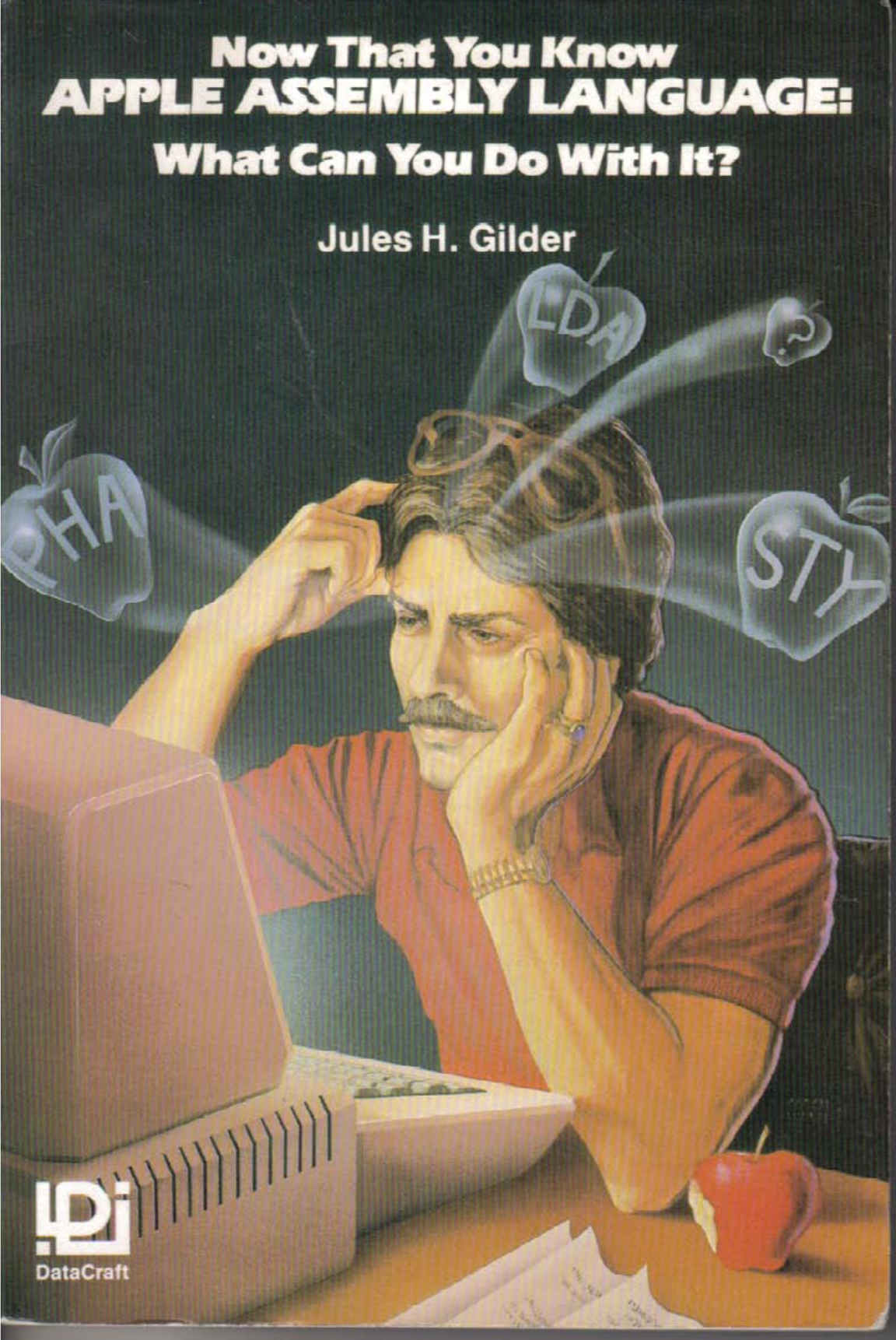
Now That You Know
APPLE ASSEMBLY LANGUAGE:
What Can You Do With It?

Jules H. Gilder



DataCraft

TABLE OF CONTENTS

LIST OF PROGRAMS

CHAPTER 2 - Getting Information Out of Your Computer

CHAPTER 3 - Getting Information Into Your Computer

CHAPTER 4 - Stealing Control of the Output

PREFACE

This book is designed to be used by the newcomer to assembly language programming, who has already spent the time required to learn assembly language programming for the 6502 microprocessor and is now anxious to put his or her new found knowledge to work.

The book comprises a work that took over six months to write and was totally produced, from program writing to typesetting, on an Apple II computer. There are over 50 programs in the book ranging from simple routines to help you input and output data, to more sophisticated programs that improve on the hardware — such as the Lowercase Input Driver — and programs that expand the Applesoft language — such as those in Chapter 8.

In addition, there are many interesting programs that you will find useful in your day-to-day work with the Apple. These include programs to help recover accidentally erased Applesoft programs, to format program listings and to improve the interface to your printer, to name a few.

Most of the programs in Chapter 6 were reprinted through the kind permission of Bob Sander-Cederlof of S-C Software. All of these programs deal with the generation of sound on the Apple. Bob puts out a monthly newsletter called Apple Assembly Line which is chock full of useful information for assembly language programmers. He also sells one of the best assemblers for the Apple, the S-C Macro Assembler. All of the programs in this book were written on that assembler. A special 10-byte patch to the assembler was provided by Bob, so that all of the assembled listings could be written directly to a text file. This file was then read by the word processing program and incorporated into the text of the book. As a result, none of the program listings were retyped, and thus you can be confident that all program listings will run as they are.

The programs in this book will work with the entire Apple II series of computers. There are some changes in the F8 ROM in the //c and //e that make it slightly incompatible with the II Plus. These occur in the KEYIN2 routine ($FD21). This entry point should not be used and programs should try to use the KEYIN entry point ($FD1B). All the programs in this book have been designed to overcome the difficulty posed by the differences in the input software.

These programs have been designed to run under DOS 3.3 although, in general, with minor changes, they can be used with ProDOS as well. Appendix D provides information you'll need to use these programs with ProDOS.

I'd like to say a special word of thanks to Dave Gordon, president of DataMost, for all the enthusiasm, encouragement and help that he has given me in producing this book.

Chapter 1

BEFORE YOU GET STARTED

The 6502 microprocessor is probably the most widely used microprocessor in personal computers. It is found in the Apple II and Apple /// families of computers, the PET, CBM and VIC computers from Commodore, the Atari 400 and Atari 800 computers, and a variety of other computers and video games. Because of the popularity of the 6502, many books have been written on how to program in 6502 assembly language.

With so many books on 6502 assembly language programming already available, you might be tempted to ask why another book is needed. That's easy. Few of these books are machine specific, and even fewer were written especially for the Apple computer. In addition, while these books can be helpful in learning the basics of assembly language programming and familiarizing you with the various op codes and their mnemonics, they fall short when it comes to supplying the reader with hard information on how to perform specific tasks in assembly language.

This should not be your first book

This book is designed to pick up where the others leave off. Most of the books that currently exist are designed to be used as a first book in assembly language programming. This book is designed as a second book. This means that the book was written with several assumptions in mind.

First, it is assumed that you have already read one of the existing books that teach 6502 assembly language and that you are familiar with the mnemonics. Another assumption that is made is that you have, or have access to, an Apple computer and know how to operate it. Finally, it is desirable that you have an assembler to use with your Apple.

What is an assembler?

For those of you who are not familiar with what an assembler is or does, it is a program that allows you to write other programs using the assembly-language mnemonics. Of course, it's possible to write the program out on paper, convert the op codes to their hexadecimal equivalents and either enter the program from the monitor, or POKE it into memory from BASIC, but that is a cumbersome and time consuming way of doing things. By using an assembler program, we let the computer do all of the hard work. In addition, we gain a lot of flexibility as well as the

ability to make changes easily. Generally, an assembler consists of two major parts:

(1) an editor that allows you to enter and manipulate your program listing and descriptive comments, and

(2) a translator that converts the mnemonic codes to machine code (hexadecimal numbers) and stores the resulting machine language program in memory, or on tape or disk.

Some assemblers contain a third part, a printer module, that allows you to print out the program that you entered with mnemonics along side of the machine-language translation of the mnemonics. However, most assemblers build this capability into the translator module.

The various modules of the assembler can all be in memory at the same time (coresident), or they can be loaded in separately as needed. The coresident assembler has the advantage that it works faster. There are probably at least a dozen assemblers available for the Apple computer, but three of the best are the S-C Macro Assembler from S-C Software, Merlin from Southwestern Data Systems and Big MAC from Call A.P.P.L.E., which is only slightly less powerful than Merlin (they were written by the same person), but is a lot less expensive. The programs in this book were all written with the S-C Macro Assembler.

For those of you who are not too familiar with assemblers, I will explain just a few of the features of the S-C Assembler that are used here. These may differ slightly in the way they are implemented on other assemblers. To begin with, there are pseudo op codes, which are really instructions to the assembler itself. All pseudo op codes begin with a period. Some of the pseudo op codes that are used in these programs are:

.OR means ORigin and it tells the assembler where the program that is being assembled is designed to run in memory. If this location does not conflict with memory locations used by the assembler, as the program is assembled, the object code (program) it produces is stored at this location. If no origin address is specified, it is assumed to be $800.

.TA means Target Address and defines where the program code will be stored as it is generated by the assembler. This pseudo op code must be used when the origin of the program conflicts with the memory locations used by the assembler. In practice, after the code has been assembled, it must by moved, with the Apple's block memory move command, to the location in which it is designed to work. If no target address is specified, it is assumed to be the same as the origin address.

.EQ means EQuate and is used to assign a value to a label. This value may be a single or a double byte and it may represent an address or data.

.AS means ASCII String and is used to store the binary value of the ASCII characters that follow it. The string itself must be enclosed in delimiters that the user can define. I have chosen to use quotation marks for these delimiters. If the first delimiter is preceded by a minus sign, the hexadecimal code generated will

have the high bit set (which is used throughout this book). If the minus sign is not present the high bit will not be set.

.HS means Hex String and is used to enter hexadecimal data, such as may be found in conversion, or address tables. It assumes the presence of two digits for every byte.

.DA means DAta and is used to define constants and/or variables.

Frequently in assembly language programs, it is necessary to find the address of a labelled subroutine. In the programs listed in this book, to define the low byte of the address the pound sign (#) is used and to define the high byte, the slash (/) is used. Thus, if COUT equals $FDED, #COUT will return the value $ED, while /COUT will return the value $FD. One final comment, all lines that start with an asterisk (*) are considered comment lines by the assembler, and are ignored by it.

As a matter of convention, in this book all hexadecimal (hex) numbers will be preceded by a dollar ($) sign. Thus $10 is a hexadecimal number which is equal to 16 in decimal and 10 (without the dollar sign) is the decimal number ten.

Chapter 2

GETTING INFORMATION OUT OF YOUR COMPUTER

Newcomers to assembly language programming often find that printing out text from an assembly language program is difficult and inconvenient do to. Consequently, they frequently resort to combining machine language and BASIC programs together so that BASIC can handle the message printing. However, by developing some standard message printing routines in assembly language, you will find that it is just as easy to print text from assembly language as it is from BASIC.

To give you an idea of just how easy things can be, take a look at the program listing for the SIMPLE MESSAGE PRINTER. The actual program itself (from $800 to $80D) is only 14 bytes long. The bulk of the memory occupied by this routine is for the text itself ($80E to $838) which is 43 bytes long, including the terminating zero byte.

How the SIMPLE MESSAGE PRINTER works

The program starts out by initializing the Y-register to zero in line 1160. This is used as a pointer to the next character and is incremented by one (line 1200) each time a character is printed. The character to be printed is fetched when the instruction in line 1170 is performed. Here, the program is telling the computer to go to the location to which the label TEXT has been assigned. Now, add the value that is in the Y-register to this address and load the character that is located at this new address into the accumulator. This method of loading the accumulator is known as Indexed Addressing.

To see how this works, let's take a look at an example. In this program, when the value in the Y-register is 2, the character that is loaded into the accumulator is 'T' whose hexadecimal equivalent is D4. This is because TEXT = $80E and it begins with two carriage returns (the .HS 8D8D in line 1240). When $2 is added to $80E, the result is $810. Looking at the listing you can see that the character located at $810 is D4 or a 'T'.

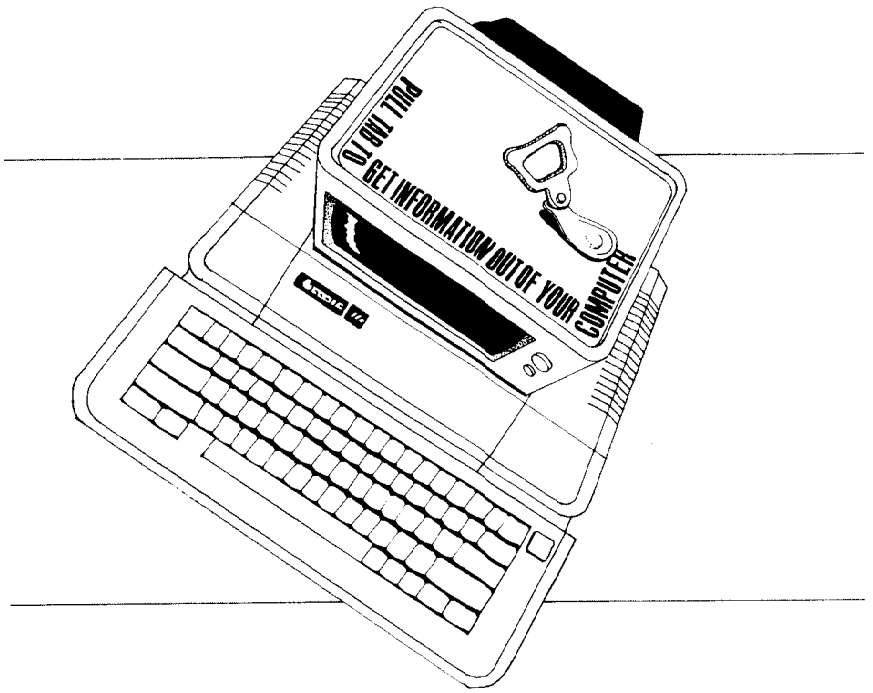
After the character is loaded into the accumulator, a check is made in line 1180 to see if the character was a zero. If it was, this is a sign that the end of the text has been reached and the program then branches, without printing, to label ENDPRT where an RTS instruction (return from subroutine) is executed, and control is returned to the calling program or mode.

If the character was not a zero byte, then the COUT ($FDED) subroutine in the Apple's ROM is called to print out the character in the accumulator. Upon returning from that subroutine call, the Y-register is incremented by one (line 1200) and a check is made to see if the value in the Y-register passed 255 and returned to zero. If it hasn't, and it shouldn't, the program branches back to line 1170 and the next character is fetched.

The message to be printed starts at line 1240 and ends in line 1260 with a BRK instruction. The BRK was used because when it is assembled it generates a zero byte. As an alternative .HS 00 could have been used to generate the required zero byte. The message begins with two carriage returns, followed by the text listed in line 1250.

Pseudo op codes tell the assembler what to do

Looking carefully at lines 1240, 1250 and also at line 1120, you will notice the pseudo op codes that we spoke about earlier. These commands do not appear in the final assembled program. They merely contain instructions to the assembler to perform certain functions. In line 1120 the .EQ pseudo op code tells the assembler to assign the address $FDED to the label COUT. In line 1240, the .HS pseudo op code tells the assembler that all the data that follows should be considered hexadecimal data.



The .AS pseudo op code in line 1250 tells the assembler that the information that follows, is an ASCII string (text). Most assemblers require that the text be enclosed by delimiters (quotation marks, slashes, etc). This assembler has an additional feature in that it allows you decide whether or not you want the high bit of the character set or not. This is done by the presence or absence of a hyphen, or minus sign, (-) after the .AS pseudo op code and before the quotation mark. If the hyphen is present, the high bit is set, if it is absent, the high bit remains a zero.

About high bits and ASCII code

Numbers, letters and certain standard symbols can be represented in the computer by a special code known as ASCII (for American Standard Code for Information Interchange - see Appendix A). This code uses 7 bits to code 128 numbers, letters and symbols. Since there are 8 bits in a byte, there's one extra bit left over.

The Apple computer uses the eighth (or high) bit to determine whether or not the character displayed on a video screen will be displayed normally or in a flashing mode. If the high bit is not set, the character will flash, if it is set it will be displayed

```
                1000 ************************************
                1010 ***                                 ***
                1020 ***     SIMPLE MESSAGE PRINTER      ***
                1030 ***                                 ***
                1040 ************************************
                1050 *
                1060 *
                1070 *
                1080 *
                1090 *
                1100 * EQUATES
                1110 *
FDED-           1120 COUT    .EQ $FDED
                1130 *
                1140 *
                1150 *
0800- A0 00     1160         LDY #$0       Initialize pointer
0802- B9 0E 08  1170 LOOP    LDA TEXT,Y    Get character
0805- F0 06     1180         BEQ ENDPRT    Done yet?
0807- 20 ED FD  1190         JSR COUT      No, print character
080A- C8        1200         INY           Increment pointer
080B- D0 F5     1210         BNE LOOP      Get next character
080D- 60        1220 ENDPRT  RTS           Return to caller
                1230 *
080E- 8D 8D     1240 TEXT    .HS 8D8D
0810- D4 C8 C9
0813- D3 A0 C9
0816- D3 A0 D4
0819- C8 C5 A0
081C- D3 C1 CD
081F- D0 CC C5
0822- A0 CD C5
0825- D3 D3 C1
0828- C7 C5 A0
082B- D4 CF A0
082E- C2 C5 A0
0831- D0 D2 C9
0834- CE D4 C5
0837- C4        1250         .AS -"THIS IS THE SAMPLE MESSAGE TO BE PRINTED"
0838- 00        1260         BRK
```


normally. A bit is said to be set, or on, when its value is equal to 1 and reset, or off, when its value is equal to 0.

Improving the SIMPLE MESSAGE PRINTER

While the SIMPLE MESSAGE PRINTER can be easily used to output text from an assembly language program, it does have a major drawback. The program is what can be referred to as an in-line routine, meaning that every time you want to print out a message, you have to add another 13 bytes (most of the time you won't need the RTS instruction at the end) to your program for the printing routine. While that might not seem like a lot, you'd be surprised at just how quickly that adds up.

A more reasonable way to do things is to convert the program into a subroutine that can be jumped to whenever it is needed. That is exactly what has been done in the IMPROVED MESSAGE PRINTER. As you can see by glancing at the listing, the program has been broken down into two major parts: the main program and the message printing subroutine.

How the IMPROVED MESSAGE PRINTER works

The main program illustrates how the printing subroutine is called. In line 1200 the low-order byte of the address of the label TEXT is loaded into the accumulator, while in line 1210, the high-order byte of the address of the TEXT label is loaded into the Y-register. Now that the program knows where the message that we want to print is located in memory, all it has to do is jump to the message printing subroutine which begins on line 1280.

The first thing that the message printing subroutine does is to store the address of the text to be printed in a two-byte pointer (low byte first) on zero page. Once that is done, the Y-register is reset to zero, so that it can be used as a pointer (or index) to the next character.

One important point you should realize when using this subroutine is that what ever is in the accumulator and the Y-register before you use this routine will be destroyed. So if you need that information, you should store it in a temporary location until you exit the printing routine and then load the values back into their respective places.

The type of indexed addressing (line 1310) used in the subroutine is slightly different from that used in the previous program. This method of loading the accumulator is known as Indirect Indexed Addressing, also sometimes referred to as Post-Indexing. In this mode, the computer goes to the location indicated by TXTPTR, which has been defined in line 1130 as location $06 on page zero of memory, and looks in locations $06 and $07 for the address of the text to be printed. Once it has this address, it adds to it the value stored in the Y-register to get the real address that is desired and then loads the accumulator with the information from that address. The rest of the program is identical to that of the previous one. A

test is made for a zero byte and if none is found, the character is printed and the next character is retrieved.

This program is quite useful and as you progress through this book, you will find that it has been used extensively as a subroutine in other programs. The subroutine itself is 17 bytes long and requires 7 bytes of code to set up the call to the subroutine. So, it is easy to see that if you have more than 2 messages to print in a program, it pays to use this program rather than the former one.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    IMPROVED MESSAGE PRINTER  ***
                1030 ***                              ***
                1040 ************************************
                1050 *
                1060 *
                1070 *
                1080 *
                1090 *
                1100 *
                1110 * EQUATES
                1120 *
0006-           1130 TXTPTR .EQ $06
FDED-           1140 COUT   .EQ $FDED
                1150 *
                1160 *
                1170 * This is the main program, which calls
                1180 * the message printing subroutine.
                1190 *
0800- A9 19     1200        LDA #TEXT       Get address low byte.
0802- A0 08     1210        LDY /TEXT       Get address high byte.
0804- 20 08 08  1220        JSR MSGPRT      Print text.
0807- 60        1230        RTS
                1240 *
                1250 *
                1260 * This is the message printing routine.
                1270 *
0808- 85 06     1280 MSGPRT STA TXTPTR      Store pointer
080A- 84 07     1290        STY TXTPTR+1    to text.
080C- A0 00     1300        LDY #$0         Init counter.
080E- B1 06     1310 LOOP   LDA (TXTPTR),Y  Get character.
0810- F0 06     1320        BEQ ENDPRT      Done yet?
0812- 20 ED FD  1330        JSR COUT        No, print character.
0815- C8        1340        INY             Increment counter.
0816- D0 F6     1350        BNE LOOP        Get next character.
0818- 60        1360 ENDPRT RTS             Return to caller.
                1370 *
0819- 8D 8D     1380 TEXT   .HS 8D8D
081B- D4 C8 C9
081E- D3 A0 C9
0821- D3 A0 D4
0824- C8 C5 A0
0827- D3 C1 CD
082A- D0 CC C5
082D- A0 CD C5
0830- D3 D3 C1
0833- C7 C5 A0
0836- D4 CF A0
0839- C2 C5 A0
083C- D0 D2 C9
083F- CE D4 C5
0842- C4        1390        .AS -"THIS IS THE SAMPLE MESSAGE TO BE PRINTED"
0843- 00        1400        BRK
```


Printing very long messages

As I indicated earlier, you will find that the IMPROVED MESSAGE PRINTER can be used for most of your text output applications. It is possible however, that under certain circumstances, you will find that not all of your text is being printed and there is no apparent reason for it. It's not really as mysterious as it may seem, because the two printing routines that we have discussed until now have had one thing in common, they are limited to a maximum message length of 255 characters. The reason for this can be found in lines 1340 and 1350 of the IMPROVED MESSAGE PRINTER program.

In line 1340, the Y-register is incremented. The INY instruction affects the Zero (or Z) flag bit in the status register, and if the INY operation results in the Y-register being set equal to zero, the Z flag is set. If the Y-register does not become zero, the Z flag is reset. In line 1350, the BNE instruction is used to see if the Y-register has been incremented past 255 and returned to zero (remember the Y-register is an 8-bit register and can only hold values up to 255).

In most cases, the Y-register never gets to zero and the message printing subroutine is terminated instead by the zero that follows the text. But for text containing more than 255 characters, the terminating zero that follows the text is never reached and instead, the Y-register becomes zero and terminates the routine. You can check this out yourself by simply using the previous program and putting in a message that is longer than 255 characters.

To overcome this size limitation, instead of using the single byte Y-register as the text pointer, we must use a two-byte pointer to the text. Such a pointer will technically enable us to print out up to 65,536 characters. In real life, we must leave some space in memory for the program and various parts of the Apple's operating system. But in essence, a two-byte pointer will let us print out messages of virtually any length.

The changes required to accommodate a two-byte pointer can be seen in the listing for Long Message Printer No. 1. The method used to call the printing subroutine (starting at line 1200) remains the same as that for the previous program, as does most of the remainder of the program. The only difference is that the INY in line 1340 of the previous program has been replaced by three lines of code that increment TXTPTR instead of the Y-register.

Line 1340 increments the low byte of TXTPTR, while line 1350 checks if this incrementing has caused this low byte to increment past 255 and back to zero. If it has, then the high-order byte, TXTPTR + 1, is also incremented by one. In any case, after adjusting TXTPTR, the program jumps back to LOOP in line 1310 where the next character is fetched.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***   LONG MESSAGE PRINTER NO. 1  ***
                1030 ***      FOR GROUPED MESSAGES    ***
                1040 ***                              ***
                1050 ************************************
                1060 *
                1070 *
                1080 *
                1090 *
                1100 *
                1110 * EQUATES
                1120 *
0006-           1130 TXTPTR .EQ $06
FDED-           1140 COUT   .EQ $FDED
                1150 *
                1160 *
                1170 * This is the main program, which calls
                1180 * the message printing subroutine.
                1190 *
0800- A9 1E     1200        LDA #TEXT
0802- A0 08     1210        LDY /TEXT
0804- 20 08 08  1220        JSR MSGPRT
0807- 60        1230        RTS
                1240 *
                1250 *
                1260 * This is the messag^ printing routine.
                1270 *
0808- 85 06     1280 MSGPRT STA TXTPTR     Save address of TEXT in
080A- 84 07     1290        STY TXTPTR+1   TXTPTR and TXTPTR+1.
080C- A0 00     1300        LDY #$0        Initialize offset to zero.
080E- B1 06     1310 LOOP   LDA (TXTPTR),Y Get next character to print.
0810- F0 0B     1320        BEQ ENDPRT     Done yet?
0812- 20 ED FD  1330        JSR COUT       No, print character.
0815- E6 06     1340        INC TXTPTR     Increment TXTPTR low byte.
0817- D0 F5     1350        BNE LOOP       If not zero get next character.
0819- E6 07     1360        INC TXTPTR+1   Otherwise increment TXTPTR+1.
081B- D0 F1     1370        BNE LOOP       Get next character.
081D- 60        1380 ENDPRT RTS            Return to caller.
                1390 *
081E- 8D 8D     1400 TEXT   .HS 8D8D
0820- D4 C8 C9
0823- D3 A0 C9
0826- D3 A0 D4
0829- C8 C5 A0
082C- D3 C1 CD
082F- D0 CC C5
0832- A0 CD C5
0835- D3 D3 C1
0838- C7 C5 A0
083B- D4 CF A0
083E- C2 C5 A0
0841- D0 D2 C9
0844- CE D4 C5
0847- C4        1410        .AS -"THIS IS THE SAMPLE MESSAGE TO BE PRINTED"
0848- 00        1420  BRK
```

Another way to print long messages

For those of you who firmly believe that "Variety is the spice of life", we have another method of printing out long messages. This one has a little different structure than all of the previous programs. Whereas former programs looked at the label associated with the text to be printed and passed its location to the printing subroutine, this program doesn't even require the message to have a label. I've only left it in for purposes of continuity.


```
                1000 ************************************
                1010 ***                              ***
                1020 ***   LONG MESSAGE PRINTER NO. 2  ***
                1030 ***      FOR IN-LINE MESSAGES    ***
                1040 ***                              ***
                1050 ************************************
                1060 *
                1070 *
                1080 *
                1090 *
                1100 * EQUATES
                1110 *
0006-           1120 TXTPTR .EQ $06
FDED-           1130 COUT   .EQ $FDED
                1140 *
                1150 *
                1160 *
                1170 * This is the main program, which calls
                1180 * the message printing subroutine.
                1190 *
0800- 20 2F 08  1200        JSR MSGPRT     Print message that follows.
0803- 8D 8D     1210 TEXT   .HS 8D8D
0805- D4 C8 C9
0808- D3 A0 C9
080B- D3 A0 D4
080E- C8 C5 A0
0811- D3 C1 CD
0814- D0 CC C5
0817- A0 CD C5
081A- D3 D3 C1
081D- C7 C5 A0
0820- D4 CF A0
0823- C2 C5 A0
0826- D0 D2 C9
0829- CE D4 C5
082C- C4        1220        .AS -"THIS IS THE SAMPLE MESSAGE TO BE PRINTED"
082D- 00        1230        BRK            End of message marker.
082E- 60        1240        RTS
                1250 *
                1260 *
                1270 * This is the message printing routine.
                1280 *
082F- 68        1290 MSGPRT PLA            Pull address of TEXT-1
0830- 85 06     1300        STA TXTPTR     off the stack and save it
0832- 68        1310        PLA            in TXTPTR and TXTPTR+1.
0833- 85 07     1320        STA TXTPTR+1
0835- A0 01     1330        LDY #$01       Set Y-register to 1.
0837- B1 06     1340 LOOP   LDA (TXTPTR),Y Get next character to print.
0839- F0 09     1350        BEQ ENDPRT     Done yet?
083B- 20 ED FD  1360        JSR COUT       No, print character.
083E- 20 4D 08  1370        JSR INCPTR     Increment TXTPTR by 1.
0841- 4C 37 08  1380        JMP LOOP       Get the next character.
                1390 *
                1400 *
                1410 * The end of the text has been reached
                1420 * so increment TXTPTR twice to get the
                1430 * correct address to return to.
                1440 *
0844- 20 4D 08  1450 ENDPRT JSR INCPTR
0847- 20 4D 08  1460        JSR INCPTR
084A- 6C 06 00  1470        JMP (TXTPTR)
                1480 *
                1490 *
                1500 * This is where TXTPTR is incremented.
                1510 * First the low byte is incremented and
                1520 * if it passes zero as it's incremented,
                1530 * then the high byte is incremented too.
                1540 *
084D- E6 06     1550 INCPTR INC TXTPTR
084F- D0 02     1560        BNE RETURN
0851- E6 07     1570        INC TXTPTR+1
0853- 60        1580 RETURN RTS
```

At first glance, the operation of the program is unclear and it even looks like it is going to crash right after it returns from its jump to the message printing subroutine, because it looks like it is going to try to execute the text as machine language instructions. Let me assure you this is not going to happen.

An interesting way to use the stack

Whenever the 6502 microprocessor executes a JSR instruction, as it does in line 1200, the address minus one, of the next instruction to be executed is pushed onto the stack (which takes up page 1 of memory). Data are pushed onto the stack starting at $1FF and work their way down to $100. When the JSR in line 1200 is executed, the microprocessor doesn't know that what follows the JSR is not another instruction, but just data, so it automatically pushes the address of TEXT-1 onto the stack. The address is pushed onto the stack high byte first, low byte last.

The first thing that the message printing subroutine does is to pull the address off the stack, low byte first (line 1290) and store it in TXTPTR and TXTPTR + 1. To compensate for the minus 1, TXTPTR could either be incremented or the Y-register can be set to 1 instead of 0, which is what was done here (line 1330). This will not pose us any problems later on because the Y-register always remains the same. Only TXTPTR and TXTPTR + 1 get incremented.

In lines 1340 to 1360 the program gets the next character, checks to see if the end of the message has been reached and prints out the character if it hasn't. In line 1370 the program jumps to a subroutine that increments TXTPTR and TXTPTR + 1 if necessary. After that, the program goes back to get the next character.

When the program does detect the end of message marker (the zero byte) it branches to ENDPRT in line 1450 where TXTPTR is incremented twice. It is incremented once to get past the BRK instruction, to which it is pointing as it enters ENDPRT, and incremented a second time to compensate for the -1 associated with the original address of TEXT. After incrementing it twice, therefore, TXTPTR is pointing to the instruction immediately following the BRK. This turns out to be the RTS instruction in line 1240. So on exiting ENDPRT, the program does an indirect jump through TXTPTR (line 1470) to return to its proper place in the program.

Decimal numbers can be output too

Until now, we've seen how we can print out textual information. But what do we do if we want to print out some numbers that were generated by our machine language program and reside in memory in a hexadecimal form? If we wanted to print out the number in hexadecimal, all we'd have to do is to load the byte(s) into the accumulator and then jump to the PRBYTE routine in the Apple monitor ROM, located at $FDDA. But, if we want to print the hexadecimal number out as a decimal number, which most of us are more familiar with, then we have to do some sort of number conversion. Both the 6502 microprocessor and the Apple system are very versatile, and you will quickly realize that there is more than one way to write an assembly language program. To illustrate this point, the next three program will all perform the same task: printing out the decimal equivalent of a two-byte hexadecimal number. If you look carefully at the previous sentence, you'll notice that I indicated that the task was printing out the decimal equivalent and not necessarily converting to the decimal equivalent. The distinction will be made clear shortly when we look at the first of the three programs.

The heart of this first program is a short routine that Steve Wozniak, one of the founders of Apple, wrote a few years ago and was published in the San Francisco Apple Core's Cider Press Magazine. Normally, when converting an integer from one base to another, the integer is repeatedly divided by the desired base. The remainder of each division becomes successively more significant digits of the answer. The process continues until the base can no longer be divided into the argument. To illustrate how this works let's convert 32 in decimal to its hexadecimal equivalent.

$$32/16 = 2 \text{ with a remainder } R = 0$$

$$2/16 = 0 \text{ and } R = 2$$

Before you get excited and say that 2/16 is .125, remember that we are dealing with integer numbers only, no fractions. So if a result is less than 1, it's set equal to zero and a remainder. Earlier we said that as the division progresses, the remainders become successively more significant digits of the answer. This means that the last remainder (2) is the most significant digit of the answer. Hence, 32 decimal is equal to $20 hexadecimal.

The process works in the reverse direction just as well. Let's convert $20 hex back to its decimal equivalent.

$$\$20/\$A = \$3 \text{ and } R = 2$$

$$\$3/\$A = \$0 \text{ and } R = 3$$

Here we divide $20 by the base we wish to convert to, which is 10 decimal or $A in hexadecimal. Once again, by taking the last remainder as our most significant digit and reading back we find that $20 hex is equal to 32 decimal, which is really no great surprise.

A shorter and faster conversion method can be implemented on microprocessors that have a decimal mode, such as the 6502. In this program, the two-byte hex number that is to be converted is stored on page zero in locations $50 and $51 which are known as LINNUM and LINNUM + 1. The answer, in binary coded decimal (BCD) form is stored in location TEMP and the two locations that follow it. These are located from $6 to $8 on page 0 of memory. TEMP contains the lowest order digit and TEMP + 2 the highest order.

