has more significant digits than the dividend; in this case, the little algorithm breaks down. However, under these conditions the quotient is zero, and

the remainder is the dividend. Therefore, simply move the dividend into NUM2LH and zero into NUM1LH. The numbers are already normalized.

For the add, subtract, or multiply operation, a line will be drawn by FELIX-ARITH under the second operand when the problem is presented on the graphics screen. The line will be formed by placing the 6-byte long line segment, Object #0, end-to-end "LINLEN" number of times. For addition and subtraction, the variable LINLEN receives the larger of the values in COUNT1 and COUNT2 after the latter is incremented to take the operation symbol into account. For multiplication, the value is the sum of COUNT1 and COUNT2; for division, it is zero.

After the answer has been normalized, it is converted to ASCII code and stored. If a zero in LINLEN indicates the division operation, the conversion and storage loop is executed a second time to store the remainder. The subroutine RANDOM is next called to generate and store, at location TABLE2, a permutation of the numbers one through six. This "roll of the die" is used by FELIX-ARITH in the graphics presentation of the answer. Before returning to EXEC, the variable FLAG is given the value \$FF to signal FELIX-ARITH that it has another problem to present on the screen.

Before exiting this section, the operation of the DECMUL routine and the RANDOM routine will be explained. In DECMUL, all numbers are stored in low-high order. The multiplier is in the 3-byte area starting at DCAC. The multiplicand is in the 4-byte area starting at DCOP. The fourth byte of the multiplicand area always contains zero. The "partial product register," as Findley calls it, is a 7-byte area starting at DCPPO. When the multiplication is complete, the product is found in the

partial product register starting with the least significant byte at DCPP0. Before multiplication begins, the partial product register is cleared.

Corresponding to the assumption of a six-digit multiplier, the multiplication is achieved by the execution of a series of six add-and-shift operations. In the add part of the operation, the multiplicand is added to the partial product as many times as is dictated by the operative digit of the multiplier. To be precise, the multiplicand is added to that number which resides in the last four bytes of the partial product register. The last of these four bytes serves as a repository for the possible carry-out of the prior 3-byte addition. The fourth byte in the multiplicand, in turn, exists simply to have a number to add to this last byte.

After the addition part of the add-and-shift operation is completed, both the partial product and the multiplier are shifted left by one digit. The multiplier shift causes the present least significant digit in the low nybble of DCAC to be shifted into the bit bucket, being replaced by the next operative digit. The first three bytes of the partial product area serve as "shift absorbers" so that no values are lost when the partial product is shifted. This shifting to the left (that is, into lower addresses) of the previous partial product written in low-high order corresponds to the shifting of subsequent partial products in ordinary multiplication. Since there is always a total of six shifts, the least significant digit of the product always finally resides in the low nybble of DCPP0.

The Monitor generates a "random" number by incrementing the 2-byte location \$4E-4F in its keyboard wait loop. Using the appropriate logic operations, the base routine RANDOM transforms the numbers at \$4E and \$4F to randomly pick a ROM address between \$F000 and \$FFFF. Starting at this address, RANDOM uses the contiguous values stored in memory to aid in generating a permutation of the numbers one through six. Infrequently, the address pointer wraps around to Page 0, but this does not cause any problems.

The permutation resides in locations TABLE2+1 to TABLE2+6. As execution of RANDOM proceeds, the numbers one through six are placed in TABLE2, starting with one and proceeding in increasing order to six. The "randomness" of the permutation derives from the fact that the values in memory are used to select the position in TABLE2 where the next number is to be placed. When a location in TABLE2 has been filled, the corresponding position in TABLE is set to record this fact. In an experiment involving 12 runs, the average number of memory accesses was 43, with the largest number being 101.

There is also a FELIX operation, RND, with opcode \$38, which calls RANDOM. Before the op-routine for RND calls RANDOM, it increments the 2-byte location \$4E-4F. Of course, the best results are achieved if RANDOM is called only once for each execution of the Monitor routine RDKEY, but this is not always possible. In some cases under these circumstances, the increment yields a passable degree of variability, especially when only one number is needed.

### 3.16 The FELIX Program FELIX-ARITH

When FELIX-ARITH begins execution, the two operands are stored in NUM1 and NUM2, with their lengths in COUNT1 and COUNT2. The operation symbol is in the variable OPER. The variable LINLEN contains the number of copies of the 6-byte long line segment, Object #0, needed to draw a line under the second operand. LINLEN equals zero if the operation is division. The solution is in the location ANSWER--and REMAIN, if the operation is division. The number of digits contained in ANSWER, and in REMAIN if appropriate, is recorded in SIG1 and SIG2, respectively. The codes for the digits and the operation symbol in NUM1, NUM2, OPER, ANSWER, and REMAIN are the object numbers for these symbols since the object number is identical to the ASCII code in every case except for the division symbol, and that code is adjusted before entry into FELIX-ARITH.

In addition to the items mentioned above, FELIX-ARITH works with two path objects, Object #1 and Object #2, and also the randomly selected permutation of the numbers one through six located in TABLE2. These are used to give some animation and variety to the presentation of the answer. Path Object #1, called three times, causes the next digit of the answer to bounce back and forth--with color changes--at the bottom of the screen before it takes its final position in the answer. Once there, it spins two complete revolutions by 45-degree turns before it settles into place. Path Object #2 is similar to Path Object #1 except that it is a few pixels lower. It is used with division since division uses two lines on the screen for its answer, allowing less space for the animation.

Starting with the least significant digit, the digits of the value in ANSWER are numbered 1, 2, ..., SIG1, and those of the remainder are numbered 1, 2, ..., SIG2. In placing the solution on the screen, the order in which the digits appear corresponds to the order in which their number designations appear in the permutation. In the case of a division problem, the same permutation is used for both the quotient and the remainder.

The initial value of the Apple FELIX program counter is always \$3200. At this location, the bordered background for CIPHER, which is stored in the 1K block of memory starting at location \$7000, is moved to the background area at \$C00. Then the FELIX operation ARITH (opcode \$2F) is executed. This causes ARITH, ARITH2, and the appropriate decimal routine in the source file DECADD to get the problem from the user, do the arithmetic, store the values, and then return to the EXEC routine. Immediately following the ARITH instruction, there is a "JUMP\$3273" FELIX instruction. \$3273 is the beginning address of the main body of FELIX-ARITH.

At the top of FELIX-ARITH, the variable FLAG is tested. If it contains \$FF, then FELIX-ARITH proceeds to display the problem and solution on the screen. If not, the user has signaled the end of the arithmetic session by an input of the letter S and, therefore, jumps out of FELIX-ARITH to location \$34FF. Any FELIX program could be placed at this location. Since, in the present case, CIPHER and SLATE are being demonstrated together, the WRITE instruction opcode, \$2E, is at location \$34FF. When the user indicates termination of the use of WRITE, or SLATE, with a Control-E input, the EXEC routine finds a "JUMP\$3200"

instruction at \$3500, and CIPHER is begun again. A "JUMP\$3200" instruction is also executed after normal execution of FELIX-ARITH.

After the FLAG test, the objects are given colors. The letters Q and P and the line segment, Object #0, are always magenta. The operation symbol is magenta unless the numbers are magenta, in which case it is blue. The numbers used to present the problem are all of one color. For each new problem, an index, CLRIDX, is incremented and then used to get a color from a 4-byte table, CLRTAB. A loop that colors all ten digits is then entered. If CLRIDX exceeds three, it is reinitialized to zero. Since the path objects recolor the objects, the solutions are varicolored.

After the coloring, the first operand is moved onto the background. COGX (REG2) is given the value \$28. Since the operands are placed on the background from right to left, each subtraction of six gives COGX the correct value for the X coordinate of the next digit's center of gravity. For the first operand COGY (REG3) is given, and retains, the value \$2B. REG0 is given the sum of the address of NUM1 and the value in COUNT1. This sets it up as a pointer into NUM1 using the Type 4 auto-decrement mode. REG1 is given the address of NUM1; when REG0 decrements down to the value in REG1, the first operand is on the background. The digits are placed on the background in the appropriate position by first executing a POSIT and then a BCKGND FELIX instruction. The second operand with the operation symbol is placed on the background in the same manner. If LINLEN is not zero, then a line is placed under the second operand. If it is zero, then the letters Q and R are placed on the extreme left of the lines where the quotient and the remainder,

respectively, will soon appear. After the above has been accomplished, the FELIX operation SHOW is executed twice, and the problem appears on the screen.

Each time through the fetch and execute cycle in the EXEC routine, a check is made for keyboard input. If there is input and if the variable CHAR contains zero, the input is stored there. If CHAR already has a nonzero value, the bell is sounded, and the input is ignored. CHAR can be accessed as a Type 3 input port. In this case, if CHAR has a zero value, the user is prompted with a bell until keyboard input is supplied. If, as in the present circumstances, bell prompting would be inappropriate, CHAR can be addressed directly by the FELIX program. This arrangement allows writing a silent "wait" routine that is not exited until the user supplies keyboard input. In FELIX-ARITH, after the problem has appeared on the graphics screen, the message "TOUCH ANY KEY FOR THE ANSWER" is printed, and such a wait loop is entered. In this way, the user can delay the presentation of the solution to the problem which is already on the screen.

An explanation of the presentation of the contents of ANSWER (and REMAIN in the case of division) is probably best centered around the roles of the three pointers, PTR0, PTR1, and PTR2. At the beginning of the presentation of ANSWER, PTR1 is given the address of TABLE2. Recall that the permutation of the numbers one through six is located in the 6-byte area starting at TABLE2+1. Unlike PTR1, PTR0 and PTR2 are initialized for each new digit of ANSWER. PTR0 is initialized to point at the first byte beyond the byte containing the least significant digit in ANSWER. PTR2 is initialized to point at the table XLOCS. As with

TABLE2, the first byte of XLOCS is wasted. For  $I = 1$  to 6,  $XLOCS+I$  contains the X coordinate for the center of gravity of the I(th) digit.

The placement of a digit begins by incrementing PTR1 into TABLE2 to pick up the next number in the permutation. If it is not larger than the total number of digits in the answer, this number determines the next digit to be put on the background. For example, if ANSWER contained the ASCII code for the number 12345 and PTR1 was pointing at the location containing the digit two, then four would be the next digit placed on the background. PTR0 and PTR2 are "fine-tuned" by subtracting the value found by PTR1 from PTR0 and by adding it to PTR2. From this point on, digit placement is as previously described. Since the digits are spun in their final position, it is noted that PTR2 is given its value in the animation subroutine.

### 3.17 The SOUND and NOTES Operations

The FELIX instruction SOUND is followed by two parameters. The first parameter contains the address of a "note duration" table, and the second parameter contains the address of a "note pitch" table. When the SOUND op-routine is entered, these values are used to modify the code of an indexed LDX instruction and an indexed LDA instruction so that the correct table areas are being accessed. For the convenience of the FELIX language programmer, space is provided for two 30-byte tables following the SOUND op-routine code, but their use is, of course, optional.

The purpose of the SOUND operation is to produce sound effects--not to play tunes. Correlating frequency and duration arguments with the notes of a scale is a far more sophisticated process than has been

attempted with the SOUND operation; such correlations are established for the NOTES operation which shall be discussed presently. On the other hand, producing a desired sound effect involves extensive experimentation with pitch and duration arguments. This is the reason two separate tables are used in the SOUND op-routine. It is simply much easier to change values using this format.