For those of you who are unfamiliar with just what a binary coded decimal is, let me explain. If from BASIC you typed POKE 0,32 and then you went into monitor mode and looked at location 0, you'd find the hexadecimal number $20 there,

which we already know is the hex equivalent of 32. However, if I had a program that converted $20 to decimal and stored the digits 3 and 2 as a single byte in a single memory location, I would have a binary coded decimal. So, if from the monitor you were to type 0:32 and then a press RETURN, you could say that you stored 32 as a binary coded decimal into location zero. Having done this, you can now use the Apple's monitor routine PRBYTE to print the byte out to the screen.

Thus if you load the accumulator with 32 and do a JSR to PRBYTE ($FDDA), the number 32 will appear on you screen. You can see therefore, that we can convert a number to BCD and then use the PRBYTE routine to display it. As far as the viewer is concerned, he is seeing a decimal number, even though if he looked in memory he would actually see BCD numbers. The advantage to using BCD numbers is that they require less memory. A 5-digit decimal number requires 5 memory locations, one for each digit. The same number in BCD form only requires 2.5 memory locations because 2 digits are packed into every byte. Thus the number 65535 would be represented in BCD as the three bytes 06 55 35. In our program, these numbers are stored in memory in reverse order: 35 55 06.

In the program, OUTPUT A DECIMAL NUMBER #1, the section of code from 1280 to 1450 converts the two-byte hex number in LINNUM to its BCD equivalent. Lines 1280 to 1300 clear the two low order bytes of the answer. The high order byte does not have to be cleared because any data stored there will be shifted out automatically during the calculation. In line 1310, a flag is initialized to zero. The flag will be used to determine whether or not a zero that is to be printed out is a leading zero. This is done to enable us to suppress leading zeroes so we don't get 065535 instead of 65535, which is what we really want.

The next thing that is done is to switch the 6502 into its decimal arithmetic mode in line 1320. In line 1330, we are setting up a loop that will be performed 16 ($10) times. Within this loop, the numbers in LINNUM and LINNUM + 1 will be shifted left, pushing the most significant bit into the carry. Then, the values in TEMP, TEMP + 1 and TEMP + 2 are doubled and the carry is added to them.

The low and middle order bytes are doubled by adding each byte to itself (lines 1360 to 1410). The high order byte is doubled by shifting its contents left once (line 1420). By doing this, there is no need to initialize TEMP + 2 to zero at the beginning, because the original contents will be shifted out during execution.

This entire process (lines 1340 to 1440) is performed 16 times to convert both hex bytes into 5 BCD digits. When the calculations are done, it is very important to return the 6502 microprocessor to its hexadecimal calculation mode by executing the CLD (clear decimal mode) instruction. Otherwise the remainder of the program will not work properly.

Once the conversion has been completed, the program then proceeds to print out the numbers. Since most of us are not used to seeing numbers with leading zeroes, I've included routines that check to see if a zero that is a candidate for being printed is a leading zero and if it is, to skip it and get the next digit.

The routine starting at line 1580 sets up a loop that retrieves the three BCD bytes. As a byte is loaded into the accumulator (line 1600), a check is made to see if its value is zero. If it is, a second test is performed to see if this is the first digit to be printed. If it is the first digit, LZFLAG will be 0 otherwise it will contain some nonzero value. If a zero byte is the first byte to be printed, the byte is discarded (line 1630) and the program jumps back to line 1590 to get the next byte.

If the whole byte is not equal to zero, a test is made (at lines 1720 and 1730) to see if the most significant digit (nibble) of the byte is zero. (NOTE: This will always be the case with the byte at TEMP + 2.) If it's not, then LZFLAG is set to indicate that a digit has already been printed, the complete original byte is retrieved (we had to modify it to do our test) and the byte is printed.

On the other hand, if the most significant nibble of the byte is zero, then the program jumps to line 1990 to find out if a byte has already been printed. If one has then this one is also printed. If nothing has been printed yet, the original byte (with its leading zero) is retrieved and stored in LZFLAG to make it nonzero, and then the right most, or least significant digit of the byte is printed using the PRHEX routine in the Apple ROM. Finally, in line 2100 the V flag of the status register is cleared and in line 2110 a branch on V clear instruction is executed. This causes the program to branch back to line 1820 and check to see if there are anymore digits to be printed.

Branching instead of jumping

Instead of using the CLV and BVC op codes in 2100 and 2110, we could simply have put in a JMP instruction. However, I wanted you to see how it's possible to implement a function — branch always — that does not exist in the 6502. Other microprocessors, such as the 6800 and the 65C02, have a BRA instruction which unconditionally branches to the desired location.

In the Apple, the V flag of the status register is very rarely used, so it's generally fairly safe to clear it and then execute a BVC instruction. Here's the listing of the program we have been discussing.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***  OUTPUT A DECIMAL NUMBER # 1  ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 * EQUATES
                1120 *
0006-           1130 TEMP   .EQ $6
0009-           1140 LZFLAG .EQ $9
0050-           1150 LINNUM .EQ $50
FDDA-           1160 PRBYTE .EQ $FDDA
FDE3-           1170 PRHEX  .EQ $FDE3
                1180 *
                1190 *
                1200 * This section of code converts a
                1210 * 2 byte unsigned binary argument in
                1220 * LINNUM and LINNUM+1 to a binary coded
                1230 * decimal number packed into 3 adjacent
                1240 * locations starting at TEMP, low byte
```

```
                1250 * first.  This conversion routine was
                1260 * written by Steve Wozniak.
                1270 *
0800- A9 00     1280        LDA #$0
0802- 85 06     1290        STA TEMP        Clear result
0804- 85 07     1300        STA TEMP+1
0806- 85 09     1310        STA LZFLAG      Clear leading 0 flag.
0808- F8        1320        SED             Set decimal mode.
0809- A0 10     1330        LDY #$10        Set for 16 bits
080B- 06 50     1340 LOOP   ASL LINNUM      Shift bit out
080D- 26 51     1350        ROL LINNUM+1    of binary argument.
080F- A5 06     1360        LDA TEMP
0811- 65 06     1370        ADC TEMP
0813- 85 06     1380        STA TEMP
0815- A5 07     1390        LDA TEMP+1      Double decimal
0817- 65 07     1400        ADC TEMP+1      result and add carry.
0819- 85 07     1410        STA TEMP+1
081B- 26 08     1420        ROL TEMP+2      Shift last bit
081D- 88        1430        DEY
081E- D0 EB     1440        BNE LOOP        Repeat 16 times.
0820- D8        1450        CLD             Clear decimal mode.
                1460 *
                1470 *
                1480 * This section contains the loop that
                1490 * fetches each of the 3 bytes that
                1500 * contain the packed binary-coded
                1510 * decimal number and checks to see if
                1520 * both numbers in the byte are zero.
                1530 * If they are, a further check is made
                1540 * to see if this is the first byte to
                1550 * be printed, in which case the whole
                1560 * byte is discarded.
                1570 *
0821- A2 03     1580        LDX #$3         Count 3 bytes.
0823- CA        1590 NEXT   DEX
0824- B5 06     1600        LDA TEMP,X      Get a byte.
0826- D0 06     1610        BNE CHKLDO      Check for leading zero.
0828- C5 09     1620        CMP LZFLAG      Yes, is it the first?
082A- F0 F7     1630        BEQ NEXT        Yes, discard.
082C- D0 08     1640        BNE PRINT2      No, print the byte.
                1650 *
                1660 *
                1670 * This section checks to see if the
                1680 * byte being processed contains a
                1690 * leading zero.
                1700 *
082E- 48        1710 CHKLDO PHA             Save the accumulator.
082F- 29 F0     1720        AND #$F0        Leading zero?
0831- F0 0B     1730        BEQ LEAD0       If zero, process it.
0833- 85 09     1740        STA LZFLAG      It's not so set flag.
0835- 68        1750 PRINT1 PLA             Restore accumulator.
0836- 20 DA FD  1760 PRINT2 JSR PRBYTE      Print byte in accumulator.
                1770 *
                1780 *
                1790 * Here the program checks to see if
                1800 * there is anymore data to output.
                1810 *
0839- E0 00     1820 CHKDON CPX #$0
083B- D0 E6     1830        BNE NEXT
083D- 60        1840        RTS
                1850 *
                1860 *
                1870 * This routine checks to see if the
                1880 * byte containing the leading zero is
                1890 * the first byte to be output.  If it
                1900 * is it throws away the zero and prints
                1910 * a single digit.  If it isn't, it
                1920 * restores the byte (which has been
                1930 * destroyed by the testing) and prints
                1940 * it out.  The leading zero flag is
                1950 * also set here so that the program
                1960 * will know it doesn't have to worry
                1970 * about them any more.
```



```
                1980 *
083E- A5 09     1990 LEAD0  LDA LZFLAG      First digit?
0840- D0 F3     2000        BNE PRINT1      No, print it.
0842- 68        2010        PLA             Yes, set flag.
0843- 85 09     2020        STA LZFLAG
                2030 *
                2040 *
                2050 * This section takes a byte with a
                2060 * leading zero and prints it out as a
                2070 * single digit without the leading zero
                2080 *
0845- 20 E3 FD  2090        JSR PRHEX       Print 1 digit
0848- B8        2100        CLV             Relative jump
0849- 50 EE     2110        BVC CHKDON      always taken.
```

As I mentioned earlier, while the previous program will print out decimal numbers to the screen, it doesn't actually generate them as five individual bytes. In some cases, it is desirable to generate the ASCII equivalent of each of the individual digits. To produce numbers on the Apple in normal mode, the digits should be in the $B0 to $B9 range (for 0 to 9).

By using the same conversion routine we used in the previous program, we can quickly write a new program that will generate the ASCII code for each individual digit. The first part of this new program (lines 1290 to 1460) is identical to the routine in lines 1280 to 1450 of the previous program. After the conversion to a binary-coded decimal has been made, all we have to do is retrieve the individual digits that have been packed into three bytes starting at TEMP, and OR them with the hex value $B0 to make them ASCII. This is what happens starting at line 1550 in the second program that outputs decimal numbers.

Separating the nibbles

Indexed addressing with the X-register (line 1560) is used to retrieve the BCD data, least significant byte first. The first thing that is done is to separate the two digits that have been combined to form a single byte, into individual bytes. In line 1570 the least significant digit of the byte is extracted by zeroing out the most significant digit. So, if the BCD value of a byte was $13 and we ANDed it with $0F, we'd get:

$$\begin{array}{l} 00010011 = \$13 \\ \underline{00001111 = \$0F} \\ 00000011 = \$03 \end{array}$$

This value is then ORed in line 1580 with $B0 to produce the ASCII value and the newly converted digit is temporarily stored on the stack (line 1590) until all digits have been processed and we're ready to print them.

The next thing to do is to retrieve the high order digit of the same byte. So, we reload that byte into the accumulator (line 1600) and then perform the logical shift right (LSR) instruction four times (lines 1610 to 1640). What this does is to move the most significant digit of a byte into the least significant position while at the same time storing a zero in the most significant digit. So, for the same byte containing the binary coded number $13, we get:

Separating the nibbles of a byte.

1st shift 2nd shift 3rd shift 4th shift

00010011 → 00001001 → 00000100 → 00000010 → 00000001

Once this operation is completed, that value in the accumulator is ORed with $B0 (line 1650) and so another ASCII digit is created and temporarily stored on the stack. This operation continues until all 6 digits (including the leading zero) in the three bytes are converted.

After all of the ASCII numbers have been stored on the stack, they are pulled off one at a time (line 1790), a check is made to see if the number is a leading zero and if it's not the number is printed using the Apple's standard output routine COUT ($FDED). After all of the numbers have been pulled off the stack and printed, the program executes an RTS instruction, returning control to the calling program or mode.


```
                1000 ************************************
                1010 ***                              ***
                1020 ***  OUTPUT A DECIMAL NUMBER # 2  ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 * EQUATES
                1120 *
0006-           1130 TEMP    .EQ $6
0009-           1140 LZFLAG  .EQ $9
0018-           1150 YSAVE   .EQ $18
0050-           1160 LINNUM  .EQ $50
FDED-           1170 COUT    .EQ $FDED
                1180 *
                1190 *
                1200 * This section of code converts a
                1210 * 2-byte unsigned binary argument in
                1220 * LINNUM and LINNUM+1 to a binary coded
                1230 * decimal number packed into 3 adjacent
                1240 * locations starting at TEMP, low byte
                1250 * first.  This conversion routine was
                1260 * written by Steve Wozniak.
                1270 *
                1280 *
0800- A9 00     1290         LDA #$0
0802- 85 06     1300         STA TEMP        Clear result
0804- 85 07     1310         STA TEMP+1
0806- 85 09     1320         STA LZFLAG      Clear leading 0 flag.
0808- F8        1330         SED             Set decimal mode.
0809- A0 10     1340         LDY #$10        Set for 16 bits
080B- 06 50     1350 LOOP    ASL LINNUM      Shift bit out
080D- 26 51     1360         ROL LINNUM+1    of binary argument.
080F- A5 06     1370         LDA TEMP
0811- 65 06     1380         ADC TEMP
0813- 85 06     1390         STA TEMP
0815- A5 07     1400         LDA TEMP+1      Double decimal
0817- 65 07     1410         ADC TEMP+1      result and add carry.
0819- 85 07     1420         STA TEMP+1
081B- 26 08     1430         ROL TEMP+2      Shift last bit
081D- 88        1440         DEY
081E- D0 EB     1450         BNE LOOP        Repeat 16 times.
0820- D8        1460         CLD             Clear decimal mode.
                1470 *
                1480 *
                1490 * This section of code converts the
                1500 * packed binary-coded decimal number
                1510 * into ASCII characters (low order byte
                1520 * first) and stores them temporarily on
                1530 * the stack.
                1540 *
0821- A2 00     1550         LDX #$0
0823- B5 06     1560 NEXT    LDA TEMP,X      Get byte and
0825- 29 0F     1570         AND #$0F        mask off 4 MSB
0827- 09 B0     1580         ORA #$B0        make it ASCII.
0829- 48        1590         PHA             Save on stack.
082A- B5 06     1600         LDA TEMP,X      Get same byte
082C- 4A        1610         LSR             and move 4 MSB
082D- 4A        1620         LSR             to 4 LSBs.
082E- 4A        1630         LSR
082F- 4A        1640         LSR
0830- 09 B0     1650         ORA #$B0        Make it ASCII.
0832- 48        1660         PHA             Save on stack.
0833- E8        1670         INX
0834- E0 03     1680         CPX #$3         Done yet?
0836- D0 EB     1690         BNE NEXT        No, get more.
                1700 *
                1710 *
```

```
                1720 * This section of code pulls the
                1730 * converted ASCII digits off the stack
                1740 * and prints them.  In doing this it
                1750 * checks for leading zeroes and
                1760 * discards them.
                1770 *
0838- A0 06     1780          LDY #$6         Set for 5 numbers.
083A- 68        1790 PRINT1   PLA              Get a number.
083B- C9 B0     1800          CMP #$B0        Is it a zero?
083D- D0 0A     1810          BNE PRINT2      No, print it.
083F- A6 09     1820          LDX LZFLAG      Yes, is it first 0?
0841- D0 06     1830          BNE PRINT2      No, print it.
0843- 88        1840          DEY             Is it the last number?
0844- D0 F4     1850          BNE PRINT1      No, throw it away.
0846- 4C ED FD  1860          JMP COUT        Yes, print it.
0849- 85 09     1870 PRINT2   STA LZFLAG     Set leading zero flag.
084B- 20 ED FD  1880          JSR COUT        Print the number.
084E- 88        1890          DEY             Done yet?
084F- D0 E9     1900          BNE PRINT1      No, get next number.
0851- 60        2000          RTS             Return to caller.
```

Use the ROMs to help print decimals

Now that you've seen how to convert hexadecimal numbers to decimal numbers the hard way, let me show you a much easier way to do it, and it only takes up seven bytes of memory. You all know that Applesoft is capable of taking a two-byte hexadecimal number and printing out its decimal equivalent. It's done all the time when you list an Applesoft program, because the line numbers of a program are stored as two hex bytes. Now, if we could find some way to use the routines that Applesoft uses, we could save a lot of time and effort.

It turns out that the task is really quite simple. In the Applesoft ROMs, at location $ED24, is the start of a routine called LINPRT. What this routine does, is take the data that are stored in the accumulator and the X-register, and convert them to decimal and print them. So, if we use the same convention that we have used in the previous examples, and store the two bytes of the number to be converted in LINNUM and LINNUM + 1, all our program has to do is load the most significant byte into the accumulator and the least significant byte into the X-register. Then all that's left to do is jump to the LINPRT routine.

If you look at line 1230 closely, you will see that the instruction is a JMP and not a JSR. At this point you might well be asking yourself, what happens after the numbers are printed out? Where does control return to? To answer the question, control is returned to the original mode or routine that called the decimal printing program to begin with. The reason is, that at the end of the LINPRT routine is an RTS. If our program had a JSR instead of a JMP, LINPRT would have returned control to our program, where we would simply have executed an RTS to return to the caller. We can save that extra byte required by the RTS in our program, by simply letting the RTS in the LINPRT routine return control to the caller.

While this method of printing decimal numbers is the simplest, it's not always the best because the minute you use the LINPRT routine you are limiting your program to running only on machines that have Applesoft in ROM or on the language card. Your program will not run on an Integer machine. Worse than that, in machines that have both Integer BASIC and Applesoft, if you use this program, you have to make sure that the Applesoft ROMs have been turned on. So, this approach to printing decimal numbers can only safely be used on Applesoft only machines, unless your program specifically turns on the Applesoft ROMs.


```
                1000 ************************************
                1010 ***                              ***
                1020 ***  OUTPUT A DECIMAL NUMBER # 3  ***
                1030 ***                              ***
                1040 ************************************
                1050 *
                1060 *
                1070 * EQUATES
                1080 *
0050-           1090 LINNUM .EQ $50
ED24-           1100 LINPRT .EQ $ED24
                1110 *
                1120 *
                1130 * This subroutine is entered with the
                1140 * hexadecimal number to be printed in
                1150 * LINNUM (low byte) and LINNUM+1 (high
                1160 * byte).  LINPRT is an Applesoft
                1170 * routine that converts the data in
                1180 * the X-register and the accumulator
                1190 * to decimal and prints it.
                1200 *
0800- A5 51     1210          LDA LINNUM+1
0802- A6 50     1220          LDX LINNUM
0804- 4C 24 ED  1230          JMP LINPRT
```

Applying a number printing routine

Now that we have learned several ways to print out a decimal number from a hexadecimal number, let's see how we can apply what we've learned to a handy little utility program. It is frequently desirable, useful or necessary to know how many lines are contained in an Applesoft program. There are several alternatives. You can print out a listing of the program and count the lines manually, you can renumber the program starting with one, in increments of one, or you can run this short APPLESOFT LINE COUNTER program. The easiest by far is the last.

To understand how this program works, you should first know how Applesoft stores a program line in memory. Let's take a simple line such as the following:

10 PRINT 123

If we were to look directly into memory, we'd see that this line is stored in the following way:

Address	801	802	803	804	805	806	807	808	809
Contents	0A	08	0A	00	BA	31	32	33	00

Looking at locations **$801 and $802** we see two numbers $0A and $08 which comprise the hex number **$80A. In 6502** microprocessor systems, numbers are

always stored in memory with the low-order byte first, followed by the high-order byte, hence $080A or $80A. This number, represents the location in memory of the start of the next line in the Applesoft program. So if we were to add another line to our program, it would start at $80A. Thus, the first two bytes of any Applesoft program are called the "next line pointer".

The next two bytes at $803 and $804 hold the hexadecimal equivalent of the line number. Since our line number is less than 255, only the low-order byte is used (it's set to $0A which equals 10 in decimal). The high order byte is set to zero. Next, on the fifth byte ($805) we have the start of our program. You will notice that $805 contains the value $BA, which is a code that represents the word PRINT. In order to conserve memory space, the programmers who wrote Applesoft decided to take all the Applesoft keywords and assign each of them a one-byte code. Thus, every time a word such as PRINT is used, it's only necessary to store the one-byte code instead of the five letters that make up the word PRINT. By the way, these special codes are called 'tokens'. For a complete list of tokens and their decimal and hexadecimal equivalents, see Appendix B.

Following the PRINT token we have the value $31 stored in $806. If we check our chart of ASCII equivalents (Appendix A) we see that $31 is the hexadecimal equivalent of the number 1. Similarly $32 and $33 that are in locations $807 and $808, represent the numbers 2 and 3. Finally we see that location $809 contains a zero. This zero is what is called an end of line marker. It tells the Applesoft interpreter that there is no more information on the current line and that it should get ready for the next line.

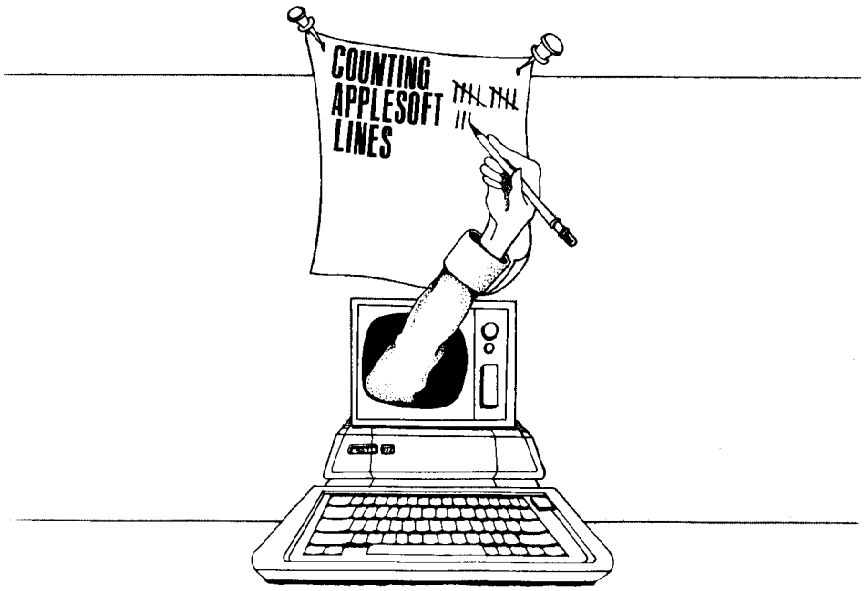
Now we have almost all of the information we need to understand this next program. We just need one more piece of data, "How does the Applesoft interpreter know when it has reached the end of the program?" The answer is simple. It follows the end of line indicator of the last line in the program with two more zeros. So, in our example above, if line 10 were the only line in our program, locations $80A and $80B would contain zeros instead of a pointer to the next program line.

Counting Applesoft program lines

The line counting program starts out by clearing the screen, printing out the program title and copyright notice and prints out the first half of the message that tells the user how many lines are in the Applesoft program. This program then goes on to count the number of lines. It starts by storing zeros in the two locations that are going to hold the line count (lines 1380 to 1400).

A pointer to the start of an Applesoft program is stored in locations $67 and $68. Generally it is set to $801, but it can change, so we pick it up instead of assuming it is $801. This is done in lines 1410 and 1420 and this information is stored in POINTER and POINTER + 1 (lines 1430 and 1440).

The next part of the program consists of a loop that examines the next line pointers of each Applesoft line and looks for a next line pointer that is equal to zero. This is an indication that the end of the program has been reached. In line



1450, the Y-register is set to zero and in line 1460 the contents of the location pointed to by POINTER plus any offset produced by the Y-register, is loaded into the accumulator. Since POINTER contains the address of where the next Applesoft line is stored in memory, the data that is loaded into the accumulator is the value of the next 'next line pointer'. This information is temporarily stored in location TEMP and TEMP + 1 (the program goes through this loop twice for each new line and increments the offset of the Y-register to 1, hence TEMP + 1).

After the value of the next line pointer has been retrieved and stored in TEMP and TEMP + 1, the value that has been stored in TEMP + 1, which is still in the accumulator, is stored in POINTER + 1 (line 1510) and then temporarily saved in the X-register (line 1520). Next, the low-order byte of the next line pointer (now in TEMP) is transferred to POINTER, completing the updating of POINTER for the next Applesoft line.

Earlier we said that at the end of an Applesoft program, the next line pointer of the last line points to the two zeros that follow the end of line marker of the last line. So, if we test for the presence of the third zero, and it's there, we know that we have reached the end of the program. That's exactly what we do in line 1550. We transferred the high-order byte of the next line pointer to the X-register a few moments earlier. If this is a zero, it would be the third zero and the program would go to lines 1570 and 1580, where the accumulator and the X-register are set up for a JSR to the LINPRT routine, which will print out the number of lines counted (line 1590) and the remainder of the text message (lines 1600 to 1620). Finally, the program executes an RTS which returns control to the caller.

If it turns out that the end of the program has not been reached, the program branches to line 1700 where both bytes of the line count are retrieved and 1 is added to the count with any carry that's generated being added to the high-order byte. After that, the program jumps back to line 1450 to get the address of the next program line.

```
                        1000 ************************************
                        1010 ***                              ***
                        1020 ***    APPLESOFT LINE COUNTER     ***
                        1030 ***                              ***
                        1040 ***     COPYRIGHT (C) 1982 BY    ***
                        1050 ***        JULES H. GILDER       ***
                        1060 ***      ALL RIGHTS RESERVED     ***
                        1070 ***                              ***
                        1080 ************************************
                        1090 *
                        1100 *
                        1110 *
                        1120 *
                        1130 * EQUATES
                        1140 *
0006-                   1150 LINECNT .EQ $6
0008-                   1160 POINTER .EQ $8
0018-                   1170 TXTPTR  .EQ $18
0067-                   1180 TXTTAB  .EQ $67
02F6-                   1190 TEMP    .EQ $2F6
ED24-                   1200 LINPRT  .EQ $ED24
FC58-                   1210 HOME    .EQ $FC58
FDED-                   1220 COUT    .EQ $FDED
FF58-                   1230 RETURN  .EQ $FF58
                        1240 *
                        1250 *
                        1260         .OR $2F8
                        1270 *
                        1280 *
                        1290 * This is the main program where the
                        1300 * program title is printed out and
                        1310 * the lines of the Applesoft program
                        1320 * are counted.
                        1330 *
02F8- 20 58 FC          1340         JSR HOME          Clear screen.
02FB- A9 58             1350         LDA #TEXT1        Point to text
02FD- A0 03             1360         LDY /TEXT1        to be printed.
02FF- 20 47 03          1370         JSR MSGPRT        Print it.
0302- A9 00             1380         LDA #$0           Initialize
0304- 85 06             1390         STA LINECNT       counter to
0306- 85 07             1400         STA LINECNT+1     zero.
0308- A5 67             1410         LDA TXTTAB        Store program
030A- A4 68             1420         LDY TXTTAB+1      starting
030C- 85 08             1430         STA POINTER       address in
030E- 84 09             1440         STY POINTER+1     POINTER.
0310- A0 00             1450 GETADDR LDY #$0           Get address of
0312- B1 08             1460 LOOP1   LDA (POINTER),Y   next line &
0314- 99 F6 02          1470         STA TEMP,Y        save it.
0317- C8                1480         INY
0318- C0 01             1490         CPY #$1           Got high byte?
031A- F0 F6             1500         BEQ LOOP1         No, go get it.
031C- 85 09             1510         STA POINTER+1     Save high byte.
031E- AA                1520         TAX               Prepare for zero test.
031F- AD F6 02          1530         LDA TEMP          Get next line
0322- 85 08             1540         STA POINTER       low byte & save it.
0324- E0 00             1550         CPX #$0           Last line?
0326- D0 0F             1560         BNE ADDCNT        No, increment count.
0328- A5 07             1570         LDA LINECNT+1     Yes, get ready
032A- A6 06             1580         LDX LINECNT       to print count
032C- 20 24 ED          1590         JSR LINPRT        Print it.
032F  A9 BE             1600         LDA #TEXT2        Point to text
0331  A0 03             1610         LDY /TEXT2        to be printed.
```



```
0333- 20 47 03          1620         JSR MSGPRT        Print it.
0336- 60                1630         RTS               Return.
                        1640 *
                        1650 *
                        1660 * This subroutine increments the line
                        1670 * count and then goes back to check for
                        1680 * another line.
                        1690 *
0337- 18                1700 ADDCNT  CLC               Clear carry bit.
0338- A5 06             1710         LDA LINECNT       Get current count low byte.
033A- 69 01             1720         ADC #$1           Add 1 to it.
033C- 85 06             1730         STA LINECNT       Save it.
033E- A5 07             1740         LDA LINECNT+1     Get high byte of count.
0340- 69 00             1750         ADC #$0           Add 0 to add carry.
0342- 85 07             1760         STA LINECNT+1     Save it.
0344- 4C 10 03          1770         JMP GETADDR       Get address of next line.
                        1780 *
                        1790 *
                        1800 * This is the message printing routine.
                        1810 *
0347- 85 18             1820 MSGPRT  STA TXTPTR        Set TXTPTR to address of
0349- 84 19             1830         STY TXTPTR+1      text to be printed.
034B- A0 00             1840         LDY #$0           Zero character counter.
034D- B1 18             1850 LOOP2   LDA (TXTPTR),Y    Get character.
034F- F0 06             1860         BEQ ENDPRT        End if it's zero.
0351- 20 ED FD          1870         JSR COUT          Print character.
0354- C8                1880         INY               Increment character counter.
0355- D0 F6             1890         BNE LOOP2         Get next character.
0357- 60                1900 ENDPRT  RTS               Return to sender.
                        1910 *
                        1920 *
                        1930 *
0358- C1 D0 D0
035B- CC C5 D3
035E- CF C6 D4
0361- A0 CC C9
0364- CE C5 A0
0367- C3 CF D5
036A- CE D4 C5
036D- D2                1940 TEXT1   .AS -"APPLESOFT LINE COUNTER"
036E- 8D 8D             1950         .HS 8D8D
0370- C2 D9 A0
0373- CA D5 CC
0376- C5 D3 A0
0379- C8 AE A0
037C- C7 C9 CC
037F- C4 C5 D2          1960         .AS -"BY JULES H. GILDER"
0382- 8D                1970         .HS 8D
0383- C3 CF D0
0386- D9 D2 C9
0389- C7 C8 D4
038C- A0 A8 C3
038F- A9 A0 B1
0392- B9 B8 B2          1980         .AS -"COPYRIGHT (C) 1982"
0395- 8D                1990         .HS 8D
0396- C1 CC CC
0399- A0 D2 C9
039C- C7 C8 D4
039F- D3 A0 D2
03A2- C5 D3 C5
03A5- D2 D6 C5
03A8- C4                2000         .AS -"ALL RIGHTS RESERVED"
03A9- 8D 8D 8D
03AC- 8D                2010         .HS 8D8D8D8D
03AD- D4 C8 C5
03B0- A0 D0 D2
03B3- CF C7 D2
03B6- C1 CD A0
03B9- C8 C1 D3
03BC- A0                2020         .AS  "THE PROGRAM HAS "
03BD- 00                2030         .HS 00
03BE- A0 CC C9
03C1- CE C5 D3
```

```
03C4- A0 C9 CE
03C7- A0 C9 D4
03CA- AE        2040 TEXT2    .AS -" LINES IN IT."
03CB- 8D 00     2050          .HS 8D00
```

Using the Applesoft line counter

The program has been assembled starting at location $2F8 so that it can be loaded into an area of memory that is not affected by Applesoft. The program can be loaded before or after an Applesoft program has been loaded into memory. To run the program it is simply necessary to type CALL 760.

Once loaded, the program will remain in memory available for use whenever you need it. There is one exception to this. Since the program starts at $2F8, it uses the last 8 bytes of the input buffer. This was done because assembling the program at $300, which is what is normally done, would cause the program to wipe out some memory locations that are used by DOS.

Very rarely is the entire input buffer filled, so this doesn't usually pose a problem. On top of that, Applesoft limits line lengths to 239 characters, much less than the 256 character capacity of the buffer. Nevertheless, if for some reason the input buffer is filled up completely (256 characters are entered before a carriage return is pressed), part of the program will be wiped out and it will have to be reloaded. After considerable use however, this problem has never occurred.

Drawing boxes and borders on the screen

Now that we've learned how to print out text and numerical data to the screen, let's see how we can come up with a way of making our screen look a little more attractive. One way of doing this is to use a border around the whole screen, or a box around just a portion of it.

Most programmers don't take the time to develop a border printing routine and thus when they need to draw one, usually wind up doing it in a very inefficient manner. The routine presented here is a simple one, and not very long. Nevertheless, it is quite a versatile routine, and by changing only four parameters you can completely change the size and shape of the box, as well as the symbol used to draw it.

The program starts out by clearing the screen in line 1250. If you want to enclose some text within the box, the routine to do it can be inserted here, or you can position the cursor to the spot you'll want to start printing at after the box is drawn. Next, the current position of the cursor is saved (lines 1260 to 1290) so that the cursor can be restored to its position after the border has been drawn. The routine that draws the border starts at line 1360 where the cursor is positioned to the top left-hand corner of the screen (lines 1360 to 1390). Next, the program jumps to line 1580 where it prints out a full line of symbols (the first line of the box).

To find out how many blank lines there will be inside the box, the program goes

to the location labeled BOXLEN and stores the number found there in the X-register. The blank lines with left and right borders on them are printed next. Lines 1430 to 1450 print the left-hand border of the blank line, while lines 1460 to 1490 print the right-hand border. Line 1490 checks to see if all the blank lines have been printed and if so, line 1500 finishes printing out the right-hand border of the last blank line. Then the program falls into the LINSYM routine which prints out the bottom line of the box. After this last line is printed, the program returns to line 1310, where the program then jumps to a routine that restores the cursor's original position. This cursor restoring routine starts at line 1690. The program ends on line 1740 where a return from this whole program is executed.

Constants that are used by the program are stored starting at line 1800 where the the number of blank lines within the box are stored. In line 1810 the location where symbols on the left side of a blank line stop is stored, while the start of symbols on the right side of a line are stored in line 1820. Finally, the symbol used to draw the box is stored in line 1830. Try running this program and varying the constants. You'll be pleased and surprised at the results.

```
                1000 ****************************************
                1010 ***                                  ***
                1020 ***            TITLE BOX             ***
                1030 ***                                  ***
                1040 ****************************************
                1050 *
                1060 *
                1070 *
                1080 *
                1090 * EQUATES
                1100 *
0018-           1110 CV2      .EQ $18
0019-           1120 CH2      .EQ $19
0024-           1130 CH       .EQ $24
0025-           1140 CV       .EQ $25
FC22-           1150 BASCAL   .EQ $FC22
FC58-           1160 HOME     .EQ $FC58
FDED-           1170 COUT     .EQ $FDED
                1180 *
                1190 *
                1200 *
                1210 * Clear the screen, and save the
                1220 * current location of the cursor for
                1230 * later.
                1240 *
0800- 20 58 FC  1250          JSR HOME        Clear the screen.
0803- A5 25     1260          LDA CV          Save the current
0805- A4 24     1270          LDY CH          cursor position.
0807- 85 18     1280          STA CV2
0809- 84 19     1290          STY CH2
080B- 20 11 08  1300          JSR BOX         Draw the box.
080E- 4C 40 08  1310          JMP POSCUR      Restore old cursor position.
                1320 *
                1330 *
                1340 * Print a box on the screen.
                1350 *
0811- A9 00     1360 BOX      LDA #$0         Place cursor
0813- 85 25     1370          STA CV          at the start
0815- 85 24     1380          STA CH          of the first
0817  20 22 FC  1390          JSR BASCAL      line.
081A  20 35 08  1400          JSR LINSYM      Print a line of symbols.
081D  AE 4C 08  1410          LDX BOXLEN      Set box depth
0820  20 ED FD  1420 NEXT     JSR COUT        Print left
0823  A4 24     1430          LDY CH          side of box.
0825  CC 4D 08  1440          CPY LFTMRG      Done?
```

```
0828- D0 F6     1450         BNE NEXT        No, do more.
082A- AC 4E 08  1460         LDY RTMRG       Print right
082D- 84 24     1470         STY CH          side of box.
082F- CA        1480         DEX             End of box?
0830- D0 EE     1490         BNE NEXT        No, do more.
0832- 20 35 08  1500         JSR LINSYM      Yes, finish.
                1510 *
                1520 *
                1530 * This subroutine prints out a line of
                1540 * symbols.  It checks CH to see if
                1550 * it has past the 40th column and
                1560 * wrapped around to column 0.
                1570 *
0835- AD 4F 08  1580 LINSYM LDA SYMBOL       Get the symbol to be used.
0838- 20 ED FD  1590 PRTSYM JSR COUT         Print it.
083B- A4 24     1600         LDY CH          Get horizontal position.
083D- D0 F9     1610         BNE PRTSYM      If not zero, print again.
083F- 60        1620         RTS             Return to caller.
                1630 *
                1640 *
                1650 * This subroutine restores the cursor
                1660 * to its original position before the
                1670 * box was drawn.
                1680 *
0840- A5 19     1690 POSCUR LDA CH2          Get original cursor
0842- A4 18     1700         LDY CV2         position.
0844- 85 24     1710         STA CH          Save in proper locations.
0846- 84 25     1720         STY CV
0848- 20 22 FC  1730         JSR BASCAL      Send cursor there.
084B- 60        1740         RTS             Return to caller.
                1750 *
                1760 *
                1770 * These are constants that are used by
                1780 * the program.
                1790 *
084C- 15        1800 BOXLEN .HS 15           Number of lines in box.
084D- 01        1810 LFTMRG .HS 01           End of symbols on left side.
084E- 27        1820 RTMRG  .HS 27           Start of symbols on right side.
084F- AA        1830 SYMBOL .HS AA           Character used to draw border.
```

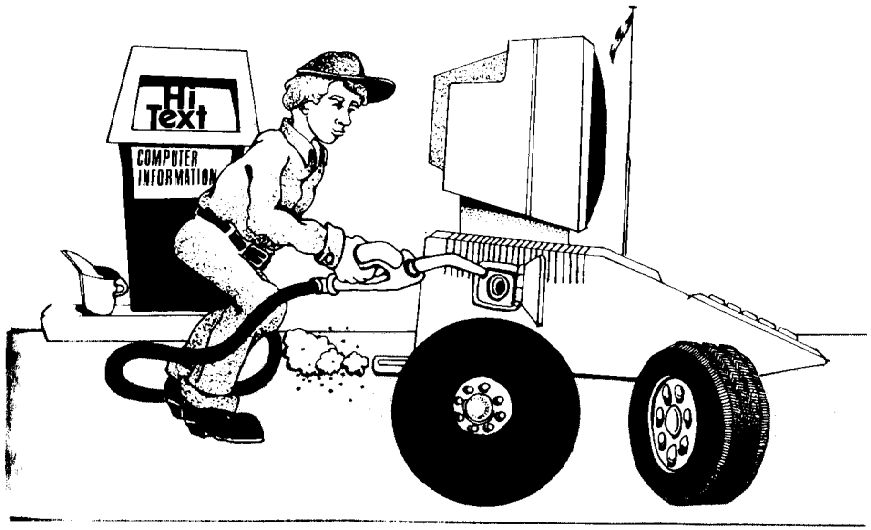
Chapter 3

GETTING INFORMATION INTO YOUR COMPUTER

You can write a lot of useful assembly language programs that only use the computer's output capabilities, but sooner or later, you're going to want to be able to input data while your program is running. Getting information into your Apple is not difficult at all as you can tell by looking at the fairly short program listing for the Simple Read Keyboard Routine.

One of the things that makes it easy to input data is the configuration of hardware in the Apple computer. Apple's designer's arranged things so that the keyboard looked like a particular memory location. So, by looking at the right place in memory, we can see if a key has been pressed and determine exactly which key it was.

As it turns out, if you look at location $C000 you can see if a key has been pressed. As long as no key is pressed, any value that is retrieved from location $C000 will be less than 128. When a key is pressed, $80 is added to the ASCII value



of the key pressed and that value remains in location $C000 until a command to clear that location is given or another key is pressed.

In our program, the memory location associated with the keyboard is read in line 1180 and in line 1190 a test is made to see if a key was pressed by checking bit 7 of the byte retrieved from $C000. If bit 7 is zero, the keyboard is read again until it has changed to 1. Once we've loaded the accumulator with the character input from the keyboard, we should clear this memory location, otherwise the next time we check to see if a key has been pressed, we'll get an indication that it has, even if it hasn't, and get the last character that was entered. In order to clear this memory location, it is only necessary to zero out bit seven of the data stored in $C000, since this will make any value stored there less than 128.

The Apple hardware has been arranged in a special way so that it is possible to turn off bit 7 by simply accessing another memory location: $C010. If this location is accessed in any way with an LDA, STA or BIT instruction, bit 7 in $C000 will be converted from a 1 to a 0. Location $C010 is called by a special name, Keyboard Strobe, and in our program, it is activated in line 1200 with a BIT instruction. We could just as easily have used an LDA instruction to achieve the same results. Some programmers use an STA instruction to clear bit 7, and while this will work, it can be a problem on those Apples that have been modified to include a keyboard buffer. The reason for this is that the STA instruction actually references the location it's storing data to twice. So, with a keyboard buffer and an STA instruction clearing the keyboard strobe twice for every character read, you'll wind up losing every other character. For best results use the LDA or BIT instructions.

After we clear bit 7 of $C000, our program prints out the character to the screen so we can see what letter we pressed (line 1210) and it then jumps back to get another character.

```
                1000 ************************************
                1010 ***                              ***
                1020 *** SIMPLE READ KEYBOARD ROUTINE  ***
                1030 ***                              ***
                1040 ************************************
                1050 *
                1060 *
                1070 *
                1080 *
                1090 *
                1100 * EQUATES
                1110 *
C000-           1120 KEYBRD  .EQ $C000
C010-           1130 KBDSTRB .EQ $C010
FDED-           1140 COUT    .EQ $FDED
                1150 *
                1160 *
                1170 *
0800- AD 00 C0  1180 GETKEY  LDA KEYBRD  Read keyboard
0803- 10 FB     1190         BPL GETKEY  If no key pressed, read again.
0805- 2C 10 C0  1200         BIT KBDSTRB Key pressed, clear strobe.
0808- 20 ED FD  1210         JSR COUT    Echo character to screen.
080B- 4C 00 08  1220         JMP GETKEY  Get next character.
```

This program works fine for very short keyboard entries, but becomes inconvenient to use for long entries. To begin with, this program doesn't print any prompt character, so you don't know where the text entry on the screen is required. In addition, the program doesn't allow for any way of terminating text input except by pressing RESET.

A better way to read the keyboard

The problems encountered with the previous program can be eliminated by taking advantage of one of the monitor ROM routines and making a small change in the program. Instead of having our program look at the keyboard directly, we can use the RDKEY routine (line 1170) in the ROM, at location $FD0C, to do that job for us. This routine puts a flashing cursor on the screen at the location where an input is expected, reads the keyboard location ($C000) and clears the strobe ($C010).

To allow us to terminate the input of data we can designate a special character as the terminator and test for its presence. In this case, the ESCape character ($9B) is used. Line 1180 checks to see if an ESCape has been entered. If it has, the program returns to the calling mode or program, if not, the character is printed out and a new character is fetched.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***          IMPROVED            ***
                1030 ***     READ KEYBOARD ROUTINE    ***
                1040 ***                              ***
                1050 ************************************
                1060 *
                1070 *
                1080 *
                1090 *
                1100 * EQUATES
                1110 *
FD0C-           1120 RDKEY   .EQ $FD0C
FDED-           1130 COUT    .EQ $FDED
                1140 *
                1150 *
                1160 *
0800- 20 0C FD  1170 GETKEY  JSR RDKEY      Read the keyboard.
0803- C9 9B     1180         CMP #$9B       Was key pressed ESC?
0805- F0 06     1190         BEQ QUIT       Yes, quit program.
0807- 20 ED FD  1200         JSR COUT       No, print key pressed.
080A- 4C 00 08  1210         JMP GETKEY     Get the next key.
080D- 60        1220 QUIT    RTS            Return to caller.
```

Both of the previous routines input text one character at a time and neither allows you to make corrections on inputted data. The reason you can't make corrections is that the text being entered is not stored in any buffer before it is processed. If it were, then if an error were caught it could be corrected while it was still in the buffer and before it was processed.

Entering text a line at a time

By taking advantage of another routine in the monitor ROM (GETLN which is located at $FD6A) we can input text into the input buffer on page 2 of memory

($200 to $2FF) and use all of the Apple's normal editing capabilities. As long as you don't press the RETURN key, it is possible to backspace and change any character and then copy over the rest of the line.

This type of program comes in particularly handy when you want the user to enter some text that is going to be printed out again later under program control. The reason is, it stores the entered text in memory the same way text that is used with the MSGPRT routine is stored. That is, it's stored with the high bit set and is terminated by a zero. One place where you'll find this routine a must is when you ask the user for the name of a file to be loaded or saved to. After the user inputs that name, it must be stored for later use.

The GETLN routine at $FD6A prints out the prompt that is currently stored in $33 before it waits for the user's input. More often than not, you'll want to ask for the user's input without using this prompt, as is the case here. To do this, another entry point into this routine, which I call GETLN1 and is located at $FD6F, is used (line 1180). Upon returning from GETLN1, the corrected text that the user entered is stored in the input buffer. It must be moved from there immediately (lines 1200 to 1240) because it could get wiped out by the next data that are entered. The end of the data in the input buffer is indicated by a carriage return ($8D). Since we want our text to be terminated by a zero and not a carriage return, the carriage return is replaced by a zero and stored at the end of the text in the user designated buffer (lines 1250 to 1260).

```
                1000 ************************************
                1010 ***                              ***
                1020 ***       TEXT INPUT ROUTINE     ***
                1030 ***                              ***
                1040 ************************************
                1050 *
                1060 *
                1070 *
                1080 *
                1090 * EQUATES
                1100 *
0200-           1110 IN      .EQ $200
0300-           1120 BUFFER  .EQ $300
FD6F-           1130 GETLN1  .EQ $FD6F
FDED-           1140 COUT    .EQ $FDED
                1150 *
                1160 *
                1170 *
0800- 20 6F FD  1180         JSR GETLN1     Get a line of text, no prompt.
0803- A0 FF     1190         LDY #$FF       Initialize character
0805- C8        1200 LOOP    INY            counter to zero.
0806- B9 00 02  1210         LDA IN,Y       Get a character
0809- 99 00 03  1220         STA BUFFER,Y   and store it in buffer.
080C- C9 8D     1230         CMP #$8D       Is it a carriage return?
080E- D0 F5     1240         BNE LOOP       No, get next character.
0810- A9 00     1250         LDA #$0        Yes, make it a zero
0812- 99 00 03  1260         STA BUFFER,Y   to indicate end of text.
0815- 60        1270         RTS            Return to caller.
```

Entering as much text as you want

You will find the TEXT INPUT ROUTINE a useful program to use when it is necessary to enter a line of text. It does have the limitation, however, that you cannot enter more than 256 characters with it. The reason is that the Y-register is used as the pointer from a base address to where the next character is to be stored.

If you want to be able to store unlimited amounts of text into memory (up to the capacity of your computer that is) then the IMPROVED TEXT INPUT ROUTINE is just what the doctor ordered. This program uses a 2 byte pointer on page zero called BUFPTR to indicate the location of the next character to be stored.

The IMPROVED TEXT INPUT ROUTINE is fairly similar the previous program, with a few exceptions. It takes text out of the input buffer every time the carriage return is pressed and stores everything, including the carriage return, in the user defined buffer. Then the program goes back and gets another line of text. This continues until the program encounters a Control-Q (line 1520), at which point it replaces the Control-Q with a zero and exits to the last active BASIC.

Because this program uses the input buffer to enter text, a maximum of 255 characters can be entered before a carriage return must be pressed. Chances of having a single line that is greater than 255 characters are small, so this should not pose any problem. If you will want to print this text out again, it will be necessary to use one of the long message printing routines that were discussed in the last chapter. And in fact, if you combine this program, with one of those, you have two major parts of a rudimentary text editor. By doing a little additional work to develop an in-memory byte editor, it's possible to write a simple text editor.

You should notice that this program uses the IMPROVED MESSAGE PRINTER that was discussed in the last chapter. Since this routine is only being used to print out a few short messages, its limitation to 256 characters is not a problem. Wherever possible throughout this book, you will find that previously developed programs are used as subroutines.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***  IMPROVED TEXT INPUT ROUTINE  ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130 * EQUATES
                1140 *
0006-           1150 BUFPTR  .EQ $6
0008-           1160 TXTPTR  .EQ $8
0200-           1170 IN      .EQ $200
03D0-           1180 WARMDOS .EQ $3D0
9000-           1190 BUFFER  .EQ $9000
E000-           1200 BASIC   .EQ $E000
FC58            1210 HOME    .EQ $FC58
FD0C            1220 RDKEY   .EQ $FD0C
FD6F            1230 GETLN1  .EQ $FD6F
FDED            1240 COUT    .EQ $FDED
                1250 *
                1260 *
                1270 * Print out the title and copyright
                1280 * notice and wait for the user to press
                1290 * any key.
```

```
                    1300 *
0800- 20 58 FC      1310          JSR HOME          Clear screen.
0803- A9 58         1320          LDA #TEXT         Get text to be
0805- A0 08         1330          LDY /TEXT         printed.
0807- 20 47 08      1340          JSR MSGPRT        Print it.
080A- 20 0C FD      1350          JSR RDKEY         Get key press.
                    1360 *
                    1370 *
                    1380 * Clear the screen and start getting
                    1390 * text from the keyboard, one line at
                    1400 * a time.  Store text in buffer area.
                    1410 *
080D- 20 58 FC      1420          JSR HOME
0810- A9 00         1430          LDA #BUFFER       Get location
0812- A0 90         1440          LDY /BUFFER       of text buffer
0814- 85 06         1450          STA BUFPTR        and store in
0816- 84 07         1460          STY BUFPTR+1      buffer pointer
0818- 20 6F FD      1470 START    JSR GETLN1        Input a line.
081B- A0 00         1480          LDY #$0
081D- A2 00         1490          LDX #$0
081F- BD 00 02      1500 LOOP1    LDA IN,X          Get input and
0822- 91 06         1510          STA (BUFPTR),Y    save in buffer.
0824- C9 91         1520          CMP #$91          End of input?
0826- F0 0E         1530          BEQ ENDIT         Yes, finish up
0828- E8            1540          INX               No, increment
0829- E6 06         1550          INC BUFPTR        indices and
082B- D0 02         1560          BNE NEXT          get more data.
082D- E6 07         1570          INC BUFPTR+1
082F- C9 8D         1580 NEXT     CMP #$8D          Carriage return?
0831- D0 EC         1590          BNE LOOP1         No, get a character.
0833- 4C 18 08      1600          JMP START         Yes, new line.
0836- A9 00         1610 ENDIT    LDA #$0           Done, store a
0838- 91 06         1620          STA (BUFPTR),Y    zero as the last byte.
083A- AD D0 03      1630          LDA WARMDOS       Check if DOS is
083D- C9 4C         1640          CMP #$4C          present.
083F- D0 03         1650          BNE NODOS         It's not, return via BASIC.
0841- 4C D0 03      1660          JMP WARMDOS       It is, return through DOS.
0844- 4C 03 E0      1670 NODOS    JMP BASIC+3       Jump to BASIC warm start.
                    1680 *
                    1690 *
                    1700 * This is the message printing routine.
                    1710 *
0847- 85 08         1720 MSGPRT   STA TXTPTR
0849- 84 09         1730          STY TXTPTR+1
084B- A0 00         1740          LDY #$0
084D- B1 08         1750 LOOP2    LDA (TXTPTR),Y
084F- F0 06         1760          BEQ ENDPRT
0851- 20 ED FD      1770          JSR COUT
0854- C8            1780          INY
0855- D0 F6         1790          BNE LOOP2
0857- 60            1800 ENDPRT   RTS
                    1810 *
0858- C9 CD D0
085B- D2 CF D6
085E- C5 C4 A0
0861- D4 C5 D8
0864- D4 A0 C9
0867- CE D0 D5
086A- D4 A0 D2
086D- CF D5 D4
0870- C9 CE C5      1820 TEXT     .AS -"IMPROVED TEXT INPUT ROUTINE"
0873- 8D 8D         1830          .HS 8D8D
0875- C2 D9 A0
0878- CA D5 CC
087B- C5 D3 A0
087E- C8 AE A0
0881- C7 C9 CC
0884- C4 C5 D2      1840          .AS -"BY JULES H. GILDER"
0887- 8D            1850          .HS 8D
0888- C3 CF D0
088B- D9 D2 C9
088E- C7 C8 D4
0891- A0 A8 C3
0894- A9 A0 B1
```



```
0897- B9 B8 B2      1860          .AS -"COPYRIGHT (C) 1982"
089A- 8D            1870          .HS 8D
089B- C1 CC CC
089E- A0 D2 C9
08A1- C7 C8 D4
08A4- D3 A0 D2
08A7- C5 D3 C5
08AA- D2 D6 C5
08AD- C4            1880          .AS -"ALL RIGHTS RESERVED"
08AE- 8D 8D 8D
08B1- 8D            1890          .HS 8D8D8D8D
08B2- D0 D2 C5
08B5- D3 D3 A0
08B8- C1 CE D9
08BB- A0 CB C5
08BE- D9 A0 D4
08C1- CF A0 C3
08C4- CF CE D4
08C7- C9 CE D5
08CA- C5            1900          .AS -"PRESS ANY KEY TO CONTINUE"
08CB- 00            1910          .HS 00
```

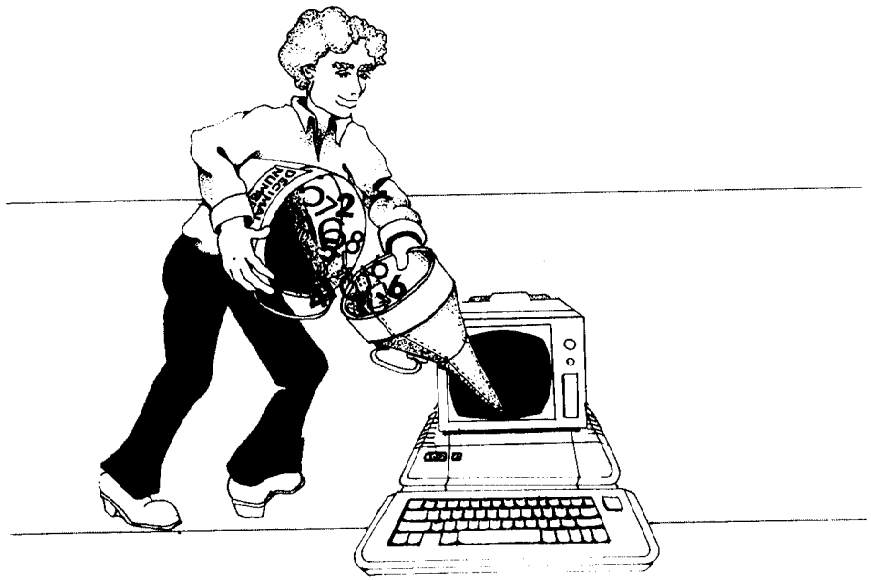
Entering decimal numbers

In the last chapter we saw how it was possible to take hexadecimal numbers and convert them so that they printed out as decimal numbers. Now we're going to do the reverse. We're going to enter decimal numbers (whole integers only) and convert them into hexadecimal numbers that can be used by our program. It is not necessary to use this approach if all you're going to do is enter a single digit, such as a number for a menu selection, because it's easier to check for the number as an ASCII character. But for entering numbers that are going to be used in calculations, you'll need this program.

The program starts our by getting a line of text from the user (line 1330). This line should contain only the decimal digits of the number we want to convert and the number should not contain more than five digits, which is the maximum number of digits that can be represented by two bytes.

The program has some simple error checking built into it. The first thing it does is check to see if a number was entered or the RETURN key was just pressed. If the return key was pressed, the length of the text entered, which is stored in the X-register in the GETLN1 routine, is zero. Since the input of this routine must be at least 1 digit, this generates an error (lines 1340 and 1350). Next is the check for a number that has more than 5 digits and its appropriate error message (lines 1360 and 1370). By the way, the error routine, which begins at line 1970, uses one of the routines in the Apple ROM. This routine, PRERR which is at $FF2D, rings the bell and prints out the message ERR. After an error is detected and the user is informed, he is given an opportunity to start over again (line 1990).

Getting back to our main program, the length of the digit entered is stored in a location called LENGTH (line 1380), for use later on when we want to see if we've processed all digits of the number. Next the two locations that will be used to hold the converted number — LINNUM and LINNUM + 1 — are initialized to zero and one last check is made to make sure that only numbers and no letters or symbols



were entered (lines 1420 to 1470). An error message is generated if anything other than numerals were entered.

Data that are entered via the GETLN1 routine consist of the ASCII code for the character to which $80 has been added. This means that the digits 0 through 9 will appear as $B0 through $B9. If somehow we were able to make the left nibble of the byte equal to zero, we'd have the decimal equivalent of all of the digits in the number. That's exactly what we do in line 1480. This conversion is done within a loop that retrieves one digit at a time and stores it temporarily on the stack (line 1490). Next, the current contents of LINNUM and LINNUM + 1 are multiplied by ten by a routine starting in line 1550 so the digits can be added to each other to build the number (e.g. 1 × 10 + 2 = 12).

The multiplication by ten is accomplished by multiplying by two (lines 1550 and 1560), saving the the results (lines 1580 to 1600) and then multiplying again by four, to get a total multiplication of 8 (lines 1610 to 1640). Then the 8 and 2 multiples are added together to get the final multiple of 10 (lines 1650 to 1700). Finally, the digit that was stored on the stack is retrieved and added to the contents of LINNUM (lines 1710 to 1730). If a carry is generated, it is added to LINNUM + 1 (lines 1740 to 1760). This whole process is carried out until all of the digits of the number that was entered have been processed. When done, the hexadecimal equivalent of the number entered can be found in LINNUM and LINNUM + 1.

If you want to limit your programs to operating on a computer with Applesoft in ROM, then you can use INPUT INTEGER ROUTINE NO. 2 to enter data. This program was originally written by Peter Meyer and was published in S-C Soft-


```
                        1000 ************************************
                        1010 ***                              ***
                        1020 ***  INPUT INTEGER ROUTINE NO. 1  ***
                        1030 ***                              ***
                        1040 ***     COPYRIGHT (C) 1982 BY    ***
                        1050 ***        JULES H. GILDER       ***
                        1060 ***      ALL RIGHTS RESERVED     ***
                        1070 ***                              ***
                        1080 ************************************
                        1090 *
                        1100 *
                        1110 *
                        1120 * EQUATES
                        1130 *
0006-                   1140 LENGTH .EQ $6
0050-                   1150 LINNUM .EQ $50
0200-                   1160 IN     .EQ $200
FD6F-                   1170 GETLN1 .EQ $FD6F
FD8E-                   1180 CROUT  .EQ $FD8E
FF2D-                   1190 PRERR  .EQ $FF2D
                        1200 *
                        1210 *
                        1220 * This section of code handles entry
                        1230 * of the number from the keyboard and
                        1240 * then checks each digit to see that it
                        1250 * is valid.  If a valid digit is found
                        1260 * the 4 most significant bits (MSBs)
                        1270 * are set to zero to get just the digit
                        1280 * by itself.  It also checks to see if
                        1290 * more than 5 digits have been entered.
                        1300 * If an error is detected an error
                        1310 * message is generated.
                        1320 *
0800- 20 6F FD          1330 START  JSR GETLN1      Get a number
0803- E0 00             1340        CPX #$0         Any entry?
0805- F0 49             1350        BEQ ERROR       No, do over.
0807- E0 06             1360        CPX #$6         Is >5 digits?
0809- B0 45             1370        BCS ERROR       Yes, do over.
080B- 86 06             1380        STX LENGTH      Save number of digits.
080D- A9 00             1390        LDA #$0         Initialize
080F- 85 50             1400        STA LINNUM      hex number to
0811- 85 51             1410        STA LINNUM+1    zero.
0813- A0 00             1420        LDY #$0
0815- B9 00 02          1430 LOOP   LDA IN,Y        Get a character.
0818- C9 B0             1440        CMP #$B0        Test to see if
081A- 90 34             1450        BCC ERROR       it is a digit
081C- C9 BA             1460        CMP #$BA        from 0 to 9.
081E- B0 30             1470        BCS ERROR
0820- 29 0F             1480        AND #$0F        Mask out 4 MSBs.
0822- 48                1490        PHA             Save digit
                        1500 *
                        1510 *
                        1520 * This section of code multiplies a
                        1530 * 16-bit number stored in LINNUM by 10.
                        1540 *
0823- 06 50             1550 MULT   ASL LINNUM      Multiply by 2
0825- 26 51             1560        ROL LINNUM+1
0827- A5 51             1570        LDA LINNUM+1    Save number
0829- 48                1580        PHA             multiplied by
082A- A5 50             1590        LDA LINNUM      2 for latter.
082C- 48                1600        PHA
082D- 06 50             1610        ASL LINNUM      Multiply by 4
082F- 26 51             1620        ROL LINNUM+1    to get a total
0831  06 50             1630        ASL LINNUM      multiplication
0833  26 51             1640        ROL LINNUM+1    of 8.
0835  68                1650        PLA             Add the 2 & 8
0836  65 50             1660        ADC LINNUM      multiples to
0838  85 50             1670        STA LINNUM      get a total
083A  68                1680        PLA             multiplication
083B  65 51             1690        ADC LINNUM+1    of 10.
083D  85 51             1700        STA LINNUM+1
083F  68                1710        PLA             Get current
0840  65 50             1720        ADC LINNUM      digit & add it
```

```
0842- 85 50     1730         STA LINNUM      to the partial
0844- A9 00     1740         LDA #$0         sum.
0846- 65 51     1750         ADC LINNUM+1
0848- 85 51     1760         STA LINNUM+1
                1770 *
                1780 *
                1790 * This section checks to see if all of
                1800 * the digits have been processed and if
                1810 * not gets another digit until there
                1820 * are no more.
                1830 *
084A- C8        1840         INY
084B- C4 06     1850         CPY LENGTH      Finished?
084D- D0 C6     1860         BNE LOOP        No, get more.
084F- 60        1870         RTS             Yes, no more.
                1880 *
                1890 *
                1900 * This subroutine rings the bell and
                1910 * prints out the message ERR followed
                1920 * by a carriage return.  Control is
                1930 * then passed back to the beginning of
                1940 * the program so that a valid number
                1950 * can be entered.
                1960 *
0850- 20 2D FF  1970 ERROR   JSR PRERR       Error message
0853- 20 8E FD  1980         JSR CROUT       Output a carriage return.
0856- 4C 00 08  1990         JMP START       Start over.
```

ware's Apple Assembly Line. The program makes extensive use of internal Applesoft routines and will give us an opportunity to see how things are done inside Applesoft. One thing should be pointed out here, and that is that using ROM routines doesn't always save you a lot of memory over writing dedicated routines. If you take a look at the length of this program and at the length of the previous program, you'll see that this one is only 13 bytes shorter than the former.

The first thing that the program does is to input a line of text into the keyboard buffer (lines 1330 and 1340). Once a carriage return has been pressed, the program then checks to see if at least one character was entered. If not, the program jumps to an error routine that sets the carry bit. If a return from this routine has the carry bit clear, the calling program will know that no errors took place. If no error is generated, the program goes on to temporarily save the length of the number entered on the stack while it does a subroutine jump do an Applesoft ROM routine called GDBUFFS.

The GDBUFFS routine, which is located at $D539, puts a zero at the end of the input buffer. It then proceeds to mask off (or zero out) the most significant bit (bit 7) on all bytes in the input buffer. This is equivalent to subtracting $80 from all bytes. The result is that all of the data in the input buffer are in their true ASCII form. Upon returning from GDBUFFS, the length of the number is retrieved from the stack (line 1450) and a check is made to see if more than five digits were entered (line 1460). If so an error is generated and the carry is set. If not, the length is transferred to the X-register, where it is used as an index into the input buffer (lines 1480 to 1500).