In the operation of the SOUND op-routine, the pitch of the "note" is determined by passing the frequency argument to the Monitor WAIT routine. Unless the frequency argument equals \$FF, indicating a rest, the speaker switch is toggled when the WAIT subroutine returns. The duration of a note is set by placing the WAIT call inside a loop indexed by the duration number. A duration number of zero signals the end of the sound production.

The core of the NOTES op-routine is a segment of code due to Hal Chamberlin (7) which consists of a pair of very meticulously constructed nested timing loops. He uses a wait loop indexed by a frequency argument, M, embedded inside a duration loop indexed by a duration argument, N. Addressing the speaker switch with a load or read instruction causes the speaker cone to move in one direction. Therefore, two read instructions are required for a complete audio cycle. For this reason, the wait loop actually consists of two loops, each of which is preceded by an LDX SPKR instruction. Each loop is executed M times.

M is chosen so that the time required for one execution of the outer duration loop is equal to the period of the audio signal being generated. The period, of course, is the reciprocal of the frequency, and tables of note frequencies are available in the literature. The book cited in Chamberlin (7) has such a table on page 143.

The details of the calculation of the frequency argument,  $M$ , are now given. Exactly 15 machine cycles are required to execute each wait loop, except that only 14 machine cycles are required for the last time through the loop since the branch to the top of the loop is not taken in this case. Therefore, the total number of cycles expended in executing the wait loops  $M$  times is  $30*M - 2$ . The number of cycles required to execute the remaining code in the outer loop is 50. This brings the number of machine cycles for one execution of the outer loop to  $30*M - 2 + 50$ , or  $30*M + 48$ .

According to Dougherty (9, p. 42), a machine cycle requires 1.023 microseconds. Thus, multiplying  $30*M + 48$  by 1.023 and dividing by 1,000,000 yields the execution time, in seconds, for the outer loop. Setting this time equal to the period corresponding to a given note enables calculation of the value of  $M$  for that note.

This calculation has been made for 45 successive notes (half-tones) starting with C below middle C. These 45 values are entries in the table PITCH1 of the NOTES op-routine, starting at PITCH1+1. The value at PITCH1+0 is an "unused" frequency argument and signifies a rest note. \$E1 is used instead of 0 because, for correct timing, even a rest note must have a "frequency."

Even though it is also frequency dependent, the duration argument,  $N$ , is much easier to calculate than the frequency argument,  $M$ . If a tempo of 100 beats or quarter-notes per minute is assumed, then a simple calculation shows that a sixteenth-note is allotted .15 seconds. Dividing the period of a note into .15 yields the duration argument for that particular note. This calculation was made for the 45 notes (and the

rest "note"), and the results constitute the entries in the table DORA of the NOTES op-routine.

The final table that is part of the NOTES op-routine is TEMTAB. The 16 entries in TEMTAB are multiples of one-eighth, starting at two and decreasing down to one-eighth. These numbers are in floating-point notation. TEMTAB is used, as described below, to provide the FELIX language programmer with some control over the tempo of a song.

The "mnemonic with parameters" for the NOTES instruction is NOTES P,Q, where P is the address of the "song table," and Q is a tempo number between \$0 and \$F. For each note of the song, there are two adjacent values in the table: a note number and a duration number. The lowest note, C below middle C, is assigned the note number one. Number assignments increase for successively higher notes, ending with \$2D being assigned to the highest note, G sharp in the fourth octave.

A sixteenth-note has a duration number of one. A quarter-note--four times longer in duration than a sixteenth-note--is assigned the number four. In a similar manner, all notes are assigned duration numbers that signify their durations as multiples of the length of a sixteenth-note.

For the correct operation of the tone generation code for a given note, the frequency argument and the duration argument are needed. The NOTES op-routine simply uses the note number from the song table as an index into the table PITCH1 in order to obtain the frequency argument M.

Obtaining the duration argument N is a bit more complex. The note number is used as an index into the table DORA to obtain the duration argument for a sixteenth-note of the given frequency. This duration

argument is multiplied by the duration number from the song table to give the correct argument for a note of the given length. Multiplying by a tempo factor yields the final result.

The tempo factor is a 2-byte entry from the table TEMTAB. The index into TEMTAB is twice the value in the parameter Q. If the Q value is larger than eight, the tempo is increased. If the Q value is less than eight, the tempo is decreased.

The calculations mentioned above are performed using some of the floating-point routines resident in the Apple's ROM. Specifically, the FLOAT, FMUL, FADD, and FIX routines are utilized. The FLOAT routine converts a 2-byte integer into a floating-point number. The FIX routine converts from floating-point to integer form. Details concerning the floating-point routines are given in the article by Martellaro (1).

## CHAPTER 4

### FUTURE DEVELOPMENT

As noted in Moshell (4, pp. 19-21), several additional FELIX instructions have been proposed but have not yet been implemented in the present versions of FELIX. Five of these are logic or byte modification instructions: AND, OR, XOR, SHIFT, and ROTATE. There is a TURN instruction which rotates an object about a point other than its center of gravity. The last two instructions in this group are TURNON and TURNOFF. TURNON puts an object on, and TURNOFF removes an object from, ONLIST. Further details about these proposed instructions can be found in the reference cited above.

The PATH instruction is one of the most interesting and potentially powerful instructions in the FELIX set. Its power is enhanced by the instruction PTHSET P,Q which is already part of the Apple FELIX instruction set. At this point, an instruction PTHRD P,Q is proposed. This instruction would return, in the variable Q, the position of object P. Subtle manipulations of an object on a path would be possible if PTHSET and PTHRD were used to complement each other's function. For instance, a large path object, utilizing all 256 bytes of its storage, could be divided into subpaths.