```
                1000 ************************************
                1010 ***                              ***
                1020 ***  INPUT INTEGER ROUTINE NO. 2  ***
                1030 ***                              ***
                1040 ***        BY PETER MEYER        ***
                1050 ***   FROM APPLE ASSEMBLY LINES   ***
                1060 ***   PUBLISHED BY S-C SOFTWARE   ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130 * EQUATES
                1140 *
0050-           1150 LINNUM   .EQ $50
009D-           1160 FACEXP   .EQ $9D
00A0-           1170 FACMO    .EQ $A0
00A1-           1180 FACLO    .EQ $A1
00A2-           1190 FACSGN   .EQ $A2
00B7-           1200 CHRGOT   .EQ $B7
00B8-           1210 TXTPTR   .EQ $B8
0200-           1220 IN       .EQ $200
D539-           1230 GDBUFFS  .EQ $D539
EBF2-           1240 QINT     .EQ $EBF2
EC4A-           1250 FIN      .EQ $EC4A
FD75-           1260 NXTCHR   .EQ $FD75
                1270 *
                1280 *
                1290 * This section gets a character from
                1300 * the keyboard and stores it in the
                1310 * input buffer ($200 to $2FF).
                1320 *
0800  A2 00     1330         LDX #$0
0802- 20 75 FD  1340         JSR NXTCHR      Get character, put in buffer.
0805- 8A        1350         TXA             Check for null entry.
0806- F0 27     1360         BEQ ERROR       Null, set carry.
                1370 *
                1380 *
                1390 * This checks for alpha input and also
                1400 * eliminates entries that would cause
                1410 * an overflow condition.
                1420 *
0808  48        1430         PHA             Save length.
0809  20 39 D5  1440         JSR GDBUFFS     Put 0 at end of input buffer.
080C  68        1450         PLA             Retrieve length.
080D  C9 06     1460         CMP #$06        More than 5 digits entered?
080F  B0 1E     1470         BCS ERROR       Yes, set carry.
0811  AA        1480         TAX             No, use length as index.
0812  CA        1490         DEX
0813  BD 00 02  1500 LOOP     LDA IN,X        Get character from buffer.
0816  C9 41     1510         CMP #'A         Is it alpha?
0818  B0 15     1520         BCS ERROR       Yes, set carry.
081A  CA        1530         DEX             No, decrement char. count.
081B  10 F6     1540         BPL LOOP        Get next character.
                1550 *
                1560 *
                1570 * Get the number from the input buffer
                1580 * and load it into the floating point
                1590 * accumulator.
                1600 *
081D  A9 00     1610         LDA #IN         Get address of
081F  A0 02     1620         LDY /IN         input buffer
0821  85 B8     1630         STA TXTPTR      and save it in a
0823  84 B9     1640         STY TXTPTR+1    zero page pointer.
0825  20 B7 00  1650         JSR CHRGOT      Get number from buffer.
0828  20 4A EC  1660         JSR FIN         Put it in floating pt. acc.
                1670 *
                1680 *
                1690 * Check to see if the number is
                1700 * negative.  If it is set the carry bit
                1710 *
082B  A5 A2     1720         LDA FACSGN      See if number is negative.
```

```
082D- 10 02     1730          BPL CHKSIZE    No, check size of number.
082F- 38        1740 ERROR    SEC            Yes, error.
0830- 60        1750          RTS
                1760 *
                1770 *
                1780 * Check to see if the number is too big
                1790 *
0831- A5 9D     1800 CHKSIZE LDA FACEXP
0833- C9 91     1810          CMP #$91
0835- B0 0C     1820          BCS END        Too large.
                1830 *
                1840 *
                1850 * Convert the number, which is now in
                1860 * the floating point accumulator into
                1870 * an integer and store it in LINNUM.
                1880 *
0837- 20 F2 EB  1890          JSR QINT       Integer conversion.
083A- A5 A1     1900          LDA FACLO      Transfer number to LINNUM.
083C- A4 A0     1910          LDY FACMO
083E- 85 50     1920          STA LINNUM
0840- 84 51     1930          STY LINNUM+1
0842- 18        1940          CLC            Value is ok.
0843- 60        1950 END      RTS
```

In the loop starting at line 1500, each of the characters that was entered is checked to see if it is an alpha character. If it is, an error is generated, otherwise the program falls into a routine that takes the number from the input buffer and puts it into the floating point accumulator (line 1610). To do this Applesoft's CHRGOT routine at $B7 on page zero is used. Before jumping to this routine however, it is necessary to set a text pointer that this routine uses to point to the first digit of the number. This is done in lines 1610 to 1640 and CHRGOT is jumped to in line 1650. Finally, a jump is made to another Applesoft routine called FIN, which is located at $EC4A. This routine takes the number retrieved by the CHRGOT routine and converts it to floating point format and places it in the floating point accumulator (line 1660).

Once the number is in the floating point accumulator, two more tests are performed on it, one to check for a negative number (lines 1720 to 1730) and one to check for too large a number (maximum size number is 65535). Finally, if the number entered passes all of these tests, it is converted into an integer number (line 1890) by still another Applesoft routine: QINT which is located at $EBF2. QINT stores the converted number, as a hexadecimal number in two locations of the floating point accumulator: FACLO and FACMO. From there, the number is taken and stored in LINNUM and LINNUM + 1 (lines 1900 to 1930), the carry bit is cleared indicating no errors were encountered.

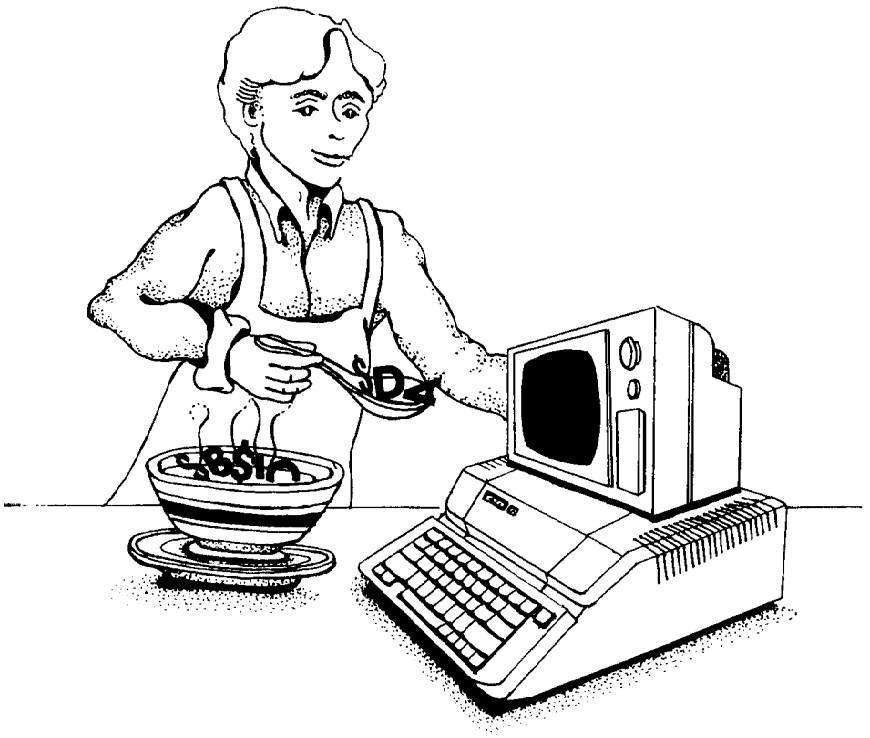
While this program was assembled to operate at $800, it can be loaded as is into any memory range and work properly. This is because there are no absolute jumps to any routines within the program. All jumps are relative branches (e.g. move down 30 locations as opposed to move to location $81E). Thus the program is completely relocatable.

Hexadecimal numbers can be entered too

While most of the number entry your programs will do will probably deal with decimal numbers, occasionally it will be necessary to allow the user to enter hexadecimal numbers as well. The general technique used is similar to the one that we used for entering decimal numbers in the program INPUT INTEGER ROUTINE NO. 1. First a line of text is requested from the user and then it is checked for the proper number of digits. In the case of hexadecimal numbers, we only wish to permit 4 digits, instead of the 5 allowed for decimal. This change is reflected in line 1350.

After the data have been entered, a check is made to see if the characters entered are numbers in the 0 to 9 range, just as was done in the integer program. Next, however, a check is also made to see if any of the non-digits are letters of the alphabet from A to F (lines 1470 to 1490), which are legal hex digits. Once all of the checking is done, the program goes about converting the legal alpha characters A to F to the numerical range of $BA to $BF. This is done by subtracting 6 from the current alpha value (line 1620).

At this point, all of the hexadecimal digits that have been entered have the proper hex digit in the right-most (least significant) nibble and a $B in the left-most (most significant) nibble. If we can get rid of the $B and combine the four least significant nibbles in the proper order, we can produce the hex number we require. This is



what is done in lines 1630 to 1720. From 1630 to 1660, the low-order nibble is shifted left four times so that it becomes the high order nibble. The $B that was there previously is thus eliminated.

Now that we have the first digit of our hexadecimal number as the high-order nibble of the accumulator, all we have to do is shift it into LINNUM and from there into LINNUM + 1. This is done by the code in lines 1670 to 1720. Now, if this whole process is repeated for each digit of the hex number, starting with the most significant digit (as we have here), the answer will appear in locations LINNUM and LINNUM + 1. As each digit is added, it gets shifted from the low-order byte of LINNUM to the high-order byte of LINNUM and then to the low-order byte of LINNUM + 1 and finally to the high-order byte of LINNUM + 1.

Throughout the last two chapters we have looked at a variety of ways of getting information into and out of the computer. We've even learned how to draw borders on the screen. Now, let's put a few of the things we've learned together to produce a program subroutine that all assembly language programmers have had to write at one time or another. We'll write a selection menu program that will print out a title and several selection choices, allow the user to pick a choice and then jump to the appropriate routine. The task of allowing the user to select one option from a list of

```
                1000 ************************************
                1010 ***                              ***
                1020 ***  INPUT A HEX NUMBER ROUTINE   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 * EQUATES
                1120 *
0006-           1130 LENGTH   .EQ $6
0050-           1140 LINNUM   .EQ $50
0200-           1150 IN       .EQ $200
FD6F-           1160 GETLN1   .EQ $FD6F
FD8E-           1170 CROUT    .EQ $FD8E
FF2D-           1180 PRERR    .EQ $FF2D
                1190 *
                1200 *
                1210 *
                1220 *
                1230 * This section of code handles entry
                1240 * of the number from the keyboard and
                1250 * then checks each digit to see that it
                1260 * is valid. It also checks to see if
                1270 * more than 4 digits have been entered.
                1280 * If an error is detected an error
                1290 * message is generated.
                1300 *
                1310 *
0800- 20 6F FD  1320 START    JSR GETLN1     Get a number
0803- E0 00     1330          CPX #$0        Any entry?
0805- F0 37     1340          BEQ ERROR      No, do over.
0807- E0 05     1350          CPX #$5        Is it >4 digits?
0809- B0 33     1360          BCS ERROR      Yes, do over.
080B- 86 06     1370          STX LENGTH     Save number of digits.
080D- A9 00     1380          LDA #$0        Initialize
080F- 85 50     1390          STA LINNUM     hex number to
0811- 85 51     1400          STA LINNUM+1   zero.
```



```
0813- A0 00     1410          LDY #$0
0815- B9 00 02  1420 LOOP     LDA IN,Y       Get a character.
0818- C9 B0     1430          CMP #$B0       Test to see if
081A- 90 22     1440          BCC ERROR      it is a digit
081C- C9 BA     1450          CMP #$BA       from 0 to 9
081E- 90 0A     1460          BCC OKAY       or A through F
0820- C9 C1     1470          CMP #$C1
0822- 90 1A     1480          BCC ERROR
0824- C9 C7     1490          CMP #$C7
0826- B0 16     1500          BCS ERROR
                1510 *
                1520 *
                1530 * This section of code converts the
                1540 * letters A through F to the
                1550 * hexadecimal values $BA through $BF
                1560 * by subtracting 6 from the value of
                1570 * the letter.  The low order nibble of
                1580 * the accumulator is moved into the
                1590 * high order nibble and the accumulator
                1600 * is shifted into LINNUM and LINNUM+1.
                1610 *
0828- E9 06     1620          SBC #$6        Convert A-F
082A- 0A        1630 OKAY     ASL            Shift lo order
082B- 0A        1640          ASL            nibble to hi
082C- 0A        1650          ASL            order nibble.
082D- 0A        1660          ASL
082E- A2 04     1670          LDX #$4
0830- 0A        1680 SHIFT    ASL            Shift
0831- 26 50     1690          ROL LINNUM     accumulator
0833- 26 51     1700          ROL LINNUM+1   into LINNUM
0835- CA        1710          DEX            and LINNUM+1
0836- D0 F8     1720          BNE SHIFT
                1730 *
                1740 *
                1750 * This section checks to see if all of
                1760 * the digits have been processed and if
                1770 * not gets another digit until there
                1780 * are no more.
                1790 *
0838- C8        1800 CHKDONE  INY
0839- C4 06     1810          CPY LENGTH     Finished?
083B- D0 D8     1820          BNE LOOP       No, get more.
083D- 60        1830          RTS            Yes, no more.
                1840 *
                1850 *
                1860 * This subroutine rings the bell and
                1870 * prints out the message ERR followed
                1880 * by a carriage return.  Control is
                1890 * then passed back to the beginning of
                1900 * the program so that a valid number
                1910 * can be entered.
                1920 *
083E- 20 2D FF  1930 ERROR    JSR PRERR      Error message.
0841- 20 8E FD  1940          JSR CROUT      Output a carriage return.
0844- 4C 00 08  1950          JMP START      Start over.
```

many and then jumping to the appropriate routine is not difficult. But frequently it is done in an inefficient manner. Here is a general purpose routine that I'm sure you'll find very useful.

Use a library to make programming easier

The SAMPLE MENU PROGRAM uses all or part of three programs that we have already discussed: LONG MESSAGE PRINTER NO. 2, TITLE BOX and the IMPROVED READ KEYBOARD ROUTINE. In fact, if you examine the program carefully, you'll see that only about 30% of it is new, the remainder is just

routines that we have already discussed. This situation illustrates a very important concept, that the best way to program is to use routines from a library of programs that you have already developed.

It also illustrates another important concept, that the programming task should be broken down into individual modules and programmed one module at a time. This makes the programming task more manageable and also makes troubleshooting a program a lot easier. Remember, programs don't always work the first time out.

How to write a menu program

Getting back to our menu program, after clearing the screen, our program jumps immediately to the message printing subroutine. This routine is different than the ones we've used recently. It is the in-line message printing routine that we examined in the last chapter. One advantage of this subroutine, is that it is a little easier to follow the flow of the program because the messages that are printed out are integrated into the program at exactly the spot they are needed. There are two distinct disadvantages to this approach however. One is that if you are trying to trace the operation of a program with an in-line printing routine and you don't have an original source code listing, it is very difficult to do. The second is that if you ever decide to do foreign language translations of your program, it is a lot easier to do if all of the text is grouped in one specific place.

The program prints out the title, the menu of choices and the prompting message asking for the user's choice. The next thing it does, is it stores the current position of the cursor (lines 1610 to 1640), which is one space after the colon on the line 'ENTER CHOICE: ' and then goes back and draws the box around the title (line 1650). After drawing the box, the program them restores the cursor to its former position right after the choice prompt. It then reads the keyboard (line 2230) looking for any number in the range of 1 to 7. If anything other than a number within this range is pressed, the entry is ignored.

If a number in the 1 to 7 range is selected, the number is converted from ASCII with the high bit set, to hexadecimal and 1 is subtracted from the number to put it in the 0 to 6 range (line 2380). Next, this number is going to be converted into an index into a table of addresses that will be used to retrieve the address of the subroutine that is desired. Since the addresses in the table all require two bytes, the number that we got from line 2380 must be doubled (line 2390). Thus, if we selected item 1, line 2380 resulted in the number 0, this is doubled and we still have zero, so the address information we want starts at the beginning of the table with no offset. If we had selected item number 4, that number would be converted to 3 which would be doubled to 6. This means that the address we want starts at the 7th byte from the beginning of the table. If each address takes of up two bytes and there are 3 choices before this one, six bytes have already been used. So it's easy to see why the information we want is at the 7th byte. The byte we retrieved with the offset of six (line 2410), was the seventh byte because we started counting from zero.

Using the stack to jump to a subroutine

Once we have the correct index into the jump table, the address of the routine we want to jump to is loaded into the accumulator and then pushed onto the stack (lines 2410 to 2450), HIGH BYTE FIRST! I emphasize this, because all other two-byte operations with the 6502 deal with the low byte first. We push the address on the stack in this manner, because the stack has a LIFO (Last In, First Out) structure. That means that when the address is pulled off the stack to jump to the appropriate subroutine, it will be pulled off in the conventional manner, low byte first.

If you take a careful look at the table of jump addresses that begins at line 2860, you'll notice two things. The first is that the addresses have been stored in the table high-byte first. This makes it easier for the programmer to read when he's looking at the source listing and also makes it easier to push the address on the stack in the proper order. More important than that, if you look at the addresses in the table and the actual address of the start of the various routines, you'll find that the address in the table is always one less than the real address. The reason for this is simple. By pushing the address on the stack, we've fooled the 6502 processor into thinking that the program executed a JSR instruction. So, when an RTS instruction is executed (line 2460), the 6502 pulls the first two bytes off the top of the stack, low-byte first, it increments the low-byte by one, and then jumps to the address, thinking it is returning from a subroutine call to execute the next available instruction. This method of implementing an absolute jump to another part of the program, based on addresses retrieved from a table is a fairly efficient way of doing things.

That's the meat of the program. The only thing left to go over is the code from

lines 2570 to 2770. This code simple implements a demonstration program that will tell us that the menu selection routine works. What it does is ring the bell the same number of times as the menu selection number. So, if item number four on the menu is chosen, the bell rings four times and if item number five is chosen it rings five times, etc. If item number seven is chosen, the program does an RTS and goes back to the calling routine or mode.

```
                        1000 ************************************
                        1010 ***                              ***
                        1020 ***      SAMPLE MENU PROGRAM     ***
                        1030 ***                              ***
                        1040 ************************************
                        1050 *
                        1060 *
                        1070 *
                        1080 *
                        1090 * CONSTANTS
                        1100 *
0003-                   1110 BOXLEN .EQ $03
0003-                   1120 LFTMRG .EQ $03
0025-                   1130 RTMRG  .EQ $25
00AA-                   1140 SYMBOL .EQ $AA
                        1150 *
                        1160 *
                        1170 * EQUATES
                        1180 *
0006-                   1190 TXTPTR .EQ $06
0018-                   1200 CV2    .EQ $18
0019-                   1210 CH2    .EQ $19
0024-                   1220 CH     .EQ $24
0025-                   1230 CV     .EQ $25
FC22-                   1240 BASCAL .EQ $FC22
FC58-                   1250 HOME   .EQ $FC58
FC9C-                   1260 CLREOL .EQ $FC9C
FD0C-                   1270 RDKEY  .EQ $FD0C
FDED-                   1280 COUT   .EQ $FDED
FF3A-                   1290 BELL   .EQ $FF3A
FF58-                   1300 RETURN .EQ $FF58
                        1310 *
                        1320 *
                        1330 * Clear the screen, print out the title
                        1340 * of the program and the selection menu
                        1350 * and save the position location of the
                        1360 * cursor for later.
                        1370 *
0800- 20 58 FC          1380 START  JSR HOME        Clear the screen.
0803- 20 41 09          1390        JSR MSGPRT      Print the message that follows.
0806- 8D 8D             1400        .HS 8D8D
0808- A0 A0 A0
080B- A0 A0 A0
080E- A0 A0 A0
0811- A0 D3 C1
0814- CD D0 CC
0817- C5 A0 CD
081A- C5 CE D5
081D- A0 D0 D2
0820- CF C7 D2
0823- C1 CD             1410        .AS -"          SAMPLE MENU PROGRAM"
0825- 8D 8D 8D
0828- 8D 8D             1420        .HS 8D8D8D8D8D
082A- C5 CE D4
082D- C5 D2 A0
0830- D4 C8 C5
0833- A0 CE D5
0836- CD C2 C5
0839- D2 A0 CF
083C- C6 A0 C2
083F- C5 CC CC
```



```
0842- D3 A0 D9
0845- CF D5 A0
0848- D7 C1 CE
084B- D4 A0 D4
084E- CF A0 A0
0851- A0 D2 C9
0854- CE C7 BA          1430        .AS -"ENTER THE NUMBER OF BELLS YOU WANT TO RING:"
0857- 8D 8D             1440        .HS 8D8D
0859- BC B1 BE
085C- A0 CF CE
085F- C5                1450        .AS -"<1> ONE"
0860- 8D                1460        .HS 8D
0861- BC B2 BE
0864- A0 D4 D7
0867- CF                1470        .AS -"<2> TWO"
0868- 8D                1480        .HS 8D
0869- BC B3 BE
086C- A0 D4 C8
086F- D2 C5 C5          1490        .AS -"<3> THREE"
0872- 8D                1500        .HS 8D
0873- BC B4 BE
0876- A0 C6 CF
0879- D5 D2             1510        .AS -"<4> FOUR"
087B- 8D                1520        .HS 8D
087C- BC B5 BE
087F- A0 C6 C9
0882- D6 C5             1530        .AS -"<5> FIVE"
0884- 8D                1540        .HS 8D
0885- BC B6 BE
0888- A0 D3 C9
088B- D8                1550        .AS -"<6> SIX"
088C- 8D                1560        .HS 8D
088D- BC B7 BE
0890- A0 D1 D5
0893- C9 D4             1570        .AS -"<7> QUIT"
0895- 8D 8D             1580        .HS 8D8D
0897- C5 CE D4
089A- C5 D2 A0
089D- C3 C8 CF
08A0- C9 C3 C5
08A3- BA A0             1590        .AS -"ENTER CHOICE: "
08A5- 00                1600        .HS 00
08A6- A5 25             1610        LDA CV          Save the current vertical
08A8- 85 18             1620        STA CV2         position of the cursor.
08AA- A5 24             1630        LDA CH          Save the current horizontal
08AC- 85 19             1640        STA CH2         position of the cursor.
08AE- 20 B4 08          1650        JSR BOX         Draw a title box.
08B1- 4C DF 08          1660        JMP CURPOS      Restore original cursor position.
                        1670 *
                        1680 *
                        1690 * Print a box around the title of the
                        1700 * program.
                        1710 *
08B4- A9 00             1720 BOX    LDA #$0         Start at row zero, column zero.
08B6- 85 25             1730        STA CV
08B8- 85 24             1740        STA CH
08BA- 20 22 FC          1750        JSR BASCAL      Position cursor.
08BD- 20 D5 08          1760        JSR LINE        Draw top line of box.
08C0- A2 03             1770        LDX #BOXLEN     Get depth of box.
08C2- 20 ED FD          1780 LOOP   JSR COUT        Draw middle lines
08C5- A4 24             1790        LDY CH          of box.
08C7- C0 03             1800        CPY #LFTMRG     At end of left margin yet?
08C9- D0 F7             1810        BNE LOOP        No, print more symbols.
08CB- A0 25             1820        LDY #RTMRG      Jump to start of
08CD- 84 24             1830        STY CH          right margin.
08CF- CA                1840        DEX             End of box?
08D0- D0 F0             1850        BNE LOOP        No, finish right margin.
08D2- 20 D5 08          1860        JSR LINE        Yes, finish right margin.
                        1870 *
                        1880 *
                        1890 * This subroutine prints out a line of
                        1900 * symbols.  It checks CH to see if
                        1910 * it has past the 40th column and
                        1920 * wrapped around to column 0.
```

```
                    1930 *
08D5- A9 AA         1940 LINE   LDA #SYMBOL     Get symbol.
08D7- 20 ED FD      1950        JSR COUT        Print it.
08DA- A4 24         1960        LDY CH          Done yet?
08DC- D0 F7         1970        BNE LINE        No, do more.
08DE- 60            1980        RTS
                    1990 *
                    2000 *
                    2010 * This subroutine restores the cursor
                    2020 * to its original position before the
                    2030 * box was drawn so that it is ready to
                    2040 * prompt the user.
                    2050 *
08DF- A5 19         2060 CURPOS LDA CH2         Get old horizontal position.
08E1- 85 24         2070        STA CH          Make it current position.
08E3- A5 18         2080        LDA CV2         Get old vertical position.
08E5- 85 25         2090        STA CV          Make it current position.
08E7- 20 22 FC      2100        JSR BASCAL      Position cursor.
08EA- 20 9C FC      2110        JSR CLREOL      Clear to the end of line.
                    2120 *
                    2130 *
                    2140 * This subroutine checks the keyboard
                    2150 * to see if a key is pressed.  When it
                    2160 * is, the key that is pressed is
                    2170 * printed out.  Then it's value is
                    2180 * checked to see if it is less than 8
                    2190 * and equal to or greater than 1.  If
                    2200 * is not, the program starts all over
                    2210 * again.
                    2220 *
08ED- 20 0C FD      2230        JSR RDKEY       Wait for a key press.
08F0- 20 ED FD      2240        JSR COUT        Print it.
08F3- C9 B8         2250        CMP #$B8        Was it greater than 7?
08F5- B0 13         2260        BCS RSTART      Yes, restart.
08F7- C9 B1         2270        CMP #$B1        No, was it less than 1?
08F9- 90 0F         2280        BCC RSTART      Yes, restart.
                    2290 *
                    2300 *
                    2310 * If the key pressed is in the correct
                    2320 * range, value is normalized to numbers
                    2330 * in the 0 to 6 range and then doubled.
                    2340 * The number thus generated is used as
                    2350 * an index into the table of jump
                    2360 * addresses.
                    2370 *
08FB- E9 B1         2380        SBC #$B1        Normalize entered digit.
08FD- 0A            2390        ASL             Multiply it by 2.
08FE- A8            2400        TAY             Put into Y-register as index.
08FF- C8            2405        INY
0900- B9 33 09      2410        LDA TABLE,Y     Retrieve address from table
0903- 48            2420        PHA             and save it on the stack.
0904- 88            2430        DEY
0905- B9 33 09      2440        LDA TABLE,Y
0908- 48            2450        PHA
0909- 60            2460        RTS             Jump to the subroutine chosen.
090A- 4C DF 08      2470 RSTART JMP CURPOS      Start over.
                    2480 *
                    2490 *
                    2500 * This is just a little routine that
                    2510 * been included to demonstrate that the
                    2520 * menu selection is working.  It is
                    2530 * entered with the Y-register
                    2540 * containing the number of bell rings
                    2550 * desired.
                    2560 *
090D- 20 3A FF      2570 RNGBEL JSR BELL
0910- 88            2580        DEY
0911- D0 FA         2590        BNE RNGBEL
0913- F0 F5         2600        BEQ RSTART
                    2610 *
                    2620 *
                    2630 * The Y-register for the bell ringing
                    2640 * routine is set here.
```



```
                    2650 *
0915- A0 01         2660 ONE    LDY #$1
0917- 4C 0D 09      2670        JMP RNGBEL
091A- A0 02         2680 TWO    LDY #$2
091C- 4C 0D 09      2690        JMP RNGBEL
091F- A0 03         2700 THREE  LDY #$3
0921- 4C 0D 09      2710        JMP RNGBEL
0924- A0 04         2720 FOUR   LDY #$4
0926- 4C 0D 09      2730        JMP RNGBEL
0929- A0 05         2740 FIVE   LDY #$5
092B- 4C 0D 09      2750        JMP RNGBEL
092E- A0 06         2760 SIX    LDY #$6
0930- 4C 0D 09      2770        JMP RNGBEL
                    2780 *
                    2790 *
                    2800 * This is the table of jump addresses.
                    2810 * The address minus 1 of the subroutine
                    2820 * that is to be jumped to is entered
                    2830 * in this table with the high-order
                    2840 * byte first.
                    2850 *
0933- 14 09         2860 TABLE  .DA ONE-1       Address of selection 1.
0935- 19 09         2870        .DA TWO-1       Address of selection 2.
0937- 1E 09         2880        .DA THREE-1     Address of selection 3.
0939- 23 09         2890        .DA FOUR-1      Address of selection 4.
093B- 28 09         2900        .DA FIVE-1      Address of selection 5.
093D- 2D 09         2910        .DA SIX-1       Address of selection 6.
093F- 57 FF         2920        .DA RETURN-1    Address of selection 7.
                    2930 *
                    2940 *
                    2950 * This is the message printing
                    2960 * subroutine.
                    2970 *
0941- 68            2980 MSGPRT PLA             Store address of text to
0942- 85 06         2990        STA TXTPTR      be printed in a zero
0944- 68            3000        PLA             page pointer.
0945- 85 07         3010        STA TXTPTR+1
0947- A0 00         3020        LDY #$0
0949- E6 06         3030 NEXT   INC TXTPTR      Increment 2-byte pointer
094B- D0 02         3040        BNE CONTIN      to text.
094D- E6 07         3050        INC TXTPTR+1
094F- B1 06         3060 CONTIN LDA (TXTPTR),Y  Get character.
0951- F0 06         3070        BEQ ENDPRT      Done yet?
0953- 20 ED FD      3080        JSR COUT        No, print it.
0956- 4C 49 09      3090        JMP NEXT        Get next character.
0959- A5 07         3100 ENDPRT LDA TXTPTR+1    Push the address of
095B- 48            3110        PHA             where to resume the program
095C- A5 06         3120        LDA TXTPTR      onto the stack
095E- 48            3130        PHA
095F- 60            3140        RTS             and jump there.
```

Using an alphabetic menu

You should find that this menu program will fill most of your needs. But what happens if you have more than nine items to choose from? This program is designed to work with a single key press, what can you do? The answer is simple. Don't use numbers, use letters. If you use letters you will have the ability to have up to 26 choices. And if you have more than that, you should consider using multiple screens.

The ALPHABETIC MENU PROGRAM is almost identical to the numerical menu program, and requires only a few minor changes to the original SAMPLE MENU PROGRAM to produce. In this version, the number of items to choose from was increased to 11 and the selections have been changed from 1-7 to A-K. Aside from the obvious changes in the text that gets displayed on the screen, changes were made in lines 2330 and 2350. In these lines we check to see if the

keypress was greater than 'K' or less than 'A' instead of greater than 7 and less than 1. Also, in line 2460, the data that was entered with the keypress is normalized to the 0 to 10 range by subtracting the ASCII value plus $80 (altogether $C1) of the letter 'A'. The only other changes to the program were to put in the new addresses for the routines in the jump table and to add the extra routines that were required.

While the ALPHABETIC MENU and the previous program SAMPLE MENU are almost the same, I have included a complete listing of the ALPHABETIC MENU so that you can compare the two listings and see exactly how the changes were made.

```
                        1000 ************************************
                        1010 ***                              ***
                        1020 ***    ALPHABETIC MENU PROGRAM   ***
                        1030 ***                              ***
                        1040 ************************************
                        1050 *
                        1060 *
                        1070 *
                        1080 *
                        1090 * CONSTANTS
                        1100 *
0003-                   1110 BOXLEN .EQ $03
0003-                   1120 LFTMRG .EQ $03
0025-                   1130 RTMRG  .EQ $25
00AA-                   1140 SYMBOL .EQ $AA
                        1150 *
                        1160 *
                        1170 * EQUATES
                        1180 *
0006-                   1190 TXTPTR .EQ $06
0018-                   1200 CV2    .EQ $18
0019-                   1210 CH2    .EQ $19
0024-                   1220 CH     .EQ $24
0025-                   1230 CV     .EQ $25
FC22-                   1240 BASCAL .EQ $FC22
FC58-                   1250 HOME   .EQ $FC58
FC9C-                   1260 CLREOL .EQ $FC9C
FD0C-                   1270 RDKEY  .EQ $FD0C
FDED-                   1280 COUT   .EQ $FDED
FF3A-                   1290 BELL   .EQ $FF3A
FF58-                   1300 RETURN .EQ $FF58
                        1310 *
                        1320 *
                        1330 * Clear the screen, print out the title
                        1340 * of the program and the selection menu
                        1350 * and save the position location of the
                        1360 * cursor for later.
                        1370 *
0800- 20 58 FC          1380 START  JSR HOME        Clear the screen.
0803- 20 84 09          1390        JSR MSGPRT      Print the message that follows.
0806- 8D 8D             1400        .HS 8D8D
0808- A0 A0 A0
080B- A0 A0 A0
080E- A0 A0 C1
0811- CC D0 C8
0814- C1 C2 C5
0817- D4 C9 C3
081A- A0 CD C5
081D- CE D5 A0
0820- D0 D2 CF
0823- C7 D2 C1
0826- CD                1410        .AS -"      ALPHABETIC MENU PROGRAM"
0827- 8D 8D 8D
082A- 8D 8D             1420        .HS 8D8D8D8D8D
082C- C5 CE D4
082F- C5 D2 A0
```



```
0832- D4 C8 C5
0835- A0 CE D5
0838- CD C2 C5
083B- D2 A0 CF
083E- C6 A0 C2
0841- C5 CC CC
0844- D3 A0 D9
0847- CF D5 A0
084A- D7 C1 CE
084D- D4 A0 D4
0850- CF A0 A0
0853- A0 D2 C9
0856- CE C7 BA          1430        .AS -"ENTER THE NUMBER OF BELLS YOU WANT TO RING:"
0859- 8D 8D             1440        .HS 8D8D
085B- BC C1 BE
085E- A0 CF CE
0861- C5                1450        .AS -"<A> ONE"
0862- 8D                1460        .HS 8D
0863- BC C2 BE
0866- A0 D4 D7
0869- CF                1470        .AS -"<B> TWO"
086A- 8D                1480        .HS 8D
086B- BC C3 BE
086E- A0 D4 C8
0871- D2 C5 C5          1490        .AS -"<C> THREE"
0874- 8D                1500        .HS 8D
0875- BC C4 BE
0878- A0 C6 CF
087B- D5 D2             1510        .AS -"<D> FOUR"
087D- 8D                1520        .HS 8D
087E- BC C5 BE
0881- A0 C6 C9
0884- D6 C5             1530        .AS -"<E> FIVE"
0886- 8D                1540        .HS 8D
0887- BC C6 BE
088A- A0 D3 C9
088D- D8                1550        .AS -"<F> SIX"
088E- 8D                1560        .HS 8D
088F- BC C7 BE
0892- A0 D3 C5
0895- D6 C5 CE          1570        .AS -"<G> SEVEN"
0898- 8D                1580        .HS 8D
0899- BC C8 BE
089C- A0 C5 C9
089F- C7 C8 D4          1590        .AS -"<H> EIGHT"
08A2- 8D                1600        .HS 8D
08A3- BC C9 BE
08A6- A0 CE C9
08A9- CE C5             1610        .AS -"<I> NINE"
08AB- 8D                1620        .HS 8D
08AC- BC CA BE
08AF- A0 D4 C5
08B2- CE                1630        .AS -"<J> TEN"
08B3- 8D                1640        .HS 8D
08B4- BC CB BE
08B7- A0 D1 D5
08BA- C9 D4             1650        .AS -"<K> QUIT"
08BC- 8D 8D             1660        .HS 8D8D
08BE- C5 CE D4
08C1- C5 D2 A0
08C4- C3 C8 CF
08C7- C9 C3 C5
08CA- BA A0             1670        .AS -"ENTER CHOICE: "
08CC- 00                1680        .HS 00
08CD- A5 25             1690        LDA CV          Save the current vertical
08CF- 85 18             1700        STA CV2         position of the cursor.
08D1- A5 24             1710        LDA CH          Save the current horizontal
08D3- 85 19             1720        STA CH2         position of the cursor.
08D5- 20 DB 08          1730        JSR BOX         Draw a title box.
08D8- 4C 06 09          1740        JMP CURPOS      Restore original cursor position.
                        1750 *
                        1760 *
                        1770 * Print a box around the title of the
```

```
                1780 * program.
                1790 *
08DB- A9 00     1800 BOX     LDA #$0          Start at row zero, column zero.
08DD- 85 25     1810         STA CV
08DF- 85 24     1820         STA CH
08E1- 20 22 FC  1830         JSR BASCAL       Position cursor.
08E4- 20 FC 08  1840         JSR LINE         Draw top line of box.
08E7- A2 03     1850         LDX #BOXLEN      Get depth of box.
08E9- 20 ED FD  1860 LOOP    JSR COUT         Draw middle lines
08EC- A4 24     1870         LDY CH           of box.
08EE- C0 03     1880         CPY #LFTMRG      At end of left margin yet?
08F0- D0 F7     1890         BNE LOOP         No, print more symbols.
08F2- A0 25     1900         LDY #RTMRG       Jump to start of
08F4- 84 24     1910         STY CH           right margin.
08F6- CA        1920         DEX              End of box?
08F7- D0 F0     1930         BNE LOOP         No, finish right margin.
08F9- 20 FC 08  1940         JSR LINE         Yes, finish right margin.
                1950 *
                1960 *
                1970 * This subroutine prints out a line of
                1980 * symbols.  It checks CH to see if
                1990 * it has past the 40th column and
                2000 * wrapped around to column 0.
                2010 *
08FC- A9 AA     2020 LINE    LDA #SYMBOL      Get symbol.
08FE- 20 ED FD  2030         JSR COUT         Print it.
0901- A4 24     2040         LDY CH           Done yet?
0903- D0 F7     2050         BNE LINE         No, do more.
0905- 60        2060         RTS
                2070 *
                2080 *
                2090 * This subroutine restores the cursor
                2100 * to its original position before the
                2110 * box was drawn so that it is ready to
                2120 * prompt the user.
                2130 *
0906- A5 19     2140 CURPOS  LDA CH2          Get old horizontal position.
0908- 85 24     2150         STA CH           Make it current position.
090A- A5 18     2160         LDA CV2          Get old vertical position.
090C- 85 25     2170         STA CV           Make it current position.
090E- 20 22 FC  2180         JSR BASCAL       Position cursor.
0911- 20 9C FC  2190         JSR CLREOL       Clear to the end of line.
                2200 *
                2210 *
                2220 * This subroutine checks the keyboard
                2230 * to see if a key is pressed.  When it
                2240 * is, the key that is pressed is
                2250 * printed out.  Then it's value is
                2260 * checked to see if it is less than 'L'
                2270 * and equal to or greater than 'A'.  If
                2280 * is not, the program starts all over
                2290 * again.
                2300 *
0914- 20 0C FD  2310         JSR RDKEY        Wait for a key press.
0917- 20 ED FD  2320         JSR COUT         Print it.
091A- C9 CC     2330         CMP #$CC         Was it greater than 'K'?
091C- B0 13     2340         BCS RSTART       Yes, restart.
091E- C9 C1     2350         CMP #$C1         No, was it less than 'A'?
0920- 90 0F     2360         BCC RSTART       Yes, restart.
                2370 *
                2380 *
                2390 * If the key pressed is in the correct
                2400 * range, value is normalized to numbers
                2410 * in the 0 to 10 range and then doubled.
                2420 * The number thus generated is used as
                2430 * an index into the table of jump
                2440 * addresses.
                2450 *
0922- E9 C1     2460         SBC #$C1         Normalize entered digit.
0924- 0A        2470         ASL              Multiply it by 2.
0925- A8        2480         TAY              Put into Y register as index.
0926  C8        2485         INY
0927  B9 6E 09  2490         LDA TABLE,Y      Retrieve address from table
```



```
092A- 48        2500         PHA              and save it on the stack.
092B- 88        2510         DEY
092C- B9 6E 09  2520         LDA TABLE,Y
092F- 48        2530         PHA
0930- 60        2540         RTS              Jump to the subroutine chosen.
0931- 4C 06 09  2550 RSTART  JMP CURPOS       Start over.
                2560 *
                2570 *
                2580 * This is just a little routine that
                2590 * been included to demonstrate that the
                2600 * menu selection is working.  It is
                2610 * entered with the Y-register
                2620 * containing the number of bell rings
                2630 * desired.
                2640 *
0934- 20 3A FF  2650 RNGBEL  JSR BELL
0937- 88        2660         DEY
0938- D0 FA     2670         BNE RNGBEL
093A- F0 F5     2680         BEQ RSTART
                2690 *
                2700 *
                2710 * The Y-register for the bell ringing
                2720 * routine is set here.
                2730 *
093C- A0 01     2740 ONE     LDY #$1
093E- 4C 34 09  2750         JMP RNGBEL
0941- A0 02     2760 TWO     LDY #$2
0943- 4C 34 09  2770         JMP RNGBEL
0946- A0 03     2780 THREE   LDY #$3
0948- 4C 34 09  2790         JMP RNGBEL
094B- A0 04     2800 FOUR    LDY #$4
094D- 4C 34 09  2810         JMP RNGBEL
0950- A0 05     2820 FIVE    LDY #$5
0952- 4C 34 09  2830         JMP RNGBEL
0955- A0 06     2840 SIX     LDY #$6
0957- 4C 34 09  2850         JMP RNGBEL
095A- A0 07     2860 SEVEN   LDY #$7
095C- 4C 34 09  2870         JMP RNGBEL
095F- A0 08     2880 EIGHT   LDY #$8
0961- 4C 34 09  2890         JMP RNGBEL
0964- A0 09     2900 NINE    LDY #$9
0966- 4C 34 09  2910         JMP RNGBEL
0969- A0 0A     2920 TEN     LDY #$A
096B- 4C 34 09  2930         JMP RNGBEL
                2940 *
                2950 *
                2960 * This is the table of jump addresses.
                2970 * The address minus 1 of the subroutine
                2980 * that is to be jumped to is entered
                2990 * in this table with the high-order
                3000 * byte first.
                3010 *
096E- 3B 09     3020 TABLE   .DA ONE-1        Address of selection A.
0970- 40 09     3030         .DA TWO-1        Address of selection B.
0972- 45 09     3040         .DA THREE-1      Address of selection C.
0974- 4A 09     3050         .DA FOUR-1       Address of selection D.
0976- 4F 09     3060         .DA FIVE-1       Address of selection E.
0978- 54 09     3070         .DA SIX-1        Address of selection F.
097A- 59 09     3080         .DA SEVEN-1      Address of selection G.
097C- 5E 09     3090         .DA EIGHT-1      Address of selection H.
097E- 63 09     3100         .DA NINE-1       Address of selection I.
0980- 68 09     3110         .DA TEN-1        Address of selection J.
0982- 57 FF     3120         .DA RETURN-1     Address of selection K.
                3130 *
                3140 *
                3150 * This is the message printing
                3160 * subroutine.
                3170 *
0984  68        3180 MSGPRT  PLA              Store address of text to
0985- 85 06     3190         STA TXTPTR       be printed in a zero
0987  68        3200         PLA              page pointer.
0988  85 07     3210         STA TXTPTR+1
098A  A0 00     3220         LDY #$0
```

```
098C- E6 06     3230 NEXT   INC TXTPTR       Increment 2-byte pointer
098E- D0 02     3240        BNE CONTIN       to text.
0990- E6 07     3250        INC TXTPTR+1
0992- B1 06     3260 CONTIN LDA (TXTPTR),Y   Get character.
0994- F0 06     3270        BEQ ENDPRT       Done yet?
0996- 20 ED FD  3280        JSR COUT         No, print it.
0999- 4C 8C 09  3290        JMP NEXT         Get next character.
099C- A5 07     3300 ENDPRT LDA TXTPTR+1     Push the address of
099E- 48        3310        PHA              where to resume the program
099F- A5 06     3320        LDA TXTPTR       onto the stack
09A1- 48        3330        PHA
09A2- 60        3340        RTS              and jump there.
```

Chapter 4

STEALING CONTROL OF THE OUTPUT

Did you ever wish that there were some way that you could see the control characters that some programmers hide in the names of programs saved out to a disk? Have you ever had a need to customize the interface between your printer and your computer? Perhaps your printer doesn't recognize a blank line and must be sent a space character before a carriage return. Or maybe you want to print something out on your printer in expanded mode but are annoyed by the fact that the expanded mode is cancelled automatically when the printer receives a carriage return and must be reinitialized. Are you tired of continually typing "CTRL-I 80N" or some other setup string required by your printer?

If you are looking for ways to overcome these and other annoying situations that deal with transferring data from your computer to some external device, fret not.

After you read this chapter, you should be able to find a way to let your Apple do all the work for you. The secret to making the Apple do all of the work is two memory locations on page zero known as CSWL and CSWL + 1 ($36 and $37). It is through the proper use of these two locations, that you will learn how to steal control away from the Apple's normal output routines, and direct it to your own machine language program.

If you were to get into the Apple's monitor mode, by typing CALL -151 from either BASIC, and then entered the monitor command 'FDEDL', the first two lines you would see printed on the screen would look like this:

```
FDED-    6C 36 00        JMP     ($0036)
FDF0-    C9 A0           CMP     #SA0
```

The subroutine located at $FDED is called COUT and is the routine that the Apple jumps to every time it wants to print out a character. You'll notice that the first thing this routine does is jump indirectly, through page zero locations $36 and $37, to the real output routine. In an Apple without DOS, if you were to examine locations $36 and $37, you'd find that the Apple jumps right back to $FDF0, which is the very next instruction after the indirect jump. $FDF0 is called COUT1 and is the routine that prints everything out to the screen.

At first glance, this method of programming might seem a bit odd, but if you think about it for a minute, you'll see that by adding the indirect jump instruction as the first line of the routine, characters that are to be printed out can first be diverted to any other desired subroutine. And this in fact, is what happens when DOS is active or when a printer is connected to the system. When DOS is active, characters to be output are first passed to a routine that starts at $9EBD and when a printer is active, characters are passed to ROM routines located on the interface card. Now, if we place the address of our own output routine in locations $36 and $37, and make sure it stays there despite DOS, then we'll be able to do all of the things I mentioned earlier, and more.

Fixing a problem with some parallel printers

The first program we're going to talk about in this chapter is what is usually classified as a patch, because it fixes a specific problem that really shouldn't have occurred to begin with. For those of you that use Centronics or Centronics-compatible parallel printers you may encounter a problem where the printer will not respond to a blank print statement such as this:

```
10 PRINT
```

Normally, statements such as this are put in a program to produce a formatted output that is pleasing to the eye. Centronics printers, and some others as well, will

not respond to a lone carriage return, which is what is generated by line 10, and require that at least one character be printed on a line before the carriage return is recognized. The PARALLEL PRINTER PATCH program intercepts all characters that are being printed out and wherever it finds a carriage return, it first prints a space and then the carriage return, insuring that printers experiencing this problem will always respond as you originally intended.

This is done by replacing the address of the output routine, which is stored in locations $36 and $37, with the address of this program. That job is handled by the routine in lines 1230 to 1310.

Line 1230 gets the low byte of the address of the new output routine while line 1240 gets the high byte of that address. Lines 1250 and 1260 store the address of the new output routine in the output hooks so that anytime a character is supposed to be printed out, it is routed to our program first.

With programs like this, that alter output hooks, programmers usually have two versions of the program: one that does an RTS right after line 1260 for systems

```
                1000 * ************************************
                1010 * ***                              ***
                1020 * ***    PARALLEL PRINTER PATCH     ***
                1030 * ***                              ***
                1040 * ************************************
                1050 *
                1060 *
                1070 *
                1080          .OR $300
                1090 *
                1100 *
                1110 * EQUATES
                1120 *
0036-           1130 CSWL     .EQ $36
03D0-           1140 WARMDOS  .EQ $3D0
03EA-           1150 CONNECT  .EQ $3EA
C200-           1160 SLOT     .EQ $C200
                1170 *
                1180 *
                1190 * Replace the normal output routine
                1200 * with this program by changing the
                1210 * output hooks $36 and $37.
                1220 *
0300- A9 13     1230          LDA #START   Get address of start of
0302- A0 03     1240          LDY /START   program and store it in
0304- 85 36     1250          STA CSWL     the output hooks.
0306- 84 37     1260          STY CSWL+1
0308- AD D0 03  1270          LDA WARMDOS  See if DOS is present.
030B- C9 4C     1280          CMP #$4C
030D- D0 03     1290          BNE NODOS    No it's not, return.
030F- 20 EA 03  1300          JSR CONNECT  It is, connect through DOS.
0312- 60        1310 NODOS    RTS          Return to caller.
                1320 *
                1330 *
                1340 * Check the character that is being
                1350 * output by the computer to see if it
                1360 * is a carriage return.  If it is print
                1370 * a space first and then print a
                1380 * carriage return.
                1390 *
0313- C9 8D     1400 START    CMP #$8D     Is it a carriage return?
0315- D0 07     1410          BNE PRINT    No, print it.
0317- A9 A0     1420          LDA #$A0     Yes, print a
0319- 20 02 C2  1430          JSR SLOT+2   space first and then a
031C- A9 8D     1440          LDA #$8D     carriage return.
031E- 4C 02 C2  1450 PRINT    JMP SLOT+2   Print contents of accumulator.
```

without DOS, and one that does a JSR $3EA for systems with DOS. This latter one is necessary for systems with DOS, because DOS modifies the hooks and sets up its own output routine. The JSR $3EA makes sure there will be no conflicts. In systems without DOS, the information in locations $3EA to $3EC is random and could cause the machine to hang up if jumped to, hence the need for two versions of the program.

However, for the price of a few extra bytes of code, we can test to see if DOS is present in the computer and then connect the output hooks by the appropriate manner. That's what the code in lines 1270 to 1310 does.

The actual program patch starts at line 1400 where the first thing that is done is the character being output is checked to see if it is a carriage return. If the character that is intercepted is a carriage return, first a space is printed out (lines 1420 and 1430) and then a carriage return is printed out (lines 1440 and 1450). If it's not, the program branches to line 1450 and prints the character on the printer, which here is connected to a card in slot 2. If your printer is in a different slot, just change the equate in line 1160. The 2 that is added to SLOT in lines 1430 and 1450, is for Apple parallel interface cards. If an Apple serial interface card is used, and this patch is required, a 7 should be used instead.

Printer interface cards from other manufacturers may require other numbers. To find out, plug your printer card with the printer attached to it and on, into slot 1 and then type PR#1. Then, while the printer is activated, look at what number is stored in location $36. This can be done by typing CALL -151 and then typing the number 36 and a carriage return. The computer will respond with 0036— XY, where XY is the number to be added to SLOT.

As you can see, this routine is very short (only 33 bytes long) and it can easily sit in the unused portion of page three. To use this routine, instead of typing PR#2, all you do is type CALL 768. The routine can still be turned off by typing PR#0.

This program showed you how it is possible to detect a particular character and then send out another character or set of characters in its place. You could easily modify this program to send a line feed every time it detected a carriage return, for those printers that require it, or you could use it to send codes to the printer to change its mode from normal to expanded, or italics, etc. The possibilities are limited only by your imagination.

Getting more out of your Epson printer

The Epson printer is probably one of the most widely used printers available today. The reason for this is its low price and its versatility. The original Epson MX-80 was able to print in condensed, normal and expanded (double width) modes as well as combinations of the above. For example, you could combine condensed and expanded to form a bold, or enhanced, mode.

When trying to use the expanded mode, by itself or in combination with another mode, you'll very quickly discover one of the shortcomings of the Epson printer: the expanded mode is automatically cancelled when a carriage return is encoun-


```
                1000 ************************************
                1010 ***                              ***
                1020 ***      EPSON PRINTER PATCH     ***
                1030 ***                              ***
                1040 ***    COPYRIGHT (C) 1982 BY     ***
                1050 ***       JULES H. GILDER        ***
                1060 ***     ALL RIGHTS RESERVED      ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $0300
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
0006-           1180 NFLAG    .EQ $6
0008-           1190 TXTPTR   .EQ $8
0036-           1200 CSWL     .EQ $36
03D0-           1210 WARMDOS  .EQ $3D0
03EA-           1220 CONNECT  .EQ $3EA
C202-           1230 PRINTER  .EQ $C202
FC58-           1240 HOME     .EQ $FC58
FDED-           1250 COUT     .EQ $FDED
                1260 *
                1270 *
                1280 * Here the program name and copyright
                1290 * notice are printed out.
                1300 *
0300- 20 58 FC  1310          JSR HOME        Clear screen.
0303- A9 53     1320          LDA #TEXT       Point to text
0305- A0 03     1330          LDY /TEXT       to be printed.
0307- 20 42 03  1340          JSR MSGPRT      Print it.
                1350 *
                1360 * This subroutine sets the Control-N flag
                1370 * (NFLAG) to zero and sets up the
                1380 * output hooks to point to the patch
                1390 * program that begins on line 1680.
                1400 *
030A- A9 00     1410          LDA #$0         Set NFLAG to
030C- 85 06     1420          STA NFLAG       zero.
030E- A9 21     1430          LDA #START      Get address of
0310- A0 03     1440          LDY /START      patch and save
0312- 85 36     1450          STA CSWL        it in output
0314- 84 37     1460          STY CSWL+1      hooks.
0316- AD D0 03  1470          LDA WARMDOS     See if DOS is
0319- C9 4C     1480          CMP #$4C        present.
031B- D0 03     1490          BNE NODOS       No, return.
031D- 20 EA 03  1500          JSR CONNECT     Connect to DOS.
0320- 60        1510 NODOS    RTS             Return.
                1520 *
                1530 *
                1540 * This routine checks to see if the
                1550 * character being printed is a
                1560 * Control-T, Control-N or a carriage
                1570 * return. If it is a Control-T it
                1580 * cancels the expanded mode.  If it
                1590 * is a Control-N it modifies NFLAG
                1600 * to indicate that Control-N mode is
                1610 * active.  If it is a carriage return,
                1620 * it checks NFLAG to see if the
                1630 * Control-N mode is active.  If it is,
                1640 * every time a carriage return is
                1650 * printed, a Control-N is printed right
                1660 * after it.
                1670 *
0321- C9 94     1680 START    CMP #$94        Is it Control-T?
0323  F0 15     1690          BEQ CANCEL      Yes, to normal.
0325  C9 8E     1700          CMP #$8E        Is it Control-N?
0327  D0 07     1710          BNE PRINT1      No, print character.
0329  85 06     1720          STA NFLAG       Yes, NFLAG   $8E.
```

```
032B- 20 02 C2 1730 PRINT1  JSR PRINTER     Print character.
032E- C9 8D    1740         CMP #$8D        Was it a carriage return?
0330- D0 07    1750         BNE RETURN      No, next character.
0332- A5 06    1760         LDA NFLAG       Control-N active?
0334- F0 03    1770         BEQ RETURN      No, next character.
0336- 20 02 C2 1780 PRINT2  JSR PRINTER     Keep active.
0339- 60       1790 RETURN  RTS             Return.
               1800 *
               1810 *
               1820 * This subroutine cancels the Expanded
               1830 * print mode, resets the Control-N flag
               1840 * and leaves the user with the printer
               1850 * active.
               1860 *
033A- 20 02 C2 1870 CANCEL  JSR PRINTER     Print Control-T.
033D- A9 00    1880         LDA #$0         Reset NFLAG to
033F- 85 06    1890         STA NFLAG       zero.
0341- 60       1900         RTS             Return.
               1910 *
               1920 *
               1930 * This is the message printing routine.
               1940 *
0342- 85 08    1950 MSGPRT  STA TXTPTR      Save pointer to
0344- 84 09    1960         STY TXTPTR+1    text to be printed.
0346- A0 00    1970         LDY #$0
0348- B1 08    1980 LOOP    LDA (TXTPTR),Y Get character.
034A- F0 06    1990         BEQ ENDPRT      Done yet?
034C- 20 ED FD 2000         JSR COUT        No, print character.
034F- C8       2010         INY
0350- D0 F6    2020         BNE LOOP        Get next character.
0352- 60       2030 ENDPRT  RTS             Return to caller.
               2040 *
               2050 *
               2060 * This is the text printed by the program.
               2070 *
0353- C5 D0 D3
0356- CF CE A0
0359- D0 D2 C9
035C- CE D4 C5
035F- D2 A0 D0
0362- C1 D4 C3
0365- C8       2080 TEXT    .AS -"EPSON PRINTER PATCH"
0366- 8D 8D    2090         .HS 8D8D
0368- C2 D9 A0
036B- CA D5 CC
036E- C5 D3 A0
0371- C8 AE A0
0374- C7 C9 CC
0377- C4 C5 D2 2100         .AS -"BY JULES H. GILDER"
037A- 8D       2110         .HS 8D
037B- C3 CF D0
037E- D9 D2 C9
0381- C7 C8 D4
0384- A0 A8 C3
0387- A9 A0 B1
038A- B9 B8 B2 2120         .AS -"COPYRIGHT (C) 1982"
038D- 8D       2130         .HS 8D
038E- C1 CC CC
0391- A0 D2 C9
0394- C7 C8 D4
0397- D3 A0 D2
039A- C5 D3 C5
039D- D2 D6 C5
03A0- C4       2140         .AS -"ALL RIGHTS RESERVED"
03A1- 8D 8D 8D
03A4- 00       2150         .HS 8D8D8D00
```

tered. This makes it impossible to list a program out in the expanded or bold format. But all is not lost, by stealing control away from the output, as we've learned to do with the last program, we can overcome the Epson's design flaw.

The expanded mode is activated by sending a Control-N to the printer and deactivated by sending a Control-T to the printer or by the printer receiving a carriage return. So, in order to retain the printer in the expanded mode, all we have to do is detect when the printer is being sent a carriage return and the immediately after that, send the printer another Control-N. This will insure that the printer will remain in the expanded mode, or any other combination mode that requires the expand option to be active (e.g. bold). To cancel the expanded mode, a Control-T is sent.

This is exactly what the EPSON PRINTER PATCH program does. After printing out the program title (lines 1310 to 1340), the program initializes NFLAG to zero (lines 1410 and 1420). NFLAG is used to determine if a Control-N has been sent at least once to the printer, so that the program will know if it must send a Control-N after every carriage return. The next thing that the program does is to find the starting address of the program, and store that address in the output hooks (lines 1430 to 1460). As we did in the last program, a check is made to see if DOS is present and the connection to DOS is made if it is (lines 1470 to 1510).

The actual program that does the checking starts on line 1680. The first thing that is done here is to check if a Control-T has been entered. If it has, the program jumps to a routine that cancels the expanded mode (line 1870) and also stores a zero in NFLAG. If the character sent to be printed was not a Control-T, a check is made to see if it is a Control-N. If it is, the Control-N is stored in NFLAG and then sent to the printer (line 1730). If it wasn't a Control-N, the program branches to 1730 to send the character to the printer.

After the character has been printed, it still remains in the accumulator so a check can be made to see if it was a carriage return (line 1740). If it wasn't, the program executes an RTS instruction (line 1790) and returns to get the next character, if any. If it was a carriage return, the program checks NFLAG to see if the expanded mode is active. If it is, a Control-N is sent to the printer to keep it active (lines 1760 to 1780) and then an RTS instruction is executed.

For those of you who do not own Epson printers, don't despair, there are programs here that will help you too. Did you ever get frustrated because you wanted to list a program so you typed PR# <slot> and the program started listing out in 40 column format? If you have, you'll understand the frustration of having to reset, activate the printer and try to remember to tell the printer to print 80 columns wide. With the next program we're going to look at, the PRINTER SETUP PROGRAM, you'll no longer have to worry about making sure your printer is in the right mode. All you do is enter your setup string the first time you use this program, and then, for as long as power is applied and page three of memory remains intact, all you'll have to do to activate the printer in the correct mode is to CALL 825.

Set up your printer automatically

Having your printer ready to operate in the mode you desire is made possible by using a program — TEXT INPUT ROUTINE — that we developed earlier in the

book in Chapter 3. In fact, if you look at lines 1750 to 1840, you'll see exactly the same program from Chapter 3. We'll come back to it later.

If you glance quickly at the PRINTER SETUP PROGRAM, you'll see that it differs somewhat from the programs we have had until now, because all of the text that is going to be printed out is at the beginning of the program instead of at the end of it (lines 1360 to 1480). The reason for this is that the text is only going to be used once, the first time that the program is run, and we want the part that's going to be used over again to remain in page 3 of memory. If the text were at the end of the program, the section of the program we want to remain permanently would reside in page 2, which is the input buffer. Thus, if a sufficiently long line of text were entered, it would get wiped out. The JMP instruction that precedes the text messages is only there so that the program can be run from its starting address.

The program begins on line 1550, where the screen is cleared, the title and copyright notice are printed, and the user is asked what slot his printer is in. Once the slot number is entered (line 1590) it is checked to make sure that the number is in the range of 0 to 7 (lines 1610 to 1640). If it's not, then the program starts over again. Once a legal entry has been verified, the most-significant nibble is set equal to zero (line 1650), resulting in a byte that contains the actual slot number and not its ASCII equivalent. This value is stored temporarily on zero page in a memory location labelled SLOT (line 1660).

Now that the program knows what slot your printer interface card is in, it asks the user what the setup string is that the printer requires (e.g. Control-I 80N, etc.). The input of this string and the storage of it in memory is handled by the text input routine that was discussed in Chapter 3 (line 1760). The data are taken in and stored in a short buffer that starts immediately after the program ends.

In this chapter we have been discussing programs that steal control away from the output. Careful examination of the listing of the PRINTER SETUP PROGRAM, will show you that the subroutine that loads an address into $36 and $37 is conspicuously missing. How then are we affecting the output? The answer lies in that portion of the program that begins on line 1930. Here, a routine in the Apple F8 monitor ROM is used to simulate the PR# <slot> that we usually do from the keyboard or a BASIC program. Whenever a PR# <slot> is executed, $36 and $37 are automatically changed to point to the software that is in the ROMs on the interface card. For Apple's parallel interface and slot number 1 this would result in $36 and $37 containing the address $C102, low-order byte first.

Back to our program, in line 1930, the slot number that was saved earlier, is now retrieved and placed into the accumulator and a jump is made to OUTPORT, to simulate the PR # <slot>. Next, a check is made to see if DOS is present, and if it is, the new output hooks are connected through DOS (lines 1950 to 1980). With the printer now connected, the address of the buffer that contains the setup string is pointed to by the accumulator and the Y-register and the program falls into the message printing subroutine. This subroutine prints out the characters that we entered earlier and sets the printer to the proper mode. Upon hitting the RTS of the message printing subroutine, control is returned to the program that originally

called PRINTER SETUP. Once the information has been entered in the buffer, and assuming the slot number that is stored on zero page remains intact along with page 3, the printer can be initialized without doing a PR# <slot> but by simply doing a CALL 825.

```
                      1000 ************************************
                      1010 ***                              ***
                      1020 ***     PRINTER SETUP PROGRAM    ***
                      1030 ***                              ***
                      1040 ***     COPYRIGHT (C) 1982 BY    ***
                      1050 ***        JULES H. GILDER       ***
                      1060 ***      ALL RIGHTS RESERVED     ***
                      1070 ***                              ***
                      1080 ************************************
                      1090 *
                      1100 *
                      1110 *
                      1120          .OR $238
                      1130 *
                      1140 *
                      1150 * EQUATES
                      1160 *
0006-                 1170 TXTPTR   .EQ $6
0008-                 1180 SLOT     .EQ $8
0200-                 1190 IN       .EQ $200
03D0-                 1200 WARMDOS  .EQ $3D0
FC58-                 1210 HOME     .EQ $FC58
FD0C-                 1220 RDKEY    .EQ $FD0C
FD6F-                 1230 GETLN1   .EQ $FD6F
FDED-                 1240 COUT     .EQ $FDED
FE95-                 1250 OUTPORT  .EQ $FE95
                      1260 *
                      1270 *
0238- 4C 00 03        1280          JMP START       Run the program.
                      1290 *
                      1300 *
                      1310 * These are the text messages printed
                      1320 * out by the program.  The title is
                      1330 * printed first and then the printer
                      1340 * slot and setup string are requested.
                      1350 *
023B- D0 D2 C9
023E- CE D4 C5
0241- D2 A0 D3
0244- C5 D4 D5
0247- D0 A0 D0
024A- D2 CF C7
024D- D2 C1 CD        1360 TEXT1    .AS -"PRINTER SETUP PROGRAM"
0250- 8D 8D           1370          .HS 8D8D
0252- C2 D9 A0
0255- CA D5 CC
0258- C5 D3 A0
025B- C8 AE A0
025E- C7 C9 CC
0261- C4 C5 D2        1380          .AS -"BY JULES H. GILDER"
0264- 8D              1390          .HS 8D
0265- C3 CF D0
0268- D9 D2 C9
026B- C7 C8 D4
026E- A0 A8 C3
0271- A9 A0 B1
0274- B9 B8 B2        1400          .AS -"COPYRIGHT (C) 1982"
0277- 8D              1410          .HS 8D
0278- C1 CC CC
027B- A0 D2 C9
027E- C7 C8 D4
0281  D3 A0 D2
0284  C5 D3 C5
0287  D2 D6 C5
```

```
028A- C4          1420          .AS -"ALL RIGHTS RESERVED"
028B- 8D 8D 8D    1430          .HS 8D8D8D
028E- D7 C8 C1
0291- D4 A0 D3
0294- CC CF D4
0297- A0 C9 D3
029A- A0 D9 CF
029D- D5 D2 A0
02A0- D0 D2 C9
02A3- CE D4 C5
02A6- D2 A0 C9
02A9- CE BF A0    1440          .AS -"WHAT SLOT IS YOUR PRINTER IN? "
02AC- 00          1450          .HS 00
02AD- 8D 8D       1460 TEXT2    .HS 8D8D
02AF- C5 CE D4
02B2- C5 D2 A0
02B5- D4 C8 C5
02B8- A0 D3 C5
02BB- D4 D5 D0
02BE- A0 D3 D4
02C1- D2 C9 CE
02C4- C7 A0 D4
02C7- C8 C1 D4
02CA- A0 D9 CF
02CD- D5 D2 A0
02D0- D0 D2 C9
02D3- CE D4 C5
02D6- D2 CE C5
02D9- C5 C4 D3
02DC- A0 C1 CE
02DF- C4 A0 D4
02E2- C8 C5 CE
02E5- A0 D0 D2
02E8- C5 D3 D3
02EB- A0 D4 C8
02EE- C5 A0 BC
02F1- D2 C5 D4
02F4- D5 D2 CE
02F7- BE A0 CB
02FA- C5 D9 BA 1470          .AS -"ENTER THE SETUP STRING THAT YOUR
PRINTERNEEDS AND THEN PRESS THE <RETURN> KEY:"
02FD- 8D 8D 00 1480          .HS 8D8D00
               1490 *
               1500 *
               1510 * This section of code clears the
               1520 * screen and asks the user what slot
               1530 * the printer interface card is in.
               1540 *
0300- 20 58 FC 1550 START    JSR HOME        Clear screen.
0303- A9 3B    1560          LDA #TEXT1      Get pointer to
0305- A0 02    1570          LDY /TEXT1      text to print.
0307- 20 4C 03 1580          JSR MSGPRT      Ask for slot.
030A- 20 0C FD 1590          JSR RDKEY       Get response.
030D- 20 ED FD 1600          JSR COUT        Echo on screen
0310- C9 B0    1610          CMP #$B0        Check that it
0312- 90 EC    1620          BCC START       is between 0
0314- C9 B8    1630          CMP #$B8        and 7.
0316- B0 E8    1640          BCS START
0318- 29 0F    1650          AND #$F         Make it a digit.
031A- 85 08    1660          STA SLOT        Save it.
               1670 *
               1680 *
               1690 * This section asks the user for the
               1700 * required setup string and stores it
               1710 * in a buffer area.
               1720 *
031C- A9 AD    1730          LDA #TEXT2      Get pointer to
031E- A0 02    1740          LDY /TEXT2      text to print.
0320- 20 4C 03 1750          JSR MSGPRT      Ask for setup string.
0323- 20 6F FD 1760          JSR GETLN1      Get string.
0326- A0 FF    1770          LDY #$FF
0328- C8       1780 LOOP1    INY
0329- B9 00 02 1790          LDA IN,Y        Transfer string to a
```