On the broader systems level, future development should include the writing of an assembler for the Apple FELIX language. A PAINTER program for drawing backgrounds has been written in the FELIX language; a DOODLER program for drawing and storing objects should also be part of

the Apple FELIX system. If there is a potential of broader interest in Apple FELIX, a determination as to how that interest should be met has to be made, and a User's Guide would need to be written.

Perhaps the most important future development for Apple FELIX would be the creation of more FELIX programs written by talented programmers, preferably working with imaginative and artistically talented co-workers. The potential of computer graphics will be realized when the computerist and the artist work together.

## REFERENCES

## REFERENCES

1. Martellaro, John, "Apple's Hidden Floating-Point Routines," MICRO-COMPUTING, No. 38, February 1980, pp. 132-135.
2. Findley, Robert, 6502 SOFTWARE GOURMET GUIDE & COOKBOOK, Scelbi Publications, Elmwood, CT, 1979.
3. APPLE II REFERENCE MANUAL, Apple Computer, Inc., Cupertino, CA, 1979.
4. Moshell, J. M., L. Wittenberg, J. Kieneger, J. McCrae, T. Shultze, C. Wong, M. Sundaram, FELIX, THE COMPUTER ANIMATION TEACHING TOOL: OVERVIEW AND USER'S GUIDE, CS-79-36 (FELIX Report 1), REVISION #1, November 1979, The University of Tennessee, Knoxville; Computer Science Department Report.
5. Moshell, J. M., L. Wittenberg, J. Kieneger, J. McCrae, T. Shultze, C. Wong, M. Sundaram, FELIX/8080 VERSION 2.1 IMPLEMENTATION, CS-79-39 (FELIX Report 2), September 1979, The University of Tennessee, Knoxville; Computer Science Department Report.
6. Moshell, J. M., L. Wittenberg, D. Johnson, B. Elliott, FELIX/8080 UTILITY PROGRAMS--IMPLEMENTATION, CS-79-40 (FELIX Report 3), March 1980, The University of Tennessee, Knoxville; Computer Science Department Report.
7. Chamberlin, Hal, "A Sampling of Techniques For Computer Performance of Music," THE BYTE BOOK OF COMPUTER MUSIC, Christopher P. Morgan, Editor, Byte Publications, Peterborough, NH, 1979, pp. 47-50.
8. Struve, Bill, "A \$19 Music Interface (and Some Music Theory for Computer Nuts)," THE BYTE BOOK OF COMPUTER MUSIC, Christopher P. Morgan, Editor, Byte Publications, Peterborough, NH, 1979, pp. 33-40.
9. Dougherty, William E., THE APPLE II MONITOR PEELED, Rainbow Computing, Inc., Northridge, CA, 1979.
10. Wozniak, Stephen, "Sweet 16: The 6502 Dream Machine," BYTE, Vol. 2, No. 11, November 1977, pp. 150-159.
11. Vile, Jr., Richard C., "Sweet-16 Programming Using Macros, MICRO, No. 20, January 1980, pp. 25-29.
12. Wadsworth, Nat, INTRODUCTION TO LOW RESOLUTION GRAPHICS, Scelbi Publications, Elmwood, CT, 1979.
13. RASCAL AND INTERPAS REFERENCE MANUAL, Version II.0, Gentleware Corp., May 1981.

14. Lucido, A. P., "Software Systems for Computer Graphics." TUTORIAL: COMPUTER GRAPHICS, Kellogg S. Booth, Editor, IEEE Computer Society, Piscataway, NJ, 1979, pp. 204-212.
15. APPLE PASCAL REFERENCE MANUAL, Apple Computer, Inc., Cupertino, CA, 1980, pp. 90-100.
16. DeFanti, Thomas, "Language Control Structures for Easy Electronic Visualization," BYTE, Vol. 5, No. 11, November 1980, pp. 90-106.

## APPENDIX

# APPENDIX A

```

* -----
* *   FELIX-ARITH   *
* -----
*
* THIS FELIX LANGUAGE PROGRAM IS THE FOURTH AND
* FINAL PART OF THE "CIPHER" OR "ARITH" SERIES OF
* PROGRAMS. ITS PURPOSE IS TO SHOW THE PROBLEM
* AND THEN THE ANSWER ON THE SCREEN. AFTER MOVING
* THE BORDERED BACKGROUND FROM ITS STORAGE AREA
* INTO THE BACKGROUND BUFFER, THE PROCESS GETS
* UNDER WAY WHEN THE OPCODE, $2F, OF THE OP-ROU-
* TINE "ARITH" IS EXECUTED BY THE "EXEC" ROUTINE.
* "ARITH", ASSISTED BY THE PROGRAMS IN THE SOURCE
* FILES "ARITH2" AND "DECADD", GETS THE PROBLEM
* FROM THE USER, CALCULATES THE SOLUTION, STORES
* THE VALUES, AND THEN RETURNS TO THE EXEC ROUTINE.
* AT THIS POINT, WITH A JUMP TO LOCATION $3273,
* FELIX-ARITH TAKES OVER.
*
      .OR $3200
FLAG3  .EQ $636E           : "FLAG" IN ARITH
OUT    .EQ $34FF
CLRIDX .EQ $3FFB           : INDEX INTO CLRTAB
CLRTAB .EQ $66AA           : COLOR TABLE
COLOR  .EQ $3FFA           : COLOR STORE
MGNTA  .EQ #$1             : MAGENTA COLOR CODE
BLUE   .EQ #$2             : BLUE COLOR CODE
*
3200: 2C                   NOTOP           : TRCOP SPACE
3201: 31 A0 00 70          MOVMEM $7000,$0C00 : BACKGROUND IN PLACE
3205: A0 00 0C
3208: 2F                   ARITH           : ARITH OP BEGINS
3209: E0 73 32             JUMP $3273       : FELIX-ARITH BEGINS
      .OR $3273
* IF FLAG3 = #$FF, THEN SHOW THE PROBLEM AND
* ANSWER; OTHERWISE, JUMP INTO SOMETHING ELSE.
*
3273: 11 40 6E 63          COMP FLAG3,$$FF
3277: 20 FF
3279: E4 7F 32             JZ CNTUE
327C: E0 FF 34             JUMP OUT
*
* THE LETTERS "Q" AND "R" ARE USED WITH DIVISION
* PROBLEMS. OBJECT #0, A LINE SEGMENT, IS USED
* WITH THE OTHER OPERATIONS. WE NOW COLOR THESE
* OBJECTS MAGENTA.
*

```

```

327F: 2C          CNTUE NOTOP          :APPLE FELIX NO-OP
3280: 1D 20 51 01      RCLROP Q,MGNTA
3284: 1D 20 52 01      RCLROP R,MGNTA
3288: 1D 00 01          RCLROP OBJ#0,MGNTA
*
* THE OPERANDS OF THE PROBLEM GET THEIR COLOR FROM
* THE FOUR BYTE TABLE "CLRTAB". WE NOW SELECT
* THIS COLOR AND STORE IT IN THE VARIABLE "COLOR".
*
328B: 10 A0 AA 66      MOVE @CLRTAB,REG0      :BASE ADDRESS
328F: E0
3290: 10 40 FB 3F      MOVE CLRIDX,REG1      :GET INDEX
3294: E1
3295: 13 E1 01 E1      ADD REG1,#1,REG1      :INCREMENT INDEX
3299: 12 E1 04          COMPU REG1,#4        :STILL IN TABLE?
329C: E2 A2 32          JMI OK              :YES
329F: 10 00 E1          MOVE #0,REG1        :NO
32A2: 10 E1 40 FB OK    MOVE REG1,CLRIDX     :UPDATE INDEX
32A6: 3F
32A7: 13 E0 E1 E0      ADD REG0,REG1,REG0    :BASE @ + INDEX
32AB: 10 80 40 FA      MOVE (REG0),COLOR    :GET AND STASH COLOR
32AF: 3F
*
* THE PARENTHESES AROUND REG0 ABOVE INDICATES THE
* TYPE 4 INDIRECT ADDRESSING TYPE OF PARAMETER.
* ALSO, WE SHALL INDICATE THE AUTO INCREMENT WITH
* AN "I" AND AUTO DECREMENT WITH A "D".
* NEXT, WE COLOR THE OPERATION SYMBOLS MAGENTA IF
* THE DIGITS ARE NOT MAGENTA; OTHERWISE, WE COLOR
* THEM BLUE.
*
32B0: 12 40 FA 3F      COMPU COLOR,MGNTA
32B4: 01
32B5: E4 BE 32          JZ DIFCLR
32B8: 10 01 E2          MOVE MGNTA,REG2      :DIGITS NOT MAGENTA
32BB: E0 C1 32          JUMP CLROPR
32BE: 10 02 E2          DIFCLR MOVE BLUE,REG2  :DIGITS ARE MAGENTA
32C1: 1D 20 2A E2 CLROPR RCLROP "*",REG2     :MULTIPLICATION SYMBOL
32C5: 1D 20 2B E2      RCLROP "+",REG2      :ADDITION SYMBOL
32C9: 1D 20 2D E2      RCLROP "-",REG2      :SUBTRACTION SYMBOL
32CD: 1D 20 5B E2      RCLROP "/",REG2      :DIVISION SYMBOL
*
* NOW COLOR THE TEN DIGITS
*
32D1: 10 20 39 E0      MOVE #$39,REG0      :OBJECT NO. FOR NINE
32D5: 10 40 FA 3F      MOVE COLOR,REG1
32D9: E1
32DA: 1D E0 E1 TOP3     RCLROP REG0,REG1     :COLOR IT
32DD: 14 E0 01 E0      SUB REG0,#1,REG0     :NEXT NUMBER
32E1: 12 E0 20 30      COMPU REG0,$30       :ZERO DONE?

```

32E5: E1 DA 32  
32E8: E0 00 33

JPL TOP3  
JUMP \$3300

:NO  
:ROOM FOR ERROR

\*

\* THIS SECTION PUTS THE OPERANDS AND THE OPERATION  
\* SYMBOL ON THE BACKGROUND. THE TWO OPERANDS ARE  
\* AT "NUM1" AND "NUM2" (IN ASCII) AND THE OBJECT  
\* NUMBER OF THE OPERATION SYMBOL IS IN "OPER".  
\* "COUNT1" AND "COUNT2" CONTAIN THE NUMBER OF DIG-  
\* ITS IN NUM1 AND NUM2, RESPECTIVELY.

\*

.OR \$3300  
FLAG .EQ REG4  
NUM1 .EQ \$6101  
NUM2 .EQ \$6106  
OPER .EQ \$6111  
COUNT1 .EQ \$610C  
COUNT2 .EQ \$610D  
COGX .EQ REG2  
COGY .EQ REG3