```
032C- 99 5D 03 1800          STA BUFFER,Y    buffer area.
032F- C9 8D    1810          CMP #$8D
0331- D0 F5    1820          BNE LOOP1
0333- C8       1830          INY
0334- A9 00    1840          LDA #$0         Terminate it with
0336- 99 5D 03 1850          STA BUFFER,Y    a zero.
               1860 *
               1870 *
               1880 * This is the warm entry into the
               1890 * Printer Setup Program.  When entered
               1900 * here, the printer will be setup using
               1910 * previously entered information.
               1920 *
0339- A5 08    1930 PRNTRON LDA SLOT        Get slot number.
033B- 20 95 FE 1940          JSR OUTPORT     Do PR# <slot>.
033E- AD D0 03 1950          LDA WARMDOS     Check for DOS.
0341- C9 4C    1960          CMP #$4C
0343- D0 03    1970          BNE NODOS       No DOS, continue.
0345- 20 D0 03 1980          JSR WARMDOS     Connect through DOS.
0348- A9 5D    1990 NODOS    LDA #BUFFER     Send setup string
034A- A0 03    2000          LDY /BUFFER     to the printer.
               2010 *
               2020 *
               2030 * This is the message printing routine.
               2040 *
034C- 85 06    2050 MSGPRT   STA TXTPTR      Get pointer to
034E- 84 07    2060          STY TXTPTR+1    text to print.
0350- A0 00    2070          LDY #$0
0352- B1 06    2080 LOOP     LDA (TXTPTR),Y  Get character.
0354- F0 06    2090          BEQ ENDPRT      Done?
0356- 20 ED FD 2100          JSR COUT        No, print character.
0359- C8       2110          INY
035A- D0 F6    2120          BNE LOOP        Get next character.
035C- 60       2130 ENDPRT   RTS             Return.
               2140 *
               2150 *
               2160 * This is where the setup string buffer
               2170 * starts.
               2180 *
035D- 00       2190 BUFFER   .HS 00
```

How to TAB past 40 columns

When writing an Applesoft program that is to produce a printed report of some kind as an output, programmers often find it desirable to TAB to a location that is greater than 40. Unfortunately, Applesoft will not respond to such a command properly and instead, will treat any TAB to a position of greater than 40 as a SPC command. This can reek havoc on formatted outputs.

A solution to this problem however has been found and publicized widely by the International Apple Core. The solution they presented was a short machine language program into which the user had to plug in the appropriate values. The whole process was a bit cumbersome, so I wrote a program to automatically setup the program with the appropriate data by the user simply answering two questions. Known as the PRINTER TABBING DRIVER, this program asks the user what slot the printer interface card is in (lines 1460 to 1490). The program waits until a key is pressed (line 1500) and then echoes the character entered on the screen (line 1510).

After the number of the slot has been entered, the program checks to make sure that a number between 1 and 7 was entered (lines 1520 to 1550) and then zeros out the most significant nibble of the byte (line 1560). After storing this number inside

the LDA instruction of the tabbing program (line 1570), the number, which is still in the accumulator, has $C0 added to it (line 1580) to convert it into the high-order byte of the printer card address. This byte is then stored inside a JSR instruction in the tabbing program (line 1590).

Next, the program asks what type of printer is being used: parallel, serial or a Silentype (lines 1720 to 1740). The data entered is checked to make sure it is in the range of 1 to 3 (lines 1760 to 1790), converted to a single digit (line 1800) and the number entered is converted to a number in the range of 0 to 2 by subtracting 1 from it (lines 1810 to 1820). This number is then transferred to the X-register to be used as an index into a table (line 1830) and also temporarily stored in a zero page location labelled DEVICE (line 1840).

Using the X-register, the program proceeds to retrieve the low-order byte of the printer address from the first of three data tables and stores it inside the JSR instruction of the tabbing program (lines 1850 to 1860). Next, the program gets the low-byte of the address of the location used by the interface card to hold the column count and stores it in the tabbing program (lines 1880 and 1890). A little bit of calculation is needed to retrieve the high-order byte of the column count location. This is done in lines 1890 to 2000.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***     PRINTER TABBING DRIVER   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130 *
                1140 * EQUATES
                1150 *
0006-           1160 TXTPTR  .EQ $06
0019-           1170 DEVICE  .EQ $19
0024-           1180 CH      .EQ $24
0036-           1190 CSWL    .EQ $36
003C-           1200 A1L     .EQ $3C
003D-           1210 A1H     .EQ $3D
003E-           1220 A2L     .EQ $3E
003F-           1230 A2H     .EQ $3F
0042-           1240 A4L     .EQ $42
0043-           1250 A4H     .EQ $43
03D0-           1260 WARMDOS .EQ $3D0
03EA-           1270 CONNECT .EQ $3EA
07F9-           1280 COLCNT  .EQ $7F9
C100-           1290 PRINTER .EQ $C100
FC58-           1300 HOME    .EQ $FC58
FD0C-           1310 RDKEY   .EQ $FD0C
FDED-           1320 COUT    .EQ $FDED
FE2C-           1330 MOVE    .EQ $FE2C
FE95-           1340 OUTPORT .EQ $FE95
                1350 *
                1360 *
                1370 * This section of the program clears
                1380 * the screen, prints out the title of
                1390 * the program and asks the user what
                1400 * slot the printer card is in.  It then
                1410 * stores the slot number in the section
```



```
                1420 * of code that's going to be customized
                1430 * and adds $C0 to it and stores it as
                1440 * the high-byte of a JSR instruction.
                1450 *
0800- 20 58 FC  1460 START   JSR HOME       Clear screen.
0803- A9 CD     1470         LDA #TEXT1     Point to text
0805- A0 08     1480         LDY /TEXT1     to be printed.
0807- 20 A1 08  1490         JSR MSGPRT     Print it.
080A- 20 0C FD  1500         JSR RDKEY      Get answer.
080D- 20 ED FD  1510         JSR COUT       Echo on screen.
0810- C9 B1     1520         CMP #$B1       Make sure it's
0812- 90 EC     1530         BCC START      between 1 & 7
0814- C9 B8     1540         CMP #$B8       or start over.
0816- B0 E8     1550         BCS START
0818- 29 0F     1560         AND #$0F       Make it a digit.
081A- 8D 7A 08  1570         STA BEGIN+1    Save it.
081D- 09 C0     1580         ORA #$C0       Convert it.
081F- 8D 98 08  1590         STA WARMPRT+2  Save it.
                1600 *
                1610 *
                1620 * Here, the user is asked if output is
                1630 * to a parallel, serial or Silentype
                1640 * printer.  The entry is converted to a
                1650 * single digit and used as an index
                1660 * into tables containing the program
                1670 * modifications. Get index to TABLE3 by
                1680 * using the formula:
                1690 *
                1700 *  Index = (DEVICE NMBR -1)*7 + SLOT-1
                1710 *
0822- A9 45     1720         LDA #TEXT2     Point to text
0824- A0 09     1730         LDY /TEXT2     to be printed.
0826- 20 A1 08  1740         JSR MSGPRT     Print it.
0829- 20 0C FD  1750 REDO    JSR RDKEY      Get answer.
082C- C9 B1     1760         CMP #$B1       Make sure it
082E- 90 F9     1770         BCC REDO       is in 1 to 3
0830- C9 B4     1780         CMP #$B4       range or redo.
0832- B0 F5     1790         BCS REDO
0834- 29 0F     1800         AND #$0F       Make a digit.
0836- 38        1810         SEC            Subtract 1 to
0837- E9 01     1820         SBC #$1        make an index
0839- AA        1830         TAX            Transfer to X.
083A- 85 19     1840         STA DEVICE     Save it too.
083C- BD B2 08  1850         LDA TABLE1,X   Get printer low
083F- 8D 97 08  1860         STA WARMPRT+1  byte and save.
0842- BD B5 08  1870         LDA TABLE2,X   Get high-byte of
0845- 8D 9C 08  1880         STA COUNT+2    column count location.
0848- A5 19     1890         LDA DEVICE     Get device
084A- 0A        1900         ASL            type number
084B- 0A        1910         ASL            multiply by 7
084C- 0A        1920         ASL            (x 8 and -1)
084D- 38        1930         SEC
084E- E5 19     1940         SBC DEVICE     Save result.
0850- 18        1950         CLC
0851- 6D 7A 08  1960         ADC BEGIN+1    Add slot number.
0854- AA        1970         TAX            Transfer to X.
0855- CA        1980         DEX            Subtract 1.
0856- BD B8 08  1990         LDA TABLE3,X   Get column count
0859- 8D 9B 08  2000         STA COUNT+1    low-byte and save
                2010 *
                2020 *
                2030 * Here, the program moves the modified
                2040 * section of code down to page 3 ($300)
                2050 * where it is designed to run.
                2060 *
085C- A9 79     2070         LDA #BEGIN     Store start of
085E- 85 3C     2080         STA A1L        program address
0860- A9 08     2090         LDA /BEGIN     in A1.
0862- 85 3D     2100         STA A1H
0864- A9 A1     2110         LDA #MSGPRT    Store end of
0866- 85 3E     2120         STA A2L        program addr.
0868- A9 08     2130         LDA /MSGPRT    in A2.
086A  85 3F     2140         STA A2H
```

```
086C- A9 00     2150          LDA #$0         Store $300 in
086E- 85 42     2160          STA A4L         A4 which is
0870- A9 03     2170          LDA #$3         start of
0872- 85 43     2180          STA A4H         destination.
0874- A0 00     2190          LDY #$0         Zero Y-register.
0876- 4C 2C FE  2200          JMP MOVE        Move code.
                2210 *
                2220 *
                2230 * This is the actual tabbing driver
                2240 * program.  It gets moved down to $300
                2250 * where it is designed to run.  This
                2260 * segment of the program initializes the
                2270 * printer by doing the equivalent of a
                2280 * PR# <slot> and then sending a
                2290 * carriage return to the printer.  It
                2300 * then modifies the output hooks $36
                2310 * and $37 to point to the part of this
                2320 * program that handles TABs.
                2330 *
0879- A9 01     2340 BEGIN    LDA #$01        Load slot number into accum.
087B- 20 95 FE  2350          JSR OUTPORT     Do PR#<accum>.
087E- A9 8D     2360          LDA #$8D        Print carriage
0880- 20 ED FD  2370          JSR COUT        return.
0883- A9 1D     2380          LDA #$1D        Change output
0885- 85 36     2390          STA CSWL        hooks.
0887- A9 03     2400          LDA #$3
0889- 85 37     2410          STA CSWL+1
088B- AD D0 03  2420          LDA WARMDOS     See if DOS is
088E- C9 4C     2430          CMP #$4C        present and if
0890- D0 03     2440          BNE NODOS       so connect
0892- 20 EA 03  2450          JSR CONNECT     through it.
0895- 60        2460 NODOS    RTS             Otherwise, return.
                2470 *
                2480 *
                2490 * Here the character in the accumulator
                2500 * is printed out and also temporarily
                2510 * saved on the stack while the column
                2520 * count on the printer is picked up and
                2530 * stored in $24 (cursor horizontal
                2540 * position).
                2550 *
0896- 20 02 C1  2560 WARMPRT  JSR PRINTER+2   Print character.
0899- 48        2570          PHA             Save it on the stack.
089A- AD F9 07  2580 COUNT    LDA COLCNT      Get column count.
089D- 85 24     2590          STA CH          Save it in HTAB.
089F- 68        2600          PLA             Retrieve character from stack.
08A0- 60        2610          RTS
                2620 *
                2630 *
                2640 * This is the message printing routine.
                2650 *
08A1- 85 06     2660 MSGPRT   STA TXTPTR      Get the address
08A3- 84 07     2670          STY TXTPTR+1    of text to be
08A5- A0 00     2680          LDY #$0         printed.
08A7- B1 06     2690 LOOP     LDA (TXTPTR),Y  Get character.
08A9- F0 06     2700          BEQ ENDPRT      If done, rtn.
08AB- 20 ED FD  2710          JSR COUT        If not print.
08AE- C8        2720          INY             Increment pointer.
08AF- D0 F6     2730          BNE LOOP        Get next character.
08B1- 60        2740 ENDPRT   RTS             Return.
                2750 *
                2760 *
                2770 * These are the data tables that
                2780 * contain the modifications to the
                2790 * TAB DRIVER program.
                2800 *
08B2- 02 07 07  2810 TABLE1   .HS 020707
08B5- 07 05 CF  2820 TABLE2   .HS 0705CF
08B8- F9 FA FB
08BB- FC FD FE
08BE- FF        2830 TABLE3   .HS F9FAFBFCFDFEFF
08BF- F9 FA FB
08C2- FC FD FE
```



```
08C5- FE        2840          .HS F9FAFBFCFDFEFE
08C6- 04 04 04
08C9- 04 04 04
08CC- 04        2850          .HS 04040404040404
                2860 *
                2870 *
                2880 * These are the text messages that are
                2890 * printed out by the program.
                2900 *
08CD- D0 D2 C9
08D0- CE D4 C5
08D3- D2 A0 D4
08D6- C1 C2 C2
08D9- C9 CE C7
08DC- A0 C4 D2
08DF- C9 D6 C5
08E2- D2        2910 TEXT1    .AS -"PRINTER TABBING DRIVER"
08E3- 8D 8D     2920          .HS 8D8D
08E5- C2 D9 A0
08E8- CA D5 CC
08EB- C5 D3 A0
08EE- C8 AE A0
08F1- C7 C9 CC
08F4- C4 C5 D2  2930          .AS -"BY JULES H. GILDER"
08F7- 8D        2940          .HS 8D
08F8- C3 CF D0
08FB- D9 D2 C9
08FE- C7 C8 D4
0901- A0 A8 C3
0904- A9 A0 B1
0907- B9 B8 B2  2950          .AS -"COPYRIGHT (C) 1982"
090A- 8D        2960          .HS 8D
090B- C1 CC CC
090E- A0 D2 C9
0911- C7 C8 D4
0914- D3 A0 D2
0917- C5 D3 C5
091A- D2 D6 C5
091D- C4        2970          .AS -"ALL RIGHTS RESERVED"
091E- 8D 8D 8D  2980          .HS 8D8D8D
0921- D7 C8 C1
0924- D4 A0 D3
0927- CC CF D4
092A- A0 C9 D3
092D- A0 D9 CF
0930- D5 D2 A0
0933- D0 D2 C9
0936- CE D4 C5
0939- D2 A0 C3
093C- C1 D2 C4
093F- A0 C9 CE
0942- BF A0     2990          .AS -"WHAT SLOT IS YOUR PRINTER CARD IN? "
0944- 00        3000          .HS 00
0945- 8D 8D     3010 TEXT2    .HS 8D8D
0947- D7 C8 C1
094A- D4 A0 D4
094D- D9 D0 C5
0950- A0 CF C6
0953- A0 D0 D2
0956- C9 CE D4
0959- C5 D2 A0
095C- C9 CE D4
095F- C5 D2 C6
0962- C1 C3 C5
0965- A0 C4 CF
0968- A0 D9 CF
096B- D5 A0 A0
096E- A0 C8 C1
0971- D6 C5 BF  3020          .AS -"WHAT TYPE OF PRINTER INTERFACE DO YOU
HAVE?"
0974- 8D 8D     3030          .HS 8D8D
0976- A0 A0 A0
0979- BC B1 BK
```

```
097C- A0 D0 C1
097F- D2 C1 CC
0982- CC C5 CC 3040          .AS -"   <1> PARALLEL"
0985- 8D       3050          .HS 8D
0986- A0 A0 A0
0989- BC B2 BE
098C- A0 D3 C5
098F- D2 C9 C1
0992- CC       3060          .AS -"   <2> SERIAL"
0993- 8D       3070          .HS 8D
0994- A0 A0 A0
0997- BC B3 BE
099A- A0 D3 C9
099D- CC C5 CE
09A0- D4 D9 D0
09A3- C5       3080          .AS -"   <3> SILENTYPE"
09A4- 8D 8D    3090          .HS 8D8D
09A6- C5 CE D4
09A9- C5 D2 A0
09AC- C3 C8 CF
09AF- C9 C3 C5
09B2- BA A0    3100          .AS -"ENTER CHOICE: "
09B4- 00       3110          .HS 00
```

Finally, after the program has been properly configured, it is moved from its current location down in memory to page three, where it is designed to run (lines 2070 to 2200). After the move is made, control is returned to the calling program or mode via the RTS instruction in the MOVE routine.

The actual tabbing routine starts at line 2340 and jumps to a monitor routine that simulates the PR# <slot> command. Then a carriage return is sent to the printer to activate it (line 2360). After that, the output hooks are changed so that they point to the routine inside this program that permits the extended tabbing (lines 2380 to 2460). The routine that allows the extended tabbing starts at line 2560 and does so by allowing CH, the location that stores the horizontal position on the screen, to hold a number greater than 40. This number is picked up from the location that holds the column count for the printer. In the process, whatever is in the accumulator is temporarily stored on the stack.

To use the PRINTER TABBING DRIVER, BLOAD the program and then type CALL 2048. The program will ask you a few questions. After you have answered them, the program will end and return control to the immediate mode. This is a sign that the program has completed the initialization phase, and is ready to use. To do this, simply use the command CALL 768 instead of PR# <slot> any time you want to use the printer. The printer can still be turned off by typing PR#0.

With the growing popularity of lowercase adapters for the Apple computer, more and more programmers are writing programs that use lowercase text. While this can be helpful and even make programs appear more attractive, there is a big problem for people who don't have lowercase adapters and still want to run those programs. Because of the way that lowercase letters are implemented on the Apple computer, if they are displayed on a computer without a lowercase adapter, the lowercase text will appear like an unrelated mess of numbers and symbols.

Getting rid of lowercase letters the easy way

One way of avoiding the problem is to write programs in uppercase text only.

While this is certainly possible, it does mean that you can't take advantage of the lowercase adapter in machines that have it. A better way around the problem is to use the LOWERCASE LETTER FILTER program. This program is a short routine that steals control away from the output by replacing the output hooks (lines 1320 to 1400), and then tests each character that is to be printed to see if it is a lowercase letter. If it is, the letter is converted to it's uppercase equivalent and then printed.

The actual filtering routine, that handles the character checking and conversion starts on line 1480. Lowercase letters on the Apple fall in the ASCII code range of $E1 to $FA (a to z). Line 1480 checks to see if the letter to be printed is less than

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    LOWERCASE LETTER FILTER    ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $300
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
0036-           1180 CSWL     .EQ $36
03D0-           1190 WARMDOS  .EQ $3D0
03EA-           1200 CONNECT  .EQ $3EA
FDF0-           1210 COUT1    .EQ $FDF0
                1220 *
                1230 *
                1240 * This section of code sets up the
                1250 * output hooks at $36 and $37 so that
                1260 * any characters that are being output
                1270 * by the computer will first pass
                1280 * through this subroutine.  Also, a
                1290 * test is made to see if DOS is present
                1300 * or not.
                1310 *
0300- A9 13     1320          LDA #START      Get START low
0302- 85 36     1330          STA CSWL        byte & save it.
0304- A9 03     1340          LDA /START      Get START high
0306- 85 37     1350          STA CSWL+1      byte & save it
0308- AD D0 03  1360          LDA WARMDOS     See if DOS is
030B- C9 4C     1370          CMP #$4C        present.
030D- D0 03     1380          BNE NODOS       It isn't, return.
030F- 20 EA 03  1390          JSR CONNECT     Connect to DOS
0312- 60        1400 NODOS    RTS
                1410 *
                1420 *
                1430 * This is the actual start of the
                1440 * filter program.  If a character to be
                1450 * printed is lowercase, it is converted
                1460 * to uppercase and then printed.
                1470 *
0313- C9 E1     1480 START    CMP #$E1        Is it lowercase?
0315- 90 06     1490          BCC PRINTIT     No, use it.
0317- C9 FB     1500          CMP #$FB        Is it lowercase?
0319- B0 02     1510          BCS PRINTIT     No, use it.
031B- 29 DF     1520          AND #$DF        Yes, convert to uppercase.
031D- 4C F0 FD  1530 PRINTIT  JMP COUT1       Print the character.
```

$E1 if it is, it's printed, if not it's checked to see if it is equal to or greater than $FB. If so, it falls outside of the range defined for lowercase letters and is once again printed. However, if it falls within the specified range, the ASCII value of the character to be printed, which is in the accumulator, is ANDed with the value $DF, to convert it to an uppercase letter (line 1520) and the character is then printed (line 1530).

Because this routine is so short, it is an ideal way to write dual function programs. Your original program can be written with lowercase text and when the user runs the program he can be asked if a lowercase adapter is being used. If the answer is no, all that has to be done is a CALL 768, and then all text will appear as uppercase.

Looking at those invisible control characters

Did you ever save a program out to disk and accidentally hit two keys at the same time while entering the program name? If you did, and didn't catch your error, chances are that you wound up with a file on your disk that you couldn't load or delete. Or maybe you bought a commercial piece of software and there are some invisible files on the disk which you can't access in the direct mode.

Wouldn't it be nice if you could somehow get to those files? You can. In both cases, chances are that there are control characters imbedded in the program name. Sometimes, hitting two keys together results in a control character being generated. And frequently, programmers will imbed backspaces or other control characters in a file name to make it either invisible or inaccessible. Now, with this simple little program, you can determine immediately if any control characters have been used, because they will displayed in inverse video whenever they occur.

The program first steals control away from the normal output routines (lines 1320 to 1400) and redirects it to the program that starts in line 1530. At line 1530, the program checks to see if a carriage return has been entered. If the character is not a carriage return, the program branches to a subroutine that checks to see if any other control characters were entered (line 1540). If it is a carriage return, the accumulator is saved on the stack and the program does a subroutine jump to the routine that checks for control characters (line 1560) and prints out the inverse letter instead. On returning from the subroutine, the $8D that was stored on the stack is retrieved and printed out (lines 1570 and 1580).

The routine that checks for the presence of a control character starts in line 1680. If the character is a control character, it will be in the range of $80 to $9F. If it's not, the character is simply printed out, otherwise, the character is exclusively ORed with $80 to convert it to the $0 to $1F range (which is the range for inverse characters) and is then printed out.

This program will display all control characters, including the carriage return, which is displayed as an inverse M. If you wish to turn off the ability to display the control-M replace the $1F in location $319 with a $12. This can be done from BASIC by typing POKE 793,18. To restore the control-M feature place $1F in $319

or POKE 793,31. What this poke does is change the JSR in line 1560 from the routine that checks for control characters to an RTS instruction (e.g. nothing is done).

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    SHOW CONTROL CHARACTERS   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $300
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
0036-           1180 CSWL     .EQ $36
03D0-           1190 WARMDOS  .EQ $3D0
03EA-           1200 CONNECT  .EQ $3EA
FDF0-           1210 COUT1    .EQ $FDF0
                1220 *
                1230 *
                1240 * This section of code sets up the
                1250 * output hooks at $36 and $37 so that
                1260 * any characters that are being output
                1270 * by the computer will first pass
                1280 * through this subroutine.  Also, a
                1290 * test is made to see if DOS is present
                1300 * or not.
                1310 *
0300- A9 13     1320          LDA #START      Get START low
0302- 85 36     1330          STA CSWL        byte & save it.
0304- A9 03     1340          LDA /START      Get START high
0306- 85 37     1350          STA CSWL+1      byte & save it.
0308- AD D0 03  1360          LDA WARMDOS     See if DOS is
030B- C9 4C     1370          CMP #$4C        present.
030D- D0 03     1380          BNE NODOS       It isn't, return.
030F- 20 EA 03  1390          JSR CONNECT     Connect to DOS
0312- 60        1400 NODOS    RTS
                1410 *
                1420 *
                1430 * This is the actual start of the
                1440 * control character display program.
                1450 * Here a check is made to see if the
                1460 * character is a Control-M (carriage
                1470 * return).  If it is, an inverse M is
                1480 * printed followed by a carriage
                1490 * return.  Otherwise control is passed
                1500 * to a routine that checks to see if
                1510 * the character is a control character.
                1520 *
0313- C9 8D     1530 START    CMP #$8D        Is it Cntrl-M?
0315- D0 08     1540          BNE CHKCTRL     No, inverse it.
0317- 48        1550          PHA             Yes, save it.
0318- 20 1F 03  1560          JSR CHKCTRL     To inverse.
031B- 68        1570          PLA             Restore it.
031C- 4C 29 03  1580          JMP PRINTIT     Print a carriage return.
                1590 *
                1600 *
                1610 * Here a check is made to see if the
                1620 * character in the accumulator is a
                1630 * control character.  If it's not, it
                1640 * is printed as is.  If it is, the
                1650 * character is converted to inverse and
                1660 * then printed.
                1670 *
```

```
031F- C9 80     1680 CHKCTRL CMP #$80        See if the accumulator
0321- 90 06     1690         BCC PRINTIT     contains a
0323- C9 9F     1700         CMP #$9F        control character.
0325- B0 02     1710         BCS PRINTIT     No, print it.
0327- 49 80     1720         EOR #$80        Yes, inverse it.
0329- 4C F0 FD 1730 PRINTIT JMP COUT1       Print character.
```

Black-on-white video with no hardware modifications

The Apple computer as it is delivered from the manufacturer normally displays text as white characters on a black background. Some people like to read black text on a white background however. To do this, they have developed a fairly simple hardware modification. But hardware changes are not necessary. By simply using a short machine language routine, called SCREEN REVERSER, you can implement this black-on-white feature on any Apple with no hardware modifications.

Once again, to implement this feature we have to steal control away from the normal output routines and direct it to our program (lines 1330 to 1410). The output is redirected to line 1800 where the program checks to see if the character to be printed is an alphanumeric character. If it's not, the whole screen is reversed (line 1810). If it is, the character is converted to an inverse character by ANDing it with the value $3F. It and the value in the Y-register are then saved on the stack and the routine that reverses the entire screen is set up so that only the last line on the screen is reversed (lines 1870 and 1880). Then the program branches to line 1560 which is in the middle of the screen reversing routine.

The screen reversing routine starts on line 1500, where the character that is currently in the accumulator is stored on the stack along with the Y-register (lines 1500 to 1520). Next, the Y-register is set equal to zero and the low-order byte of POINTER is set equal to zero (lines 1530 and 1540). Then the high-order byte of POINTER is set to 4 so that POINTER and POINTER+1 contain the address $400, which is the start of the screen storage area (lines 1550 and 1560).

The screen is reversed by using POINTER to point to the next character to be picked up off the screen and inverted. The character is retrieved from the screen in line 1570 and converted to an inverse character in line 1580 where it is ANDed with $3F. Then, the character is placed back on the screen in its original position (line 1590) and the Y-register is incremented so that the next character can be retrieved. This process continues until 256 characters have been converted.

After 256 characters, the high byte of POINTER is incremented by one (line 1620). After its incremented, a check is made to see if the end of the video screen has been reached (lines 1630 and 1640). If not, the next character is retrieved and reversed (line 1650). Otherwise, the contents of the Y-register and accumulator before the routine was entered are retrieved from the stack and restored (lines 1660 to 1680). Finally, the character in the accumulator is output (line 1690).

Because of the relative jump in line 1890, the program is relocatable. This means that although this program was assembled to run at $300, it can be moved anywhere in memory and run. The only change required is the address that is stored in the output hooks (lines 1330 and 1340).


```
              1000 ************************************
              1010 ***                              ***
              1020 ***        SCREEN REVERSER       ***
              1030 ***                              ***
              1040 ***     COPYRIGHT (C) 1982 BY    ***
              1050 ***       JULES H. GILDER        ***
              1060 ***      ALL RIGHTS RESERVED     ***
              1070 ***                              ***
              1080 ************************************
              1090 *
              1100 *
              1110 *
              1120         .OR $300
              1130 *
              1140 *
              1150 *
              1160 * EQUATES
              1170 *
0018-         1180 POINTER .EQ $18
0036-         1190 CSWL    .EQ $36
03D0-         1200 WARMDOS .EQ $3D0
03EA-         1210 CONNECT .EQ $3EA
FDF0-         1220 COUT1   .EQ $FDF0
              1230 *
              1240 *
              1250 * This section of code sets up the
              1260 * output hooks at $36 and $37 so that
              1270 * any characters that are being output
              1280 * by the computer will first pass
              1290 * through this subroutine.  Also, a
              1300 * test is made to see if DOS is present
              1310 * or not.
              1320 *
0300- A9 35   1330         LDA #START       Get START addr
0302- A0 03   1340         LDY /START       and store it
0304- 85 36   1350         STA CSWL         in output
0306- 84 37   1360         STY CSWL+1       hooks.
0308- AD D0 03 1370        LDA WARMDOS      See is DOS is
030B- C9 4C   1380         CMP #$4C         present.
030D- D0 03   1390         BNE NODOS        No, set hooks.
030F- 20 EA 03 1400        JSR CONNECT      Connect to DOS
0312- 60      1410 NODOS   RTS
              1420 *
              1430 *
              1440 * This is a routine that picks up every
              1450 * character on the screen, converts it
              1460 * to an inverse character by ANDing
              1470 * with #$3F and puts it back on the
              1480 * screen where it came from.
              1490 *
0313- 48      1500 REVERSE PHA              Save the
0314- 98      1510         TYA              accumulator
0315- 48      1520         PHA              and Y-register
0316- A0 00   1530         LDY #$0          Set to start
0318- 84 18   1540         STY POINTER      of video
031A- A9 04   1550         LDA #$4          display.
031C- 85 19   1560 LOOP1   STA POINTER+1
031E- B1 18   1570 LOOP2   LDA (POINTER),Y  Get character.
0320- 29 3F   1580         AND #$3F         Inverse it.
0322- 91 18   1590         STA (POINTER),Y  Put it back.
0324- C8      1600         INY              Increment character count.
0325- D0 F7   1610         BNE LOOP2        Done a page?
0327- E6 19   1620         INC POINTER+1    Yes increment page.
0329- A9 08   1630         LDA #$8          All video
032B- C5 19   1640         CMP POINTER+1    inversed?
032D- D0 EF   1650         BNE LOOP2        No, do more.
032F- 68      1660         PLA              Yes restore
0330- A8      1670         TAY              Y-register and
0331- 68      1680         PLA              accumulator.
0332- 4C F0 FD 1690        JMP COUT1        Output character.
              1700 *
              1710 *
              1720 * This is where characters to be
              1730 * outputted by the computer go to first.
```

```
                1740 * Here a check is made to see if an
                1750 * alphanumeric character is being sent.
                1760 * If so, it is inversed and printed and
                1770 * only the last line is reversed. If
                1780 * not, the whole screen is reversed.
                1790 *
0335- C9 A0     1800 START   CMP #$A0          Is it alphanumeric?
0337- 90 DA     1810         BCC REVERSE      No, reverse whole screen.
0339- 29 3F     1820         AND #$3F         Yes, reverse
033B- 48        1830 SAVEAY  PHA              and save it
033C- 98        1840         TYA              and Y-register
033D- 48        1850         PHA
033E- A0 D0     1860         LDY #$D0         Set to reverse
0340- A9 07     1870         LDA #$7          the last line on
0342- B8        1880         CLV              the video screen.
0343- 50 D7     1890         BVC LOOP1
```

Format your text into pages

When I first got my Apple computer and was writing BASIC programs, I always wished that there was a way that I could print out the program listings as individual pages instead of as one continuous listing. Breaking it down into pages makes it easy to organize in a folder or looseleaf binder.

Eventually, some printer manufacturers realized that programmers wanted this capability and built it into their printers. But there are still many printers available without this feature, so here is a short machine language program that will allow you to format any kind of printed text into pages of any length with any number of lines. The program is set up to print 60 lines on a page, and the page length is set to 66 lines per page. In addition, the program can be set to allow for a pause after each page is printed, which is the default condition. But by changing one byte, this feature can be eliminated and printing will complete automatically.

The program starts by initializing the carriage return, or line, counter to $FF, and then sets up the output hooks so that they point to the start of the line counting program (lines 1340 to 1440). The first thing that the new output routine does when it is hooked in is to print the character that is currently in the accumulator (line 1540). Since the printing process is non-destructive, the character is still in the accumulator when the program returns from the printing operation. Thus, it can be checked to see if the character that was printed was a carriage return (line 1550). If it wasn't, the program executes an RTS instruction and returns to get the next character (line 1560). If it was, the line count (COUNT) is incremented by one (line 1570) and a check is made to see if 60 lines were printed yet (lines 1580 and 1590). If not, the program returns to get the next character (line 1600).

If at least 60 lines have already been printed, the program checks to see if 66 lines have been printed (lines 1610 and 1620). When 66 lines have been printed, the program branches to line 1670 where the line count is reset to zero (line 1630), and then falls into a routine (line 1760) that allows the program to pause after each page is printed. This is done by loading the accumulator with a $1 (line 1770) and then checking to see if the value in the accumulator is zero (line 1780). Since it isn't, the program then reads the keyboard within a loop until a key is pressed (lines 1790 and 1800). Once a key is pressed, the strobe is cleared (line 1810) and the program returns to get the next character (line 1820).

By changing the value in location $337 from a one to a zero, the pause after each page is printed can be eliminated. This can be done from the monitor or by typing POKE 823,0 from BASIC. The pause feature can be restored by typing POKE 823,1.

If at least 60 lines have been printed and less than 66 lines have been printed, the program proceeds to print out carriage returns until 66 lines have been printed. This takes place in line 1640 to 1660. A relative jump is used here instead of an absolute jump so that the program can be moved to and used from any location in memory.

The number of lines printed per page is set to 60 in line 1590. This can be changed from BASIC by typing POKE 803,X where X represents the number of lines you wish printed on each page. The page length, also referred to as form length, is determined in line 1620. It is set to a default value of 66 lines per page. This is standard for an 11-inch long page printed at 6 lines per inch. If your page is of a different length, or you are printing at a different density (maybe 8 lines per inch) then just multiply the lines per inch that your printer works at by the length of the paper being used (in inches). To change the page length, change the value that is stored in location $329 to whatever value you desire. From BASIC, you can change this value by typing POKE 809,X, where X is the new page length.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***        PAGE FORMATTER        ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***       JULES H. GILDER        ***
                1060 ***     ALL RIGHTS RESERVED      ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $300
                1130 *
                1140 *
                1150 * EQUATES
                1160 *
0018-           1170 COUNT    .EQ $18
0036-           1180 CSWL     .EQ $36
0200-           1190 IN       .EQ $200
03D0-           1200 WARMDOS  .EQ $3D0
03EA-           1210 CONNECT  .EQ $3EA
C000-           1220 KYBRD    .EQ $C000
C010-           1230 STROBE   .EQ $C010
C200-           1240 SLOT     .EQ $C200
FD8E-           1250 CROUT    .EQ $FD8E
FE95-           1255 OUTPORT  .EQ $FE95
                1260 *
                1270 *
                1280 * This routine turns on the printer
                1285 * in slot 1, initializes the carriage
                1290 * return counter and sets the output
                1300 * hooks to point to this program, which
                1310 * counts the number of carriage returns
                1320 * printed and pages the output.
                1325 *
02FB- A9 01     1330          LDA #$1         Set slot 1 for output
02FD- 20 95 FE  1335          JSR OUTPORT     and turn it on.
```

```
0300- A0 FF    1340          LDY #$FF        Initilize carriage return
0302- 84 18    1350          STY COUNT       counter.
0304- A9 17    1360          LDA #START      Set output
0306- 85 36    1370          STA CSWL        hooks to this
0308- A9 03    1380          LDA /START      program.
030A- 85 37    1390          STA CSWL+1
030C- AD D0 03 1400          LDA WARMDOS     Check to see
030F- C9 4C    1410          CMP #$4C        if DOS is present
0311- D0 03    1420          BNE NODOS       It isn't, return.
0313- 20 EA 03 1430          JSR CONNECT     Yes it is, connect
0316- 60       1440 NODOS    RTS             to it.
               1450 *
               1460 *
               1470 * This is the actual start of the code
               1480 * that counts the number of carriage
               1490 * returns already sent.  If 60 have
               1500 * been sent, then send 6 more carriage
               1510 * returns to get to the top of the next
               1520 * page.
               1530 *
0317- 20 02 C2 1540 START    JSR SLOT+2      Print character.
031A- C9 8D    1550          CMP #$8D        Is it a carriage return?
031C- D0 24    1560          BNE RTN         No, get next character.
031E- E6 18    1570          INC COUNT       Yes, add 1 to COUNT
0320- A5 18    1580          LDA COUNT
0322- C9 3C    1590          CMP #$3C        Does COUNT = 60?
0324- D0 1C    1600          BNE RTN         No, return.
0326- A5 18    1610 CHKPAGE  LDA COUNT
0328- C9 42    1620          CMP #$42        Does COUNT = 66?
032A- F0 06    1630          BEQ RSTCNT      Yes, reset it to zero.
032C- 20 8E FD 1640          JSR CROUT       No, print a carriage return.
032F- B8       1650          CLV             Relative jump
0330- 50 F4    1660          BVC CHKPAGE     always taken.
0332- A9 00    1670 RSTCNT   LDA #$0         Reset COUNT to
0334- 85 18    1680          STA COUNT       zero.
               1690 *
               1700 *
               1710 * This section allows the user to press
               1720 * any key to continue printing after
               1730 * each page.  To eliminate this feature
               1740 * change the LDA #$1 to an LDA #$0, or
               1750 * eliminate the instruction altogether.
               1760 *
0336- A9 01    1770          LDA #$1
0338- F0 08    1780          BEQ RTN
033A- AD 00 C0 1790 RDKYBRD  LDA KYBRD       Read keyboard until
033D- 10 FB    1800          BPL RDKYBRD     a key is pressed.
033F- 2C 10 C0 1810          BIT STROBE      Clear the strobe.
0342- 60       1820 RTN      RTS             Return to caller.
```

Send your output to the disk instead of the printer

Sometimes, it is desirable to send the output that would normally be printed, to a disk as a text file for printing at a later time, to be fixed up with an editor or to be formatted with a word processor. Writing a text file is very easy from BASIC, all you do is open the file, issue the write command and from then on, until the file is closed, everything that gets printed out is sent to the text file.

Writing a text file from a machine language program is not quite as simple. There are two ways it can be done. One is to use the file manager in DOS, but that requires intimate knowledge of how the file manager works and can be somewhat confusing. An easier method is to fool the computer into thinking that an Applesoft program is running, when a machine language program is really running, and then issuing the OPEN and WRITE commands from the machine language program.