\*

|       |                |                         |                       |
|-------|----------------|-------------------------|-----------------------|
| 3300: | 2C             | NOTOP                   |                       |
| 3301: | 10 00 E4       | MOVE #0,FLAG            | :FLAG NUM1 PLACEMENT  |
| 3304: | 10 A0 01 61    | MOVE @NUM1,REG1         | :THE STOP LOCATION    |
| 3308: | E1             |                         |                       |
| 3309: | 10 E1 E0       | MOVE REG1,REG0          |                       |
| 330C: | 13 E0 40 0C    | ADD REG0,COUNT1,REG0    | :INDEX INTO NUM1      |
| 3310: | 61 E0          |                         |                       |
| 3312: | 10 20 28 E2    | MOVE #\$28,COGX         | :WILL DECREMENT BY 6  |
| 3316: | 10 20 2B E3    | MOVE #\$2B,COGY         |                       |
| 331A: | 14 E2 06 E2 L1 | SUB COGX,#6,COGX        | :NEXT X POSITION      |
| 331E: | 07 90 E2 E3    | POSIT D(REG0),COGX,COGY | :POSITION OBJECT      |
| 3322: | 0F 80          | BCKGND (REG0)           | :OBJECT TO BACKGROUND |
| 3324: | 11 E0 E1       | COMP REG0,REG1          | :LAST DIGIT?          |
| 3327: | E3 1A 33       | JNZ L1                  | :NO - GET NEXT DIGIT  |
| 332A: | 11 E4 01       | COMP FLAG,#1            | :NUM2 DONE?           |
| 332D: | E4 4C 33       | JZ L2                   | :BOTH #'S ON BCKGND.  |
| 3330: | 10 01 E4       | MOVE #1,FLAG            | :FLAG NUM2 PLACEMENT  |
| 3333: | 10 A0 06 61    | MOVE @NUM2,REG1         | :INITIALIZE RUN #2    |
| 3337: | E1             |                         |                       |
| 3338: | 10 E1 E0       | MOVE REG1,REG0          |                       |
| 333B: | 13 E0 40 0D    | ADD REG0,COUNT2,REG0    |                       |
| 333F: | 61 E0          |                         |                       |
| 3341: | 10 20 28 E2    | MOVE #\$28,COGX         | :OFF STAGE - RIGHT    |
| 3345: | 10 20 23 E3    | MOVE #\$23,COGY         | :DOWN ONE "LINE"      |
| 3349: | E0 1A 33       | JUMP L1                 | :PLACE NUM2           |
| 334C: | 14 E2 06 E2 L2 | SUB COGX,#6,COGX        | :PLACING THE          |
| 3350: | 07 40 11 61    | POSIT OPER,COGX,COGY    | :OPERATION SYMBOL.    |
| 3354: | E2 E3          |                         |                       |
| 3356: | 0F 40 11 61    | BCKGND OPER             |                       |

\*  
 \* THIS SECTION OF CODE PUTS THE LINE UNDER THE  
 \* SECOND OPERAND FOR ADD, SUBTRACT, OR MULTIPLY.  
 \* FOR DIVISION, IT PUTS A Q (QUOTIENT) AND AN R  
 \* (REMAINDER) AT THE LOWER LEFT OF THE BACKGROUND.  
 \* "LINLEN" CONTAINS THE NUMBER OF TIMES WE LAY  
 \* DOWN A SIX PIXEL LONG SEGMENT TO FORM THE LINE.  
 \* IF LINLEN EQUALS ZERO, THE OPERATION IS DIVISION.  
 \*

Q .EQ #\$51  
 R .EQ #\$52  
 LINLEN .EQ \$6113  
 COUNT .EQ REG1  
 LINSEG .EQ #0  
 \*

|                      |                        |                      |
|----------------------|------------------------|----------------------|
| 335A: 10 40 13 61    | MOVE LINLEN,COUNT      | :NUMBER OF SEGMENTS  |
| 335E: E1             |                        |                      |
| 335F: E4 7D 33       | JZ L4                  | :IT'S DIVISION       |
| 3362: 10 20 28 E2    | MOVE #\$28,COGX        | :INITIALIZE THE      |
| 3366: 10 1E E3       | MOVE #\$1E,COGY        | :COORDINATES.        |
| 3369: 14E2 06 E2 L3  | SUB COGX,#6,COGX       | :ADJUST X COORD.     |
| 336D: 07 00 E2 E3    | POSIT LINSEG,COGX,COGY | :POSITION SEGMENT    |
| 3371: 0F 00          | BCKGND LINSEG          | :PLACE IT            |
| 3373: 14 E1 01 E1    | SUB COUNT,#1,COUNT     |                      |
| 3377: E3 69 33       | JNZ L3                 | :NEXT SEGMENT        |
| 337A: E0 8D 33       | JUMP L5                | :SHOW PROBLEM        |
| 337D: 07 20 51 04 L4 | POSIT Q,#4,\$\$1B      | :PLACE Q AND R       |
| 3381: 1B             |                        |                      |
| 3382: 0F 20 51       | BCKGND Q               |                      |
| 3385: 07 20 52 04    | POSIT R,#4,\$\$13      |                      |
| 3389: 13             |                        |                      |
| 338A: 0F 20 52       | BCKGND R               |                      |
| 338D: 01 L5          | SHOW                   | :BACKGROUND TO ONBUF |
| 338E: 01             | SHOW                   | :THERE IT IS!        |
| 338F: E0 BB 33       | JUMP \$33BB            |                      |

\*  
 \* ASK FOR USER RESPONSE AND THEN WAIT FOR THAT  
 \* RESPONSE BEFORE SHOWING THE ANSWER.  
 \*

\*  
 .OR \$33BB  
 CHAR .EQ \$5096  
 MSG .EQ \$66B6  
 \*

|                       |                |                    |
|-----------------------|----------------|--------------------|
| 33BB: 10 00 40 96     | MOVE #0,CHAR   | :RESET "CHAR"      |
| 33BF: 50              |                |                    |
| 33C0: 29              | MXTMDE         | :CLEAR TEXT AREA   |
| 33C1: 25 A0 B6 66     | PRNTP MSG      | :ASK FOR KEYSTROKE |
| 33C5: 10 40 96 50 LL1 | MOVE CHAR,REG0 | :WAIT LOOP         |
| 33C9: E0              |                |                    |

33CA: E4 C5 33

JZ LL1

\*  
 \* SINCE THIS IS THE FIRST TIME THROUGH THE CODE  
 \* BELOW, PUT ZERO IN "FLAG2". THERE WILL BE A  
 \* SECOND TIME ONLY IF WE HAVE A DIVISION PROBLEM  
 \* - TO SHOW THE REMAINDER. IF THE VALUE IN  
 \* "ANSWER" IS A QUOTIENT, THE Y COORDINATE OF THE  
 \* CENTER OF GRAVITY OF THE DIGITS IS #\$1A; OTHER-  
 \* WISE, IT IS #\$18. THE PATH OBJECT USED FOR  
 \* DIVISION IS OBJECT #2; THE OTHER OPERATIONS USE  
 \* OBJECT #1. THE ONLY DIFFERENCE IS THAT THE PATH  
 \* OF OBJECT #1 IS A FEW PIXELS HIGHER.