```
               1000 ************************************
               1010 ***                              ***
               1020 ***     PRINT TO DISK SPOOLER    ***
               1030 ***                              ***
               1040 ***     COPYRIGHT (C) 1982 BY    ***
               1050 ***        JULES H. GILDER       ***
               1060 ***      ALL RIGHTS RESERVED     ***
               1070 ***                              ***
               1080 ************************************
               1090 *
               1100 *
               1110 *
               1120 *
               1130          .OR $2EE
               1140 *
               1150 *
               1160 * EQUATES
               1170 *
0006-          1180 TXTPTR   .EQ $6
0008-          1190 SAVPRM   .EQ $8
0009-          1200 SAVLANG  .EQ $9
0033-          1210 PROMPT   .EQ $33
0036-          1220 CSWL     .EQ $36
0075-          1230 CURLIN   .EQ $75
03EA-          1240 CONNECT  .EQ $3EA
AAB6-          1250 LANG     .EQ $AAB6
FC58-          1260 HOME     .EQ $FC58
FDED-          1270 COUT     .EQ $FDED
FDF0-          1280 COUT1    .EQ $FDF0
               1290 *
               1300 *
               1310 * This section of the program fools the
               1320 * computer into thinking that Applesoft
               1330 * is running instead of a machine
               1340 * language program.  It then opens a
               1350 * file called TEXT FILE and sets it up
               1360 * to be written to. Then the output
               1370 * hooks are set up so that they point
               1380 * to a routine that checks for a 0
               1390 * which is an end-of-file marker.
               1400 *
02EE- 20 58 FC 1410          JSR HOME        Clear screen.
02F1- A5 33    1420          LDA PROMPT      Save current
02F3- 85 08    1430          STA SAVPRM      prompt.
02F5- A9 06    1440          LDA #$6         Change prompt
02F7- 85 33    1450          STA PROMPT      to run value.
02F9- AD B6 AA 1460          LDA LANG        Save current
02FC- 85 09    1470          STA SAVLANG     language flag.
02FE- A9 40    1480          LDA #$40        Tell computer
0300- 8D B6 AA 1490          STA LANG        APPLESOFT is running.
0303- 85 76    1500          STA CURLIN+1    Show run mode.
0305- A9 45    1510          LDA #TEXT1      Print title and
0307- A0 03    1520          LDY /TEXT1      open TEXT FILE
0309- 20 34 03 1530          JSR MSGPRT
030C- A9 18    1540          LDA #START      Point to new
030E- A0 03    1550          LDY /START      output routine
0310- 85 36    1560          STA CSWL
0312- 84 37    1570          STY CSWL+1
0314- 20 EA 03 1580          JSR CONNECT     Connect to DOS
0317- 60       1590          RTS
               1600 *
               1610 *
               1620 * This routine checks for end-of-file.
               1630 * If it is found all values changed on
               1640 * entry are restored.
               1650 *
0318- C9 00    1660 START    CMP #$0         End of file?
031A- F0 03    1670          BEQ DONE        Yes, end up.
031C- 4C F0 FD 1680          JMP COUT1       No, print character.
031F- A9 B9    1690 DONE     LDA #TEXT2      Close TEXT
0321- A0 03    1700          LDY /TEXT2      FILE.
0323- 20 34 03 1710          JSR MSGPRT
0326- A5 08    1720          LDA SAVPRM      Restore prompt.
```

```
0328- 85 33     1730          STA PROMPT
032A- A5 09     1740          LDA SAVLANG     Restore
032C- 8D B6 AA  1750          STA LANG        language flag.
032F- A9 FF     1760          LDA #$FF        Indicate
0331- 85 76     1770          STA CURLIN+1    immediate mode
0333- 60        1780          RTS             Return.
                1790 *
                1800 *
                1810 * This is the message printing routine.
                1820 *
0334- 85 06     1830 MSGPRT   STA TXTPTR
0336- 84 07     1840          STY TXTPTR+1
0338- A0 00     1850          LDY #$0
033A- B1 06     1860 LOOP     LDA (TXTPTR),Y
033C- F0 06     1870          BEQ ENDPRT
033E- 20 ED FD  1880          JSR COUT
0341- C8        1890          INY
0342- D0 F6     1900          BNE LOOP
0344- 60        1910 ENDPRT   RTS
                1920 *
                1930 *
                1940 * These are the various text mesages
                1950 * printed out by this program.
                1960 *
0345- D0 D2 C9
0348- CE D4 A0
034B- D4 CF A0
034E- C4 C9 D3
0351- CB A0 D3
0354- D0 CF CF
0357- CC C5 D2  1970 TEXT1    .AS -"PRINT TO DISK SPOOLER"
035A- 8D 8D     1980          .HS 8D8D
035C- C2 D9 A0
035F- CA D5 CC
0362- C5 D3 A0
0365- C8 AE A0
0368- C7 C9 CC
036B- C4 C5 D2  1990          .AS -"BY JULES H. GILDER"
036E- 8D        2000          .HS 8D
036F- C3 CF D0
0372- D9 D2 C9
0375- C7 C8 D4
0378- A0 A8 C3
037B- A9 A0 B1
037E- B9 B8 B2  2010          .AS -"COPYRIGHT (C) 1982"
0381- 8D        2020          .HS 8D
0382- C1 CC CC
0385- A0 D2 C9
0388- C7 C8 D4
038B- D3 A0 D2
038E- C5 D3 C5
0391- D2 D6 C5
0394- C4        2030          .AS -"ALL RIGHTS RESERVED"
0395- 8D 8D 8D
0398- 8D 84     2040          .HS 8D8D8D8D84
039A- CF D0 C5
039D- CE D4 C5
03A0- D8 D4 A0
03A3- C6 C9 CC
03A6- C5        2050          .AS -"OPENTEXT FILE"
03A7- 8D 84     2060          .HS 8D84
03A9- D7 D2 C9
03AC- D4 C5 D4
03AF- C5 D8 D4
03B2- A0 C6 C9
03B5- CC C5     2070          .AS -"WRITETEXT FILE"
03B7- 8D 00     2080          .HS 8D00
03B9- 8D 84     2090 TEXT2    .HS 8D84
03BB- C3 CC CF
03BE- D3 C5 A0
03C1- D4 C5 D8
03C4- D4 A0 C6
03C7- C9 CC C5  2100          .AS -"CLOSE TEXT FILE"
03CA- 8D 00     2110          .HS 8D00
```


This is the approach used in the program PRINT TO DISK SPOOLER.

The program starts by clearing the screen (line 1410) in preparation for writing the title and copyright notice out and then saves the current value of certain locations normally associated with Applesoft. In lines 1420 to 1450, the current value of PROMPT is saved and replaced by the value that is normally found when an Applesoft program is running. Next, the data that tells the Apple which language is active, is saved and replaced by information that tells it that Applesoft is running (lines 1460 to 1490). Now, if Applesoft is active, it must be operating on a particular line. So, in line 1500, we give the computer a phony line number so it will be convinced that an Applesoft program is running.

With all the details that fool it into thinking that an Applesoft program is running taken care of, the program now goes on to print out the title of the program (lines 1510 to 1530). The next thing that it does is to open a file. The name of the file is automatically set to TEXT FILE in line 2050. Since the computer thinks an Applesoft program is running, to open the file all we have to do is print at least one carriage return and then a control-D, followed by the phrase OPEN TEXT FILE. This is done in lines 2040 and 2050. Following that, another carriage return and control-D are sent and the WRITE TEXT FILE command is issued.

Before returning control to the caller, the program gets the starting address of the routine that will automatically close the open file when a zero is encountered. This address is then placed in the output hooks (lines 1540 to 1580) and the program returns ready to start writing text to the file. Everything that is printed out will be stored in the text file as well until an ASCII zero is sent either by the machine language program or from BASIC by sending CHR$(0).

The routine that checks for the zero starts at line 1660. If the character is not a zero, it is printed out (line 1680). But if it is a zero, a carriage return and a control-D are sent, followed by the words CLOSE TEXT FILE and another carriage return. This closes the file and prevents anything further from being written to the file. After that, the program restores the prompt and the language flag that were stored at the beginning of the program (lines 1720 to 1750) and stores an $FF in the high-order byte of the current line number storage location to indicate that a program is not running and that the computer is in the immediate mode (lines 1760 and 1770). Control is then returned to the calling program in line 1780.

To use this subroutine, have your main machine language program do a JSR $2EE, or from BASIC do a CALL 750. All output will then go to a text file. To stop output to the file, print out an ASCII 0 or CHR$(0).

Chapter 5

STEALING CONTROL OF THE INPUT

In the last chapter we saw how it was possible to steal control away from the Apple's normal output routines and direct the computer to send characters destined to be printed to our programs instead. It's possible to do the same thing with characters that are being input to the Apple. Frequently, it is desirable to replace the routine that manages the entry of data from the keyboard with another program that either fetches the data from some other device (such as a disk drive) or first processes the data being entered.

When writing a replacement input routine, there are several things you must bear in mind. If the information being entered is to be echoed on the screen, the contents of the accumulator must be stored at (BASL),Y — where BASL equals $28. If the new input routine prevents the ESCape key and the Control-U key from being entered, then the Y-register need not be saved. If it doesn't, however, then the Y-register must remain unaltered. This can be accomplished by saving the register when entering the routine and restoring it when leaving. In all cases, the X-register must remain unaltered, so if it is needed, the same storing and restoring procedure will be necessary.

The basic read-a-key routine is located in the Apple monitor ROM at location $FD0C. Here the monitor picks up a character off the screen, converts it to its flashing equivalent, stores it back on the screen (this is the way we get the flashing box cursor). Then the program does an indirect jump through KSWL to the routine that handles the inputting of data. In an Apple system with no DOS, the input hooks are set for $FD1B, which is the location immediately following the indirect jump. In a system with DOS, it is set for $9E81, which is the routine that checks for DOS commands.

Customize your cursor

To illustrate how to steal control away from the input, the first program in this chapter will show you how to replace the blinking white square that is normally used as a cursor, with any other character you desire. In the example given, an underline character is used.

The program starts off in lines 1320 to 1400 by stealing control from the input in much the same way that we learned to steal control from the output. The big difference here, comes in lines 1340 and 1350, where instead of using locations CSWL and CSWL + 1, we use locations KSWL and KSWL + 1 ($38 and $39) which are the input hook locations.

The actual replacement routine starts on line 1480 where the character that is currently in the accumulator is saved temporarily by pushing it on the stack. Next, the character that is going to be used as the cursor character is loaded into the accumulator (line 1490) and stored on the screen (line 1500). Finally, the character that was in the accumulator is restored and the program checks to see if a key has

```
                1000 ************************************
                1010 ***                              ***
                1020 ***          CUSTOM CURSOR       ***
                1030 ***                              ***
                1040 ***      COPYRIGHT (C) 1982 BY   ***
                1050 ***         JULES H. GILDER      ***
                1060 ***       ALL RIGHTS RESERVED    ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130          .OR $300
                1140 *
                1150 *
                1160 *
                1170 * EQUATES
                1180 *
0028-           1190 BASL     .EQ $28
0038-           1200 KSWL     .EQ $38
03D0-           1210 WARMDOS  .EQ $3D0
03EA-           1220 CONNECT  .EQ $3EA
C000-           1230 KEYBD    .EQ $C000
C010-           1240 KBDSTRB  .EQ $C010
                1250 *
                1260 *
                1270 * This section steals control of the
                1280 * input and passes all characters to
```

```
                1290 * be output to the routine beginning
                1300 * with START.
                1310 *
0300- A9 13     1320          LDA #START      Get the address of the
0302- A0 03     1330          LDY /START      start of the program.
0304- 85 38     1340          STA KSWL        Store it in the
0306- 84 39     1350          STY KSWL+1      input hooks.
0308- AD D0 03  1360          LDA WARMDOS     Is DOS present?
030B- C9 4C     1370          CMP #$4C
030D- D0 03     1380          BNE NODOS       No.
030F- 20 EA 03  1390          JSR CONNECT     Yes, connect to DOS.
0312- 60        1400 NODOS    RTS             Return.
                1410 *
                1420 *
                1430 * This routine replaces the normal
                1440 * flashing cursor with whatever
                1450 * cursor character you want.  Here, an
                1460 * underscore is used as the cursor.
                1470 *
0313- 48        1480 START    PHA             Save the accumulator.
0314- AD 28 03  1490          LDA CURSOR      Get the cursor character.
0317- 91 28     1500          STA (BASL),Y    Put it on the screen.
0319- 68        1510          PLA             Restore the accumulator.
031A- 2C 00 C0  1520 GETKEY   BIT KEYBD       Key pressed?
031D- 10 FB     1530          BPL GETKEY      No, keep checking.
031F- 91 28     1540          STA (BASL),Y    Yes, display accumulator.
0321- AD 00 C0  1550          LDA KEYBD       Get keypress.
0324- 2C 10 C0  1560          BIT KBDSTRB     Clear keyboard strobe.
0327- 60        1570          RTS
                1580 *
                1590 *
                1600 * This is where the cursor character is
                1610 * stored.
                1620 *
0328- 9F        1630 CURSOR   .HS 9F
```

been pressed (line 1520). If it hasn't, the program loops back to 1520 and waits until a key has been pressed (line 1530). Once a key has been pressed, the character that is in the accumulator is displayed on the screen (line 1540). Next, the hexadecimal value of the key that was pressed is retrieved and the keyboard strobe is cleared to prepare the keyboard for the next keypress (lines 1550 to 1560).

The character that is used as the cursor is stored in the location labelled CURSOR. Here an underline is used ($9F), but any other character can be used as well.

While the previous program was useful to illustrate how to steal control away from the input routines, in all honesty I must admit that the application is one of the less urgently needed programs. A much more useful and practical program is the SCREEN PRINTER program presented next. By simply pressing two keys, this program will allow you to print out the text screen exactly as it appears on your video display, onto a parallel or serial printer.

Dump your screen to a printer

As it is currently set up, this program will run with a parallel printer card in slot 2. This can be changed however by simply changing the value of WARMPRT, which here is $C202. WARMPRT is the address to which the output hooks are set after the printer card has been initialized. In order to find out what the value of WARMPRT is for your printer interface card, simply activate it by doing a PR#<slot> and while it's active typing:

CALL -151
36.37

If you do this, the computer will respond with two numbers which represent the low and high bytes respectively of WARMPRT. For a parallel printer in slot 2 we get:

0036- 02 C2

and for a serial printer in slot one we would get:

0036- 07 C1

The C1 and C2 represent slots 1 and 2 respectively. The low-order byte of WARMPRT will vary with the interface card used. The values presented here are for interface cards from Apple Computer Inc. Other manufacturer's cards could have different values. For example, one parallel printer interface card that I have has a low-order byte of 21 instead of 2.

Getting back to the program, it starts out on line 1400 by retrieving the high-order byte of WARMPRT and ANDs it with $0F (line 1410) to get the slot number that the printer interface card is in. Once the program knows the slot number (which is now in the accumulator) it uses that number to simulate a PR#<slot> using the monitor's OUTPORT routine (line 1420). This turns the printer on. Next, a set-up string is sent to the printer (lines 1430 to 1480). This string of characters consists of Control-I 40N. The reason for sending this to the printer is that it will turn off the screen any time that the printer is activated. After the printer has been initialized, it is turned off (lines 1490 and 1500) until it is needed.

Now that the initialization phase has been completed, the program replaces the input hooks with the address of this program (lines 1600 to 1680). The replacement input routine starts on line 1740, where the first thing that happens is a subroutine jump to the monitor's KEYIN routine. Once a character has been entered by this routine, a check is made to see if that character was a Control-P (line 1750). Control-P was the character chosen as the signal to the program that a dump of the screen on the printer is wanted. Any other value could be used as well by simply replacing the $90 in line 1750 with the desired character. If the character was not a Control-P, the program returns to get the next character. If it was, it branches to the PRTSCRN routine that starts at line 1830.

The first thing that the PRTSCRN routine does is to save all of the registers (using the monitor ROM's SAVE routine) so they may be restored before the program is exited. In addition to that, the current horizontal location of the cursor is saved so that it too may be restored later on (lines 1840 and 1850). Next, the printer is turned on through its warm start address by sending it a carriage return (lines 1860 and 1870). The warm start address is used so that all of the printer initialization that was done previously will remain in effect. If the cold start ad-

```
                1000 ************************************
                1010 ***                              ***
                1020 ***        SCREEN PRINTER        ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $300
                1130 *
                1140 *
                1150 * EQUATES
                1160 *
0024-           1170 CH       .EQ $24
0028-           1180 BASL     .EQ $28
0038-           1190 KSWL     .EQ $38
03D0-           1200 WARMDOS  .EQ $3D0
03EA-           1210 CONNECT  .EQ $3EA
C202-           1220 WARMPRT  .EQ $C202
FBC1-           1230 BASCALC  .EQ $FBC1
FD0C-           1240 RDKEY    .EQ $FD0C
FD1B-           1250 KEYIN    .EQ $FD1B
FDED-           1260 COUT     .EQ $FDED
FE95-           1270 OUTPORT  .EQ $FE95
FF3F-           1280 RESTORE  .EQ $FF3F
FF4A-           1290 SAVE     .EQ $FF4A
                1300 *
                1310 *
                1320 * Here the printer is initialized by
                1330 * doing the equivalent of a PR# <slot>.
                1340 * After that, the screen is turned off
                1350 * by sending the printer a set-up
                1360 * string consisting of Control-I 40N.
                1370 * Finally, the printer is turned off by
                1380 * doing the equivalent of a PR#0.
```

dress were used (e.g. $C200), the set-up string would have to be sent to the printer again. This way it doesn't.

Now that the printer is ready to print without disturbing the screen (we turned it off remember?) the X-register, and subsequently the accumulator, are loaded with the number of the first line we want to print. In hexadecimal we want to print lines $0 to $17 which is 0 to 23 in decimal. The Apple video screen in structured in a peculiar manner and text that appears continuous on the screen, is not continuous in memory. To handle this strange layout, a special routine is used to calculate the starting address in memory of any particular line. The routine is called BASCALC and it is located at $FBC1 in the Apple's monitor ROM. To use it, all you have to do is place the number of the line you want in the accumulator and then jump to BASCALC. This is what is done in lines 1890 and 1900. Upon returning from this subroutine, the starting address in memory of the desired line is found in locations BASL and BASL + 1 ($28 and $29) on page zero.

Once the program knows where the line starts in memory, it's easy to pick up the characters off the screen and send them to the printer. The routine starting at line 1910 does just that, and more. Indirect indexed addressing is used in line 1920 to retrieve a character from the desired line on the screen. Once retrieved, the character is tested to see if it is a normal white-on-black character. If it's not, $40 is added to it (line 1950) and it is checked again to see if it's normal. If it's still not normal,


```
                1390 *
0300- A9 02     1400          LDA #WARMPRT Get printer
0302- 29 0F     1410          AND #$0F     slot number.
0304- 20 95 FE  1420          JSR OUTPORT  Do PR#<slot>.
0307- A0 00     1430          LDY #$0      Send printer
0309- B9 6C 03  1440 LOOP     LDA TEXT,Y   the set-up
030C- F0 06     1450          BEQ NEXT     string.
030E- 20 ED FD  1460          JSR COUT
0311- C8        1470          INY
0312- D0 F5     1480          BNE LOOP
0314- A9 00     1490 NEXT     LDA #$0      Do a PR#0.
0316- 20 95 FE  1500          JSR OUTPORT
                1510 *
                1520 *
                1530 * This routine transfers the input
                1540 * hooks ($38 and $39) from the keyboard
                1550 * to this program, where a check can be
                1560 * made to determine if a Control-P has
                1570 * been pressed.  If so, the screen is
                1580 * then printed out.
                1590 *
0319- A9 2C     1600          LDA #START   Get program's
031B- A0 03     1610          LDY /START   start address.
031D- 85 38     1620          STA KSWL     Store in input
031F- 84 39     1630          STY KSWL+1   hooks.
0321- AD D0 03  1640          LDA WARMDOS  Check to see
0324- C9 4C     1650          CMP #$4C     if DOS is present
0326- D0 03     1660          BNE NODOS    No, return.
0328- 20 EA 03  1670          JSR CONNECT  Yes, connect
032B- 60        1680 NODOS    RTS          to it.
                1690 *
                1700 *
                1710 * Here the keyboard is checked to see
                1720 * if a Control-P has been entered.
                1730 *
032C- 20 1B FD  1740 START    JSR KEYIN    Get a key.
032F- C9 90     1750          CMP #$90     Is it Ctrl-P?
0331- F0 01     1760          BEQ PRTSCRN  Yes print screen.
0333- 60        1770          RTS          No, return.
```

another $40 is added to it. By this point all characters must be normal. Characters that appear on the screen in inverse video fall in the $0 to $3F range and thus must pass through the CHKNORM loop twice to get $80 added on to their value, while flashing characters, which are in the $40 to $7F range when displayed on the screen, only have to pass through this loop once.

After a character has been converted to normal it is printed in line 1970. Then the Y-register, which is used to point to the particular character on the line that is being accessed, is incremented in preparation for retrieving the next character. Before doing that, a check is made to see if we've already picked up the last character on the line (line 1990). If we have not, the next character is retrieved (line 2000). But if we have, a carriage return is sent to the printer, and the X-register, which is used as the line counter, is incremented by one (lines 2010 to 2030). Before continuing, a check is made to see if we have just finished printing the last line on the screen (line 2040). If we have not, the program jumps back to line 1890 where it gets the address in memory of the next line and continues printing it out. If we've finished printing out the last line on the screen, the horizontal cursor position is retrieved as are the various registers (lines 2060 to 2080). Finally, the program does an absolute jump to the monitor's RDKEY subroutine and waits for the next key to be pressed (line 2090).

```
                    1780 *
                    1790 *
                    1800 * A Control-P has been entered so now
                    1810 * it's time to print the screen out.
                    1820 *
0334- 20 4A FF      1830 PRTSCRN JSR SAVE      Save registers.
0337- A5 24         1840         LDA CH        Save cursor's
0339- 48            1850         PHA           horizontal position.
033A- A9 8D         1860         LDA #$8D      Send printer a
033C- 20 02 C2      1870         JSR WARMPRT   carriage return.
033F- A2 00         1880         LDX #$0       Set up for 1st
0341- 8A            1890 GETLINE TXA           screen line.
0342- 20 C1 FB      1900         JSR BASCALC   Calculate video line.
0345- A0 00         1910         LDY #$0       Init char counter.
0347- B1 28         1920 GETCHAR LDA (BASL),Y  Get character
0349- C9 A0         1930 CHKNORM CMP #$A0      Is it normal?
034B- B0 04         1940         BCS PRINTIT   Yes, print it.
034D- 69 40         1950         ADC #$40      No, adjust it.
034F- D0 F8         1960         BNE CHKNORM   Normal now?
0351- 20 02 C2      1970 PRINTIT JSR WARMPRT   Yes, print it.
0354- C8            1980         INY           Increment char counter.
0355- C0 28         1990         CPY #$28      40 characters yet?
0357- D0 EE         2000         BNE GETCHAR   No, get more.
0359- A9 8D         2010         LDA #$8D      Yes, print a
035B- 20 02 C2      2020         JSR WARMPRT   carriage return.
035E- E8            2030         INX           Increment line counter.
035F- E0 18         2040         CPX #$18      24 lines yet?
0361- D0 DE         2050         BNE GETLINE   No, get more.
0363- 68            2060         PLA           Yes, restore
0364- 85 24         2070         STA CH        horiz. cursor.
0366- 20 3F FF      2080         JSR RESTORE   Restore registers.
0369- 4C 0C FD      2090         JMP RDKEY     Get a keypress.
                    2100 *
                    2110 *
                    2120 * This is the printer set-up string.
                    2130 * It consists of a Control-I 40N and is
                    2140 * followed by a carriage return.
                    2150 *
036C- 89            2160 TEXT    .HS 89
036D- B4 B0 CE      2170         .AS -"40N"
0370- 8D 00         2180         .HS 8D00
```

Add a numeric key pad for free

Business and accounting software often require that the user enter large amounts of numerical data. This can be very inconvenient with the conventional Apple keyboard, because the numbers are spread across the top row of the keyboard. To solve this problem, many hardware manufacturers have developed accessory key pads that plug into the Apple and provide the user with a calculator-like layout of numerical keys. These key pads range in price from $80 to $300.

By now, you are probably aware that the Apple computer is a very versatile machine and that most problems one encounters can be solved either in hardware or in software. The vendors of accessory key pads have solved the problem using hardware, at considerable cost. But here, you have a solution to the problem using software, and it's free.

Using the NUMERIC KEY PAD program, you'll be able to treat a section of the standard Apple keyboard as a numeric key pad. The software key pad uses the 7, 8 and 9 of the Apple keyboard as its top row. The three keys underneath these, the U, I and O represent 4, 5 and 6 respectively, while the three keys under these — J, K and L — represent 1, 2 and 3 respectively. In addition to these, a few other keys

have been translated. The M key represents a 0, the semicolon represents a plus sign (+) and the P key represents the multiplication (×) sign. These last two assignments mean that all mathematical operators are now available as single key, unshifted entries. In addition to these keys, the Apple still recognizes the regular number and mathematical operator keys.

Once the program is activated, it can be turned on by entering a Control-K and turned off by entering a Control-Q. This toggling on and off requires the use of a flag byte to determine what mode is currently active. The program starts off in line 1340 by setting the flag byte to a nonzero value to indicate that the key pad is active. After that, control is taken away from the normal input routine and given to the input routine of this program (lines 1360 to 1440).

The input routine for this program starts on line 1520, where a subroutine jump is made to one of the monitor ROM's input routines KEYIN. After this routine gets a character from the keyboard and puts it into the accumulator, it returns to our program where several tests are performed. The first one is a test to see if the character that was entered was a Control-K (line 1530). If it was a Control-K, the contents of the accumulator (which is $8B, the ASCII code for a Control-K) is stored in FLAG (line 1550) to indicate that the key pad is active. Having done this, the program then jumps to the monitor to read the keyboard once more (line 1560). If it was not a Control-K, another test is made to see if the character entered was a Control-Q (line 1570). If it was, the program jumps to a subroutine called TURN-OFF (line 1820), which stores a zero in FLAG, effectively turning off the key pad.

This routine ends by jumping to a routine in the monitor to read the keyboard for the next key pressed (line 1840).

If neither a Control-K or Control-Q are entered, the program proceeds to line 1590 where FLAG is checked to determine whether the key pad is supposed to be active or not. If the value stored in FLAG is zero, the key pad is not active and an RTS instruction is executed. This causes the character that was entered at the keyboard, and which currently is in the accumulator, to pass through the numeric key pad program unaffected. However, if the value of FLAG is not zero, the key pad is active and any character that is entered passes through this program and is checked to see if it is one of the nine characters which have been reassigned.

Each character that is entered when the key pad is active, is checked (line 1620) against the table of values in line 1920. If no match is made with any of the nine characters in TABLE 1, the character is allowed to pass through unchanged (line 1670). However, if a match is made (line 1630) the program branches to a routine called SWITCH at line 1740, which substitutes a character in TABLE 2 for the current character in the accumulator.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***        NUMERIC KEY PAD       ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130          .OR $300
                1140 *
                1150 *
                1160 *
                1170 * EQUATES
                1180 *
0006-           1190 SAVKSWL .EQ $6
0008-           1200 FLAG    .EQ $8
0038-           1210 KSWL    .EQ $38
03D0-           1220 WARMDOS .EQ $3D0
03EA-           1230 CONNECT .EQ $3EA
FD0C-           1240 RDKEY   .EQ $FD0C
FD1B-           1250 KEYIN   .EQ $FD1B
                1260 *
                1270 *
                1280 * This section steals control of the
                1290 * input and passes all characters to
                1300 * be input to the routine beginning
                1310 * with START.  It also sets the keypad
                1320 * flag so the pad will become active.
                1330 *
0300- A9 8B     1340          LDA #$8B        Set keypad active
0302- 85 08     1350          STA FLAG        flag.
0304- A9 17     1360          LDA #START      Get the address of the
0306- A0 03     1370          LDY /START      start of the program.
0308- 85 38     1380          STA KSWL        Store it in the
030A- 84 39     1390          STY KSWL+1      input hooks.
030C- AD D0 03  1400          LDA WARMDOS     Is DOS present?
030F- C9 4C     1410          CMP #$4C
0311- D0 03     1420          BNE NODOS       No.
0313- 20 EA 03  1430          JSR CONNECT     Yes, connect to DOS.
0316  60        1440 NODOS    RTS             Return.
```



```
                1450 *
                1460 *
                1470 * This routine replaces the normal
                1480 * input routine with this program so
                1490 * certain keys on the Apple can be
                1500 * interpreted as a numeric key pad.
                1510 *
0317- 20 1B FD  1520 START    JSR KEYIN       Read the keyboard.
031A- C9 8B     1530          CMP #$8B        Is it a Cntrl-K?
031C- D0 05     1540          BNE NEXT        No, is it Ctrl-Q?
031E- 85 08     1550          STA FLAG        Yes, set FLAG.
0320- 4C 0C FD  1560          JMP RDKEY       Get next key.
0323- C9 91     1570 NEXT     CMP #$91        Is it Ctrl-Q?
0325- F0 15     1580          BEQ TURNOFF     Yes, turn off keypad.
0327- A4 08     1590          LDY FLAG        See is keypad is active.
0329- F0 0C     1600          BEQ RETURN      No, it isn't.
032B- A0 00     1610          LDY #$0         Initialize index.
032D- D9 43 03  1620 LOOP1    CMP TABLE1,Y    Find character in table.
0330- F0 06     1630          BEQ SWITCH      Replace with new character.
0332- C8        1640          INY             Increment index.
0333- C0 09     1650          CPY #$09        End of table?
0335- D0 F6     1660          BNE LOOP1       No, chek more.
0337- 60        1670 RETURN   RTS             Return to caller.
                1680 *
                1690 *
                1700 * The character has been found in
                1710 * TABLE1 and therefore a substitute
                1720 * character from TABLE2 will replace it.
                1730 *
0338- B9 4C 03  1740 SWITCH   LDA TABLE2,Y    Substitute new character.
033B- 60        1750          RTS
                1760 *
                1770 *
                1780 * This routine restores the FLAG
                1790 * byte to zero and turns off the
                1800 * keypad interpreter.
                1810 *
033C- A9 00     1820 TURNOFF LDA #$0          Reset key pad flag.
033E- 85 08     1830          STA FLAG
0340- 4C 0C FD  1840          JMP RDKEY       Get next key.
                1850 *
                1860 *
                1870 * These are the two character tables.
                1880 * TABLE1 contains the keyboard equivalents
                1890 * and TABLE2 what the new key has been
                1900 * defined as.
                1910 *
0343- CF C9 D5
0346- CC CB CA
0349- CD D0 BB  1920 TABLE1   .AS -"OIULKJMP;"
034C- B6 B5 B4
034F- B3 B2 B1
0352- B0 AA AB  1930 TABLE2   .AS -"6543210*+"
```

As you can see, the NUMERIC KEY PAD program is a short one and can easily reside in the unused portion of page three. This program can be combined with BASIC programs to simplify the entering of numerical data for such programs. While the same thing could probably be done in BASIC, the routine would be longer and much slower. In fact, with this program, it should be pretty easy to write a BASIC program that simulates a desktop calculator.

Supplying characters from a different source

Until now, the programs we have looked at that steal control away from the input have only done it so that specific characters could be checked for. There is another

reason to steal control of the input however, and that's to input data from another source altogether. DOS does this when it inputs data from an EXEC (text) file. It reads a text file and then fools the computer into thinking that the data coming from the text file came from the keyboard.

Fooling the computer into thinking that text is coming from the keyboard, when it is really coming from somewhere else, is not difficult. First you have to point the input hooks ($38 and $39) to your substitute program. The first thing that the new input handler should do is to save the X-register so that it can be restored before the new routine is exited. Also, if the new input program is going to allow the user to enter ESCape characters and the right arrow (Control-U) the Y-register must also be saved on entry and restored on exit.

After you save the appropriate registers, the accumulator must be loaded with the character you want input. The source can be anything, disk, cassette, even memory. Once the accumulator contains the desired data, the X-register should be restored if it was modified and an RTS instruction executed. It is the execution of the RTS that actually enters the character as if it came from the keyboard.

In order to eliminate extraneous characters and spaces at the end of the entry of data, a program should be able to determine if the current character that is being loaded into the accumulator is the last character of the text to be entered. To do this, a program should be able to look ahead to see if the next character is the text terminating character, or it should be able to determine if the high bit of the last character is set (assuming of course that the high bit of all the other characters is not set).

It is very important to note here that after all of the text has been entered from the external source, the program must return control of the input to the keyboard. If this is not done, the program may hang up and the keyboard will be inactive.

EXECing without a disk drive

Now that we have the basic knowledge that we need to write our own programs for entering data automatically, let's write a program that simulates DOS' EXEC command. Instead of using a disk to store our text file, however, we're going to use a vacant area of memory. This means that even those people that don't have disk drives, and that number of people is constantly shrinking, can have the advantages of an EXEC capability.

The IN-MEMORY EXEC SIMULATOR program starts out by printing out the title page and waiting for the user to press any key to continue (lines 1340 to 1410). After doing that, the program gets the address of the beginning of the text buffer and stores it in a two-byte, zero page pointer called TXTPTR (lines 1490 to 1520). Then it gets the starting address of the new input routine and stores it in the input hooks (lines 1530 to 1610).

The new input routine starts on line 1720, where the Y-index register is set to zero. In the next line, a character is loaded into the accumulator from the buffer area pointed to by TXTPTR, and in line 1740, this character is saved temporarily in

a location labelled ASAVE. Next, the program does a subroutine jump to INCR (line 1970) where the two-byte text pointer is incremented so that it points to the next character. In line 1760, this next character is loaded into the accumulator and then a test is performed to see if the character is a hexadecimal zero, the end of text marker. If it's not, the previous character, which was temporarily saved in ASAVE, is retrieved (line 1780) and an RTS instruction is executed (line 1790), causing the character to be entered. Since this routine does not modify the X-register, there was no need to save and restore it.