\*  
 ADDRES .EQ \$3FFC  
 FLAG2 .EQ \$3FFE  
 YLOC .EQ \$3FFF  
 SIG1 .EQ \$6372  
 ANSWER .EQ \$6362  
 NUMLEN .EQ REG3  
 DIGCNT .EQ REG4  
 PTHOBJ .EQ REG7

|                        |                      |                      |
|------------------------|----------------------|----------------------|
| 33CD: 10 00 40 FE      | MOVE #0, FLAG2       | :FIRST TIME THROUGH  |
| 33D1: 3F               |                      |                      |
| 33D2: 10 40 72 63      | MOVE SIG1, DIGCNT    | :NUMBER OF DIGITS IN |
| 33D6: E4               |                      | : "ANSWER".          |
| 33D7: 10 40 72 63      | MOVE SIG1, NUMLEN    |                      |
| 33DB: E3               |                      |                      |
| 33DC: 10 A0 62 63      | MOVE @ANSWER, ADDRES |                      |
| 33E0: C0 FC 3F         |                      |                      |
| 33E3: 10 40 13 61      | MOVE LINLEN, REG0    | :QUOTIENT?           |
| 33E7: E0               |                      |                      |
| 33E8: E3 F6 33         | JNZ LOWY             | :NO                  |
| 33EB: 10 1A 40 FF      | MOVE #\$1A, YLOC     | :YES, ANSWER IS      |
| 33EF: 3F               |                      | :A QUOTIENT.         |
| 33F0: 10 02 E7         | MOVE #2, PTHOBJ      | :OBJECT #2 IS PATH   |
| 33F3: E0 FE 33         | JUMP VTOP            |                      |
| 33F6: 10 18 40 FF LOWY | MOVE #\$18, YLOC     | :ANSWER IS NOT       |
| 33FA: 3F               |                      | :A QUOTIENT.         |
| 33FB: 10 01 E7         | MOVE #1, PTHOBJ      | :OBJECT #1 IS PATH   |

\*  
 \* IF THERE IS A REMAINDER TO SHOW, THE SECOND RUN  
 \* BEGINS, AT "VTOP". THIS INSTRUCTION POINTS  
 \* "PTR1" AT THE BEGINNING OF A SIX BYTE AREA HOLD-  
 \* ING A PERMUTATION OF THE NUMBERS ONE TO SIX.  
 \* THIS PERMUTATION WAS FORMED BY THE BASE ROUTINE  
 \* "RANDOM" IN THE "ARITH2" SOURCE FILE.

\*  
 TABLE2 .EQ \$656D

```

PTR1 .EQ REG1
*
33FE: 10 A0 6D 65 VTOP MOVE @TABLE2,PTR1
3402: E1
*
* "TOP" IS THE ENTRY POINT FOR PROCESSING THE NEXT
* DIGIT AS RANDOMLY SELECTED BY THE PERMUTATION.
* FOR EXAMPLE, IF THE NEXT PERMUTATION NUMBER IS
* ONE, THEN THE LEAST SIGNIFICANT DIGIT (DIGIT
* NUMBER ONE) IS THE NEXT DIGIT TO PROCESS
*
XLOCS .EQ $66AF
PTR0 .EQ REG0
PTR2 .EQ REG2
OBJECT .EQ REG5
*
3403: 10 C0 FC 3F TOP MOVE ADDRES,PTR0 :POINT ONE BYTE
3407: E0 :PAST THE
3408: 13 E0 E3 E0 ADD PTR0,NUMLEN,PTR0 :ANSWER.
340C: 10 A0 AF 66 MOVE @XLOCS,PTR2 :TABLE OF X CENTER OF
3410: E2 :GRAVITY VALUES.
*
3411: 13 E1 01 E1 TOP1 ADD PTR1,#1,PTR1 :GET NEXT
3415: 11 E3 81 COMP NUMLEN,(PTR1) :PERMUTATION NUMBER.
3418: E1 1E 34 JPL LL2 :IT'S NOT TOO LARGE
341B: E0 11 34 JUMP TOP1 :TOO LARGE-TRY AGAIN
*
341E: 14 E0 81 E0 LL2 SUB PTR0,(PTR1),PTR0 :POINT AT DIGIT
3422: 10 80 E5 MOVE (PTR0),OBJECT :GET OBJECT NUMBER
3425: E7 C0 34 CALL ANIMAT :ANIMATION OF DIGIT
3428: E0 32 34 JUMP $3432 :TEN SPARE BYTES
3432: 0F E5 BCKGND OBJECT :DIGIT ON BACKGROUND
3434: 01 SHOW :BACKGROUND TO ONBUF
3435: 01 SHOW :ONBUF TO SHOBUF
3436: 14 E4 01 E4 SUB DIGCNT,#1,DIGCNT :CHECK IT OFF
343A: E4 60 34 JZ CHKS :LAST DIGIT DONE
343D: E0 03 34 JUMP TOP :MORE TO DO
*
* IF LINLEN IS NOT ZERO, WE ARE DONE. IF LINLEN
* IS ZERO AND FLAG2 IS ONE, WE ARE DONE. OTHER-
* WISE, WE SET UP TO DISPLAY THE REMAINDER.
*
3460: 11 40 13 61 CHKS COMP LINLEN,#0
3464: 00
3465: E3 8C 34 JNZ DONE
3468: 11 40 FE 3F COMP FLAG2,#0 :FIRST TIME THROUGH?
346C: 00
346D: E3 8C 34 JNZ DONE :NO! WRAP IT UP.
*
* SET UP FOR SHOWING REMAINDER

```

```

*
SIG2 .EQ $6373
REMAIN .EQ $6368
*
3470: 10 01 40 FE      MOVE #1,FLAG2          :FLAG RUN NO. 2
3474: 3F
3475: 10 12 40 FF      MOVE #$12,YLOC          :NEXT LINE
3479: 3F
347A: 10 40 73 63      MOVE SIG2,DIGCNT        :NUMBER OF DIGITS
347E: E4
347F: 10 E4 E3          MOVE DIGCNT,NUMLEN
3482: 10 A0 68 63      MOVE @REMAIN,ADDRES      :NEW SPOT TO
3486: C0 FC 3F          :FIND DIGITS
3489: E0 FE 33          JUMP VTOP                :PROCESS REMAINDER
*
* WE ARE DONE. PRINT THE MESSAGE, "TOUCH ANY KEY
* TO CONTINUE", AND WAIT FOR RESPONSE. AFTER
* RESPONSE, JUMP TO $3200.
*
348C: 01              DONE SHOW                  :COULD BE A NOTOP
348D: 29              MXTMDE                     :CLEAR TEXT AREA
348E: 10 00 40 96      MOVE #0,CHAR              :RESET CHAR
3492: 50
3493: 25 A0 E5 66      PRNTP MSG2                 :PRINT INSTRUCTION
3497: 10 40 96 50 LL3  MOVE CHAR,REG0             :WAIT LOOP
349B: E0
349C: E4 97 34          JZ LL3
349F: E0 00 32          JUMP $3200                :START OVER
*
*
* ANIMATION SUBROUTINE
*
* THIS ROUTINE USES THE PATH OPERATION THREE TIMES
* TO MAKE THE OBJECT TAKE THREE DIFFERENT POSI-
* TIONS ON A HORIZONTAL LINE AT THE BOTTOM OF THE
* SCREEN. THE OBJECT ALSO CHANGES COLOR SEVERAL
* TIMES. AFTER THE OBJECT IS PUT IN ITS FINAL
* POSITION, IT IS SPUN FOR TWO REVOLUTIONS BEFORE
* SETTling INTO PLACE.
*
      .OR $34C0
COUNT .EQ REG6
*
34C0: 06 01 E5      ANIMAT LISTOP #1,OBJECT      :DIGIT ON ONLIST
34C3: 10 03 E6      MOVE #3,COUNT                :LOOP 3 TIMES
34C6: 24 E5 E7      SUBL1 PATH OBJECT,PTHOBJ      :BOUNCE THE DIGIT
34C9: E4 C6 34      JZ SUBL1                     :END OF PATH

```

```

34CC: 01          SHOW
34CD: 2D 20 17    SLOWOP #$13          :SHOW IT AWHILE
34D0: 14 E6 01 E6  SUB COUNT,#1,COUNT
34D4: E3 C6 34    JNZ SUBL1
*
* AFTER GETTING THE X COORDINATE, PLACE THE DIGIT
* IN ITS FINAL POSITION.
*
34D7: 13 E2 81 E2  ADD PTR2,(PTR1),PTR2 :ADDRESS OF X COORD.
34DB: 07 E5 82 40  POSIT OBJECT,(PTR2),YLOC
34DF: FF 3F
34E1: 01          SHOW                  :SHOW IT WITH
34E2: 2D 20 20    SLOWOP #$20          :A DELAY.
*
* SPIN THE DIGIT TWO REVOLUTIONS IN INCREMENTS
* OF 45 DEGREES.
*
34E5: 10 10 E6    MOVE #$10,COUNT      :TWO REVOLUTIONS
34E8: 0A E5 03    SUBL2 SPIN OBJECT,#3  :45 DEGREES
34EB: 01          SHOW
34EC: 2C          NOTOP                 :SPACE FOR "SLOWOP"
34ED: 2C          NOTOP                 :INSTRUCTION.
34EE: 14 E6 01 E6  SUB COUNT,#1,COUNT
34F2: E3 E8 34    JNZ SUBL2
34F5: 2D 20 0C    SLOWOP #$C          :SHOW AWHILE
34F8: 06 00      LISTOP #0            :CLEAR ONLIST
34FA: 04          RETOP                 :RETURN TO CALLER
*
*
* OUT
*
* IF THE USER HALTS "CIPHER", EXECUTE "SLATE".
* IF THE USER HALTS "SLATE", EXECUTE "CIPHER".
*
      .OR $34FF
*
34FF: 2E          OUT WRITE             :EXECUTE SLATE
3500: E0 00 32    JUMP $3200           :EXECUTE CIPHER

```

## VITA

John Stambaugh was born June 11, 1932 in Lewes, Delaware. He attended Lewes public schools, graduating in June 1950. He completed three years of study at Princeton University before transferring to the University of Texas at Austin, where he obtained his Bachelor of Arts degree in English in June 1957. After receiving his bachelor's degree, he taught at Briggs High School in Briggs, Texas. Subsequently, he returned to the University of Texas on an NSF grant and received his Master of Arts degree in Mathematics in January 1965. After obtaining the M.A. degree, he completed the course work toward a doctoral degree.

From 1967 until 1980, he was employed at Lyndon State College in Lyndonville, Vermont, where he was chairman of the Mathematics Department for seven years. During this time, he designed a Mathematics major for the college and taught most of the upperclass courses in this major. He was awarded the Master of Science degree in Computer Science in June 1981.