If the end of text marker has been reached, the program branches to line 1870, where the contents of the X-register are temporarily stored in XSAVE. Next, a jump is made to a monitor ROM subroutine called SETKBD. What this routine does is essentially simulate the BASIC command IN#0. By doing this, control of the input is restored to the keyboard. The X-register was stored because SETKBD modifies the X-register when it is called. After returning input control to the keyboard, the program restores the X-register, retrieves the last character to be entered and executes an RTS instruction.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***   IN MEMORY EXEC SIMULATOR   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110          .OR $2C6

                1120 *
                1130 *
                1140 * EQUATES
                1150 *
0006-           1160 TXTPTR   .EQ $6
0008-           1170 XSAVE    .EQ $8
0009-           1180 ASAVE    .EQ $9
0038-           1190 KSWL     .EQ $38
03D0-           1200 WARMDOS  .EQ $3D0
03EA-           1210 CONNECT  .EQ $3EA
FC58-           1220 HOME     .EQ $FC58
FD0C-           1230 RDKEY    .EQ $FD0C
FDED-           1240 COUT     .EQ $FDED
FE89-           1250 SETKBD   .EQ $FE89
                1260 *
                1270 *
                1280 * This section of the program prints
                1290 * out the program title and copyright
                1300 * notice and calls the RDKEY routine
                1310 * where the program ceases to continue
                1320 * until a key is pressed by the user.
                1330 *
02C6- 20 58 FC  1340          JSR HOME        Clear the screen.
02C9- A0 00     1350          LDY #$0         Print out
02CB- B9 15 03  1360 LOOP     LDA TEXT,Y      opening screen
02CE- F0 06     1370          BEQ NEXT1
02D0- 20 ED FD  1380          JSR COUT
02D3- C8        1390          INY
02D4- D0 F5     1400          BNE LOOP
02D6- 20 0C FD  1410 NEXT1    JSR RDKEY       Wait for a keypress.
                1420 *
                1430 *
```

```
                    1440 * Here the address of the buffer area
                    1450 * is placed in a pointer and the normal
                    1460 * keyboard input routine is replaced
                    1470 * by this program.
                    1480 *
02D9- A9 85         1490          LDA #BUFFER     Get buffer
02DB- A0 03         1500          LDY /BUFFER     address and
02DD- 85 06         1510          STA TXTPTR      store in a
02DF- 84 07         1520          STY TXTPTR+1    pointer.
02E1- A9 F4         1530          LDA #START      Get address
02E3- A0 02         1540          LDY /START      for input
02E5- 85 38         1550          STA KSWL        routine and
02E7- 84 39         1560          STY KSWL+1      store in hooks.
02E9- AD D0 03      1570          LDA WARMDOS     Check for DOS.
02EC- C9 4C         1580          CMP #$4C
02EE- D0 03         1590          BNE NODOS       No DOS present.
02F0- 20 EA 03      1600          JSR CONNECT     Connect through it.
02F3- 60            1610 NODOS    RTS
                    1620 *
                    1630 *
                    1640 * This is the replacement input routine
                    1650 * to which the computer comes every time
                    1660 * it would normally look for input from
                    1670 * the keyboard.  Here data are taken
                    1680 * from the buffer area (pointed to by
                    1690 * TXTPTR) and used as if they came from
                    1700 * the keyboard.
                    1710 *
02F4- A0 00         1720 START    LDY #$0         Set offset to zero.
02F6- B1 06         1730          LDA (TXTPTR),Y  Get character.
02F8- 85 09         1740          STA ASAVE       Save it.
02FA- 20 0E 03      1750          JSR INCR        Increment pointer.
02FD- B1 06         1760          LDA (TXTPTR),Y  Get next the character.
02FF- F0 03         1770          BEQ DONE        Last character, finish up.
0301- A5 09         1780          LDA ASAVE       Retrieve character.
0303- 60            1790          RTS             Enter it.
                    1800 *
                    1810 *
                    1820 * When there is no more data in the
                    1830 * EXEC file buffer, an IN#0 is
                    1840 * simulated (JSR SETKBD), and the X-reg
                    1850 * is restored.
                    1860 *
0304- 86 08         1870 DONE     STX XSAVE       Save X-register.
0306- 20 89 FE      1880          JSR SETKBD      Do IN#0.
0309- A6 08         1890          LDX XSAVE       Restore X-registore.
030B- A5 09         1900          LDA ASAVE       Restore accumulator.
030D- 60            1910          RTS
                    1920 *
                    1930 *
                    1940 * This routine increments the two-byte
                    1950 * pointer TXTPTR.
                    1960 *
030E- E6 06         1970 INCR     INC TXTPTR      Increment low byte.
0310- D0 02         1980          BNE NEXT2
0312- E6 07         1990          INC TXTPTR+1    Increment high byte.
0314- 60            2000 NEXT2    RTS             Return.
                    2010 *
                    2020 *
                    2030 * This is the program title and
                    2040 * copyright notice.
                    2050 *
0315- C9 CE AD
0318- CD C5 CD
031B- CF D2 D9
031E- A0 C5 D8
0321- C5 C3 A0
0324- D3 C9 CD
0327- D5 CC C1
032A- D4 CF D2      2060 TEXT     .AS -"IN-MEMORY EXEC SIMULATOR"
032D- 8D            2070          .HS 8D
032E- C2 D9 A0
0331- CA D5 CC
```



```
0334- C5 D3 A0
0337- C8 AE A0
033A- C7 C9 CC
033D- C4 C5 D2      2080          .AS -"BY JULES H. GILDER"
0340- 8D            2090          .HS 8D
0341- C3 CF D0
0344- D9 D2 C9
0347- C7 C8 D4
034A- A0 A8 C3
034D- A9 A0 B1
0350- B9 B8 B2      2100          .AS -"COPYRIGHT (C) 1982"
0353- 8D            2110          .HS 8D
0354- C1 CC CC
0357- A0 D2 C9
035A- C7 C8 D4
035D- D3 A0 D2
0360- C5 D3 C5
0363- D2 D6 C5
0366- C4            2120          .AS -"ALL RIGHTS RESERVED"
0367- 8D 8D 8D
036A- 8D            2130          .HS 8D8D8D8D
036B- D0 D2 C5
036E- D3 D3 A0
0371- C1 CE D9
0374- A0 CB C5
0377- D9 A0 D4
037A- CF A0 C3
037D- CF CE D4
0380- C9 CE D5
0383- C5            2140          .AS -"PRESS ANY KEY TO CONTINUE"
0384- 00            2150          .HS 00
                    2160 *
                    2170 *
                    2180 * This is a sample 'EXEC' file that is
                    2190 * automatically executed when this
                    2200 * this program is run by doing a CALL 710.
                    2210 *
0385- CE C5 D7      2220 BUFFER   .AS -"NEW"
0388- 8D            2230          .HS 8D
0389- B1 B0 C8
038C- CF CD C5      2240          .AS -"10HOME"
038F- 8D            2250          .HS 8D
0390- B2 B0 C6
0393- CF D2 D8
0396- BD B1 D4
0399- CF B2 B5
039C- B5            2260          .AS -"20FORX=1TO255"
039D- 8D            2270          .HS 8D
039E- B3 B0 BF
03A1- C3 C8 D2
03A4- A4 A8 D8
03A7- A9 BB         2280          .AS -"30?CHR$(X);"
03A9- 8D            2290          .HS 8D
03AA- B4 B0 CE
03AD- C5 D8 D4
03B0- D8            2300          .AS -"40NEXTX"
03B1- 8D            2310          .HS 8D
03B2- B5 B0 BF      2320          .AS -"50?"
03B5- 8D            2330          .HS 8D
03B6- D2 D5 CE      2340          .AS -"RUN"
03B9- 8D            2350          .HS 8D
03BA- CC C9 D3
03BD- D4            2360          .AS -"LIST"
03BE- 8D            2370          .HS 8D
03BF- 00            2380          .HS 00
```

As you can see from the listing, the actual replacement input routine is only 32 bytes long ($2F4 to $314). The 46 bytes that precede these are used to print out the title screen and set up the new input program. The bulk of the space is taken up by

text for the screen, which begins on line 2060 and the buffer which begins on line 2220.

This program resides in memory starting at $2C6 and can be activated by typing CALL 710 from the immediate mode in Applesoft. When run in this manner, the program will execute a sample 'EXEC' file that has been included. The file shows how both immediate mode commands and deferred mode program lines can be entered. When 'EXEC'ed, the file will perform the NEW command, enter a short program that prints out the entire ASCII character set, run the program and then list it. By changing the address of the buffer and its contents, you can EXEC anything you want.

Save keystrokes by using Applesoft shorthand

By combining both of the major characteristics associated with new input handling routines: the ability to check for the pressing of specific keys and the the ability to input text from memory or some other source, we can write a program that will significantly reduce the number keys pressed when writing Applesoft programs. Known as APPLESOFT SHORTHAND, this program makes it possible to enter the most frequently used Applesoft commands with a single keystroke. This entry of several letters or words with a single keystroke, is often referred to as a keyboard macro.

The theory of operation of the program is that the normal input routine is replaced with a new one that allows entry of data either from the keyboard or memory, depending on the status of a flag byte. When keyboard entry is allowed, the character entered is tested to see if it is one of twenty-one preselected control characters. If it's not, the control character is entered as it normally would be. But, if it is part of the designated set, then instead of being entered, the control character is used to specify a string of characters that is to be entered instead.

In this program, most of the characters that will be entered in place of the control codes are Applesoft keywords, a table of which resides in the Applesoft interpreter ROMs starting at address $D0D0 and ending at address $D25E. This table of keywords is also known as a token table. The text stored in this table is unusual in that the high bit of the last letter of the word is set. This is used as an end of text delimiter instead of the zero that has been used throughout the programs in this book so far. It's chief advantage is that it saves one byte per text message.

You'll notice that I said MOST of the characters that will be entered in place of the control codes are Applesoft keywords, not all of them. In fact, two of the control characters — Control-@ and Control-E — are used to enter frequently used phrases. Control-@ enters the call to the monitor 'CALL-151', including the carriage return that follows it. Thus, by pressing Control-@, the user is automatically dropped into the monitor mode. The Control-E is used when you want to edit Applesoft program lines because it automatically executes (that means it includes the carriage return) the phrase 'POKE 33,33'. A listing of the addresses, the control code and the keyword or phrase that is printed when the particular control

key is pressed is shown in the table that starts at line 2560 of the accompanying program listing.

The program starts on line 1310 with a jump past the title page text (which sits in page two) to the beginning of the program on line 1520. Lines 1520 to 1580 clear the screen and print out the title page, while lines 1590 to 1660 set the input hooks to the start of the replacement input routine. Lines 1670 to 1690 set the input mode flag to zero so that the program will recognize input from the keyboard, and then passes control to the new input routine by executing an RTS instruction.

The input replacement routine starts on line 1850, where the contents of the X-register and the accumulator are saved for later. The first thing that this program does after storing the registers, is to determine what mode it is in. It does this by loading the input mode flag into the accumulator (line 1870) and checking its value. If the flag is set to zero, the program is in the keyboard mode and data can be accepted from the keyboard. If, on the other hand, the value of flag is anything but zero, the program is in the macro mode and data will be entered from the appropriate table. During this time, the keyboard will be dead.

The routine that handles the keyboard entry of data starts on line 1890, where the former contents of the accumulator are restored. After doing that, the program jumps to the monitor ROM's input routine to read the keyboard (line 1900). Once a character has been entered, it is compared with all of the preassigned control codes (lines 1910 to 1960). If no match is found, the X-register is restored (line 1970) and the character is allowed to pass through this routine unmodified and is entered by executing the RTS in line 1980.

If, however, a match is found, the program branches to line 2060 and the input mode flag is incremented — switching it from the keyboard mode to the macro mode. The contents of the X-register, which was used as an index into the control code table is transferred to the accumulator (line 2070) where it is doubled (line 2080) and then placed back in the X-register. The reason for the doubling is that while the control code table consisted of individual bytes, each entry in the macro address table consists of two bytes. The X-register is now used as an index into the MACRO table of addresses and the low and high bytes of the addresses are retrieved in turn and stored in the page zero pointer TXTPTR (lines 2100 to 2140). Since the X-register was modified by this routine, it is now restored to its former value (line 2150) and control is passed to the routine that handles the input of text from memory. This routine is called MACROIN and starts on line 2210.

In line 2220, a character is retrieved from the address pointed to by TXTPTR. Next, a check is made to see if the high bit is set, a signal that this is the last character of the current macro (line 2230). If the high bit is not set (line 2240), a branch is made to line 2270, where the high bit is set. Line 2250 is reached only if the character currently in the accumulator is the last one to be printed and hence its high bit is set. Here the mode flag is reset to zero so that the program will know that the next character that is to be input will come from the keyboard. Next, the program falls into the routine that sets the high bit (line 2270). Since the high bit of

this character is already set, nothing happens here and the program goes on to increment the two-byte pointer TXTPTR, restore the X-register and enter the character currently in the accumulator (lines 2280 to 2300).

A table called CODES, that contains all of the control codes that have been assigned as shorthand keys, is located on lines 2460 to 2490, while the table containing the addresses of the text to be printed out for each key, begins on line 2560. Lines 2770 to 2800 contain the macros for Control-@ and Control-E. On line 2770, notice that there is no hyphen preceding the first quotation mark as there is in most of the other programs in this book. The absence of the hyphen, as was described earlier in the book, indicates that the text is to be assembled without the high bit set. The presence of the hyphen causes the high bit to be set. The last character in each of the two macros listed here is $8D, which is a carriage return with the high bit set. These bytes serve as terminators for the macros.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***      APPLESOFT SHORTHAND     ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***     ALL RIGHTS RESERVED      ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $28A
                1130 *
                1140 *
                1150 * EQUATES
                1160 *
0006-           1170 TXTPTR   .EQ $6
0008-           1180 XSAVE    .EQ $8
0009-           1190 FLAG     .EQ $9
0018-           1200 ASAVE    .EQ $18
0038-           1210 KSWL     .EQ $38
03D0-           1220 WARMDOS  .EQ $3D0
03EA-           1230 CONNECT  .EQ $3EA
FC58-           1240 HOME     .EQ $FC58
FD0C-           1250 RDKEY    .EQ $FD0C
FD1B-           1260 KEYIN    .EQ $FD1B
FDED-           1270 COUT     .EQ $FDED
                1280 *
                1290 *
                1300 *
028A- 4C EB 02  1310          JMP BEGIN
                1320 *
                1330 *
                1340 * This is the text for the title page.
                1350 *
028D- C1 D0 D0
0290- CC C5 D3
0293- CF C6 D4
0296- A0 D3 C8
0299- CF D2 D4
029C- C8 C1 CE
029F- C4 A0 C9
02A2- CE D4 C5
02A5- D2 D0 D2
02A8- C5 D4 C5
02AB- D2        1360 TEXT     .AS -"APPLESOFT SHORTHAND INTERPRETER"
02AC  8D 8D     1370          .HS 8D8D
```



```
02AE- C2 D9 A0
02B1- CA D5 CC
02B4- C5 D3 A0
02B7- C8 AE A0
02BA- C7 C9 CC
02BD- C4 C5 D2  1380          .AS -"BY JULES H. GILDER"
02C0- 8D        1390          .HS 8D
02C1- C3 CF D0
02C4- D9 D2 C9
02C7- C7 C8 D4
02CA- A0 A8 C3
02CD- A9 A0 B1
02D0- B9 B8 B2  1400          .AS -"COPYRIGHT (C) 1982"
02D3- 8D        1410          .HS 8D
02D4- C1 CC CC
02D7- A0 D2 C9
02DA- C7 C8 D4
02DD- D3 A0 D2
02E0- C5 D3 C5
02E3- D2 D6 C5
02E6- C4        1420          .AS -"ALL RIGHTS RESERVED"
02E7- 8D 8D 8D
02EA- 00        1430          .HS 8D8D8D00
                1440 *
                1450 *
                1460 * This part of the program steals control
                1470 * away from the input and sets the
                1480 * address of START in the input hooks.
                1490 * It also sets the mode flag for input
                1500 * of data from the keyboard.
                1510 *
02EB- 20 58 FC  1520 BEGIN    JSR HOME          Clear the screen.
02EE- A0 00     1530          LDY #$0           Print out the
02F0- B9 8D 02  1540 LOOP     LDA TEXT,Y        opening screen.
02F3- F0 06     1550          BEQ BEGIN2
02F5- 20 ED FD  1560          JSR COUT
02F8- C8        1570          INY
02F9- D0 F5     1580          BNE LOOP
02FB- A9 12     1590 BEGIN2   LDA #START        Get the address of the
02FD- A0 03     1600          LDY /START        start of the program
02FF- 85 38     1610          STA KSWL          and store it in
0301- 84 39     1620          STY KSWL+1        the input hooks.
0303- AD D0 03  1630          LDA WARMDOS       Is DOS present?
0306- C9 4C     1640          CMP #$4C
0308- D0 03     1650          BNE NODOS         No.
030A- 20 EA 03  1660          JSR CONNECT       Yes, connect through DOS.
030D- A9 00     1670 NODOS    LDA #$0           Set input flag to
030F- 85 09     1680          STA FLAG          value for keyboard.
0311- 60        1690          RTS               Return to caller.
                1700 *
                1710 *
                1720 * Here, the X-register is saved for
                1730 * restoring later and a check is made
                1740 * to see whether the keyboard or macro
                1750 * mode is active.  If the macro mode is
                1760 * active a branch to the appropriate
                1770 * routine is made.  Otherwise, a character
                1780 * is input from the keyboard and a check
                1790 * is made to see if it is one of the macro
                1800 * codes.  If not the character passes
                1810 * through as is, otherwise the program
                1820 * branches to a routine to setup the
                1830 * macro input.
                1840 *
0312- 86 08     1850 START    STX XSAVE         Save X-register.
0314- 85 18     1860          STA ASAVE         Save accumulator.
0316- A5 09     1870          LDA FLAG          Check mode flag.
0318- D0 26     1880          BNE MACROIN       Not in keyboard mode.
031A- A5 18     1890          LDA ASAVE         Restore accumulator.
031C- 20 1B FD  1900          JSR KEYIN         Read the keyboard.
031F- A2 00     1910          LDX #$0           Zero offset counter.
0321- DD 5B 03  1920 LOOP1    CMP CODES,X       See if valid control code.
0324- F0 08     1930          BEQ VALID         Valid code entered.
```

```
0326- E8        1940          INX               Increment offset.
0327- E0 15     1950          CPX #$15          All codes checked?
0329- D0 F6     1960          BNE LOOP1         No, do more.
032B- A6 08     1970          LDX XSAVE         Restore X-register.
032D- 60        1980          RTS               Yes, return.
                1990 *
                2000 *
                2010 * Here the index of the control code is
                2020 * converted to an index into the macro
                2030 * address table and the mode flag is set
                2040 * to the macro mode.
                2050 *
032E- E6 09     2060 VALID    INC FLAG          Set macro mode.
0330- 8A        2070          TXA               Transfer X to accumulator.
0331- 0A        2080          ASL               Double it.
0332- AA        2090          TAX               Put it back in X.
0333- BD 70 03  2100          LDA MACRO,X       Get low-byte of macro.
0336- 85 06     2110          STA TXTPTR        Store it in TXTPTR.
0338- E8        2120          INX               Increment pointer.
0339- BD 70 03  2130          LDA MACRO,X       Get hi-byte of macro.
033C- 85 07     2140          STA TXTPTR+1      Store it in TXTPTR+1.
033E- A6 08     2150          LDX XSAVE         Restore X-register.
                2160 *
                2170 *
                2180 * This is the routine that actually prints
                2190 * out the macro.
                2200 *
0340- A0 00     2210 MACROIN  LDY #$0           Set offset to zero.
0342- B1 06     2220          LDA (TXTPTR),Y    Get character
0344- C9 80     2230          CMP #$80          Is high bit set?
0346- 90 04     2240          BCC SETHI         No, set it.
0348- A0 00     2250          LDY #$0           Reset mode flag to
034A- 84 09     2260          STY FLAG          keyboard mode.
034C- 09 80     2270 SETHI    ORA #$80          Set high bit.
034E- 20 54 03  2280          JSR INCR          Increment TXTPTR.
0351- A6 08     2290          LDX XSAVE         Restore X-register.
0353- 60        2300          RTS
                2310 *
                2320 *
                2330 * This routine increments the two-byte
                2340 * pointer used to retrieve the text of
                2350 * the macro.
                2360 *
0354- E6 06     2370 INCR     INC TXTPTR        Increment TXTPTR low byte.
0356- D0 02     2380          BNE RETURN
0358- E6 07     2390          INC TXTPTR+1      Increment TXTPTR high byte.
035A- 60        2400 RETURN   RTS
                2410 *
                2420 *
                2430 * This is a table of control codes that
                2440 * have been set aside for the shorthand codes.
                2450 *
035B- 80 81 82
035E- 83 85 86
0361- 87 89 8A
0364- 8B 8C     2460 CODES    .HS 808182838586878988B8C
                2470 *            @ A B C E F G I J K L
0366- 8E 8F 90
0369- 91 92 94
036C- 96 97 99
036F- 9A        2480          .HS 8E8F9091929496979A
                2490 *             N O P Q R T V W Y Z
                2500 *
                2510 *
                2520 * This is a table of two-byte addresses
                2530 * for each shorthand entry.  Entries are
                2540 * made low-order byte first.
                2550 *
0370- 9A 03     2560 MACRO    .DA MONITOR       @  CALL-151
0372- 4C D2     2570          .HS 4CD2          A  CHR$
0374- 50 D2     2580          .HS 50D2          B  LEFT$
0376- F9 D0     2590          .HS F9D0          C  CALL
0378- A3 03     2600          .DA EDIT          E  POKE 33,33
037A- D3 D0     2610          .HS D3D0          F  FOR
```



```
037C- A4 D1     2620          .HS A4D1          G  GOSUB
037E- DE D0     2630          .HS DED0          I  INPUT
0380- 93 D1     2640          .HS 93D1          J  GOTO
0382- 3B D2     2650          .HS 3BD2          K  PEEK
0384- D4 D1     2660          .HS D4D1          L  LIST
0386- D6 D0     2670          .HS D6D0          N  NEXT
0388- C7 D1     2680          .HS C7D1          O  POKE
038A- CB D1     2690          .HS CBD1          P  PRINT
038C- 4F D1     2700          .HS 4FD1          Q  INVERSE
038E- 97 D1     2710          .HS 97D1          R  RUN
0390- EF D1     2720          .HS EFD1          T  THEN
0392- 64 D1     2730          .HS 64D1          V  VTAB
0394- 49 D1     2740          .HS 49D1          W  NORMAL
0396- A9 D1     2750          .HS A9D1          Y  RETURN
0398- E3 D1     2760          .HS E3D1          Z  TAB(
039A- 43 41 4C
039D- 4C 2D 31
03A0- 35 31     2770 MONITOR  .AS "CALL-151"
03A2- 8D        2780          .HS 8D
03A3- 50 4F 4B
03A6- 45 33 33
03A9- 2C 33 33  2790 EDIT     .AS "POKE33,33"
03AC- 8D        2800          .HS 8D
```

Teach your Apple to recognize lowercase letters

As you probably already know, the original Apple computer that comes fresh out of the box had no lowercase letter capability. For some strange reason, the ability to enter and display lowercase characters was not included in the Apple II Plus computer. Demand for lowercase grew however, and soon, quite a few companies started selling inexpensive adapters that could be easily installed in an Apple and allow it to display lowercase letters. And, when the Apple //e and //c were introduced lowercase capability was finally available.

But displaying lowercase letters is only half the problem, the other half is entering them from the keyboard. Although the Apple II Plus keyboard does have a SHIFT key and it can be used to generate some shifted characters — symbols and punctuation only — it does not allow you to generate the proper ASCII codes for upper and lowercase characters. In addition, even if it did, there is a routine in the Apple monitor ROM called CAPTST, the won't permit the entry of a lowercase character even if some how it were generated.

The routine, which is located at $FD7E, checks to see if the character being input is in the range of $E0 to $FF. If it is, the character is ANDed with the value $DF. This converts the characters to the $C0 to $DF range, which represents the upper case letters. By simply changing a single byte at $FD83 from $DF to $FF which was done in the //e and //c, it would be possible to permit the entry of lowercase characters, assuming of course that they could be generated by the keyboard. Since most Apple II Plus owners use a ROM version of Applesoft, it is not too convenient to change the one byte required, and it doesn't solve the rest of the problem anyway.

Another approach to the problem is to write a special input routine that will allow you to generate the lowercase codes directly from the keyboard and enter them. To do this, it will be necessary to, once more, steal control away from the normal Apple input routines and direct it to a new input program. While this

program, LOWER CASE INPUT DRIVER, will let you enter lowercase characters, they will not be displayed unless you have a lowercase adapter. This will vary from a single chip to a circuit board that plugs into the Apple and prices range from $20 to $80. Some of the adapters come with software that will let you enter lowercase letters. And some of the software is not very good. Some of it will simply consist of a few lines of BASIC program code that do a crude job in handling lowercase letters.

You will find this program, however, to be very handy, very reliable, and very user friendly (that's a term you'll be hearing more and more often). The LOWER CASE INPUT DRIVER is very versatile and will work either with or without a hardware modification that makes the SHIFT key on an Apple II Plus active. Most other lowercase programs work only with the modification or only without it. What this modification does, is connect the SHIFT key to pin 4 on the game I/O connector, where it can be checked by software to see if the key is pressed or not. Instructions on how to implement this modification are in Appendix C.

If you have not made the modification to the SHIFT key when you installed your lowercase adapter, you can use the ESCape key for a single character shift or use Control-A to toggle back and forth between upper and lowercase lock modes. In the upper case lock mode, the Apple keyboard behaves as it normally does. In the lowercase lock mode you get both lowercase letters and additional symbols not normally available.

The program has one additional feature that makes it very user friendly. Whenever the ESCape key is pressed to produce a capital letter, or the Control-A is pressed to enter either the upper or lowercase modes, the cursor that marks the place where the next character will appear turns into a flashing 'U' or 'L' depending on whether the next character that will be entered is upper or lowercase. With this feature, you will always be aware of when the case of the character being input changes.

The program starts out as most programs of this type, by replacing the address in the input hooks with the address of the new input routine (lines 1400 to 1440). Then, the upper case flag is reset so that the program comes up running in the caps lock mode, just as the Apple normally does (lines 1450 to 1480). The new input routine starts on line 1580, where a subroutine jump to the monitor's KEYIN routine is performed. This reads the keyboard and waits for a key to be pressed. Once a key has been pressed, the program saves the character that was entered (which is now in the accumulator) and the X-register (lines 1590 and 1600) and then checks the return address that is on the stack to see if it is $FD77. It does this by first retrieving the stack pointer from the location where DOS temporarily stored it, and loading that value into the X-register (line 1610).

Once that is done, we can access the return address that is on the stack and see if it is $FD77 (lines 1620 to 1650). This is actually one less than the real address (which is $FD78) because as the address is pulled off the stack later on, the 6502 increments it by one. If the address is not $FD77, the program branches to line 1840 where the accumulator and the X-register are restored and processing continues.

But, if the address on the stack is $FD77, it is changed to ADDCHR-1, which is $390 (lines 1670 to 1700). ADDCHR is the routine that we use to replace the ROM code which converts all incoming characters to uppercase. It is similar to the code in the F8 ROM between $FD78 and $FD83, except that it eliminates the CAPTST routine, which does the uppercase conversion.

After the address on the stack has been changed, the accumulator and the X-register are restored (lines 1840 and 1850) and the program checks to see if the key that was pressed earlier, was a Control-A (line 1860). If it was, the program looks at the input mode flag (line 1880) to find out what mode (upper or lowercase) is currently active. If the flag is zero, the program is currently in the upper case mode. And, since a Control-A was pressed, the user has told the computer that the mode should be changed. The branch at line 1890 causes the program to go to line 1940 where the flag is incremented by one putting the program in the lowercase mode. In line 1950, the ASCII code for a flashing 'L' is placed in the accumulator, and in line 1960, the flashing 'L' is placed in the screen in the position where the next entered character will appear. Thus, the user is alerted to the fact that the next character that will be entered will be a lowercase character. The program then branches back to line 1580 to get the next character.

If the mode flag indicates the lowercase mode is currently active, the flag is decremented by one (line 1900) causing the flag to be reset to zero. This tells the program that the upper case mode should be active. After resetting the mode flag, a flashing 'U' is loaded into the accumulator (line 1910) and then stored on the screen in the position where the next entered character would appear (line 1920). The program then branches back to line 1580 to get the next character.

If the character that was entered in line 1580 is not a Control-A, control is passed to line 2090 where the character is temporarily saved on the stack and the mode flag is examined (line 2100) to determine what mode the program is in. If the flag indicates that the caps lock mode is active (line 2110), the program ceases doing any processing on the character that was input, retrieves it from the stack (line 2620) and inputs it as it was entered from the keyboard by executing an RTS instruction (line 2630).

On the other hand, if the program is in the lowercase lock mode, the character that was entered is retrieved from the stack (line 2120) and several tests and appropriate modifications are performed. In line 2130, the character is tested to see if it is the ESCape key. If it is, the case flag is set by incrementing it by one (line 2150), a flashing 'U' is placed on the screen (lines 2160 and 2170) and the next character is input (line 2180).

Taking advantage of the SHIFT key modification

Until now, the program has been dealing with ways of letting the user enter upper and lowercase letters without making the hardware modification. However, most people who know how to type, are used to pressing the SHIFT key to get an upper case letter, so the next routine (CHKSHFT), which starts on line 2300, will

check for the modified SHIFT key and make letters entered while it is pressed upper case.

The first thing that the CHKSHFT routine does is to temporarily store the last character entered on the stack. It then checks the pushbutton port that has been assigned to game paddle 2 (this does not affect the normal usage of paddles 0 and 1) to which the SHIFT key has now been connected (line 2310). If the value retrieved from this pushbutton port ($C063) is between $0 and $7F, the SHIFT key is being pressed and the program branches to the capitalization routine (line 2480). If it is $80 or greater, the SHIFT key is not being pressed.

Now that we know the program is not in the caps lock mode and the SHIFT key is not being pressed, there's only one more thing we have to do, and that is to check if the ESCape key was pressed just prior to entering this character. If it was, we know this character is supposed to be upper case, just as if the SHIFT key were being pressed (lines 2340 and 2350).

Upon determining that this character is not supposed to be shifted, it is retrieved from the stack (line 2360) and checked to see if it is a number or letter (line 2370). If it is a number (line 2380) it is printed out as is. And if it is a letter, it is made

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    LOWER CASE INPUT DRIVER    ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY     ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130         .OR $300
                1140 *
                1150 *
                1160 *
                1170 * EQUATES
                1180 *
0006-           1190 FLAG    .EQ $6
0007-           1200 MODE    .EQ $7
0008-           1210 ASAVE   .EQ $8
0009-           1220 XSAVE   .EQ $9
0024-           1230 CH      .EQ $24
0028-           1240 BASL    .EQ $28
0038-           1250 KSWL    .EQ $38
0100-           1260 STACK   .EQ $100
03EA-           1270 CONNECT .EQ $3EA
AA59-           1280 SSAVDOS .EQ $AA59
C063-           1290 SHIFT   .EQ $C063
FD1B-           1300 KEYIN   .EQ $FD1B
FD75-           1310 NXTCHR  .EQ $FD75
FD84-           1320 ADDINP  .EQ $FD84
                1330 *
                1340 *
                1350 * This section steals control of the
                1360 * input and passes all characters to
                1370 * be input to the routine beginning
                1380 * with START.
                1390 *
0300- A9 12     1400         LDA #START     Get the address of the
0302- A0 03     1410         LDY /START     start of the program.
```


lowercase by ORing the value in the accumulator with $20 (line 2390). The program then branches to the end of the program where the letter is input. The branch on line 2400 is always taken because the value in the accumulator never equals zero at this stage of the program.

In the normal Apple, the shifted M, N and P letters produce the], ^ and @ characters. The next routine, called CAP, corrects this so that they produce the normal capitalized letters instead. To get those three symbols, it is necessary to first be in the upper case lock mode and then press the SHIFT key and the M, N or P keys. In line 2480, the character is retrieved from the stack and in line 2490, it is tested to see if it is a letter. If it's not a letter, the program branches to line 2590 and prints it out. If it is a letter, it is checked to see if it is a P, M or N (lines 2510 to 2580) and if it is, the accumulator is loaded with the ASCII code for the appropriate capital letter.

The final part of this routine, labelled DONE and on line 2590, temporarily stores the character on the stack, resets the caps mode flag to lowercase (lines 2600 and 2610), retrieves the character from the stack (line 2620) and finally enters the character by executing an RTS instruction (2630).

```
0304- 85 38     1420         STA KSWL       Store it in the
0306- 84 39     1430         STY KSWL+1     input hooks.
0308- 20 EA 03  1440         JSR CONNECT    Connect to DOS.
030B- A9 00     1450         LDA #$0        Reset upper case flag.
030D- 85 06     1460         STA FLAG
030F- 85 07     1470         STA MODE
0311- 60        1480         RTS            Return.
                1490 *
                1500 *
                1510 * This routine replaces the normal
                1520 * input routine.  A keypress is gotten
                1530 * and the accumulator and X-register are
                1540 * saved while the return address on the
                1550 * stack is changed so the CAPTST routine
                1560 * in the monitor ($FD7E) is bypassed.
                1570 *
0312- 20 1B FD  1580 START   JSR KEYIN      Read the keyboard.
0315- 85 08     1590         STA ASAVE      Save the accumulator.
0317- 86 09     1600         STX XSAVE      Save the X-register.
0319- AE 59 AA  1610         LDX SSAVDOS    Get stack pointer.
031C- BD 03 01  1620         LDA STACK+3,X  See if the
031F- C9 77     1630         CMP #NXTCHR+2  return address on the
0321- BD 04 01  1640         LDA STACK+4,X  stack is $FD78-1.
0324- E9 FD     1650         SBC /NXTCHR+2
0326- 90 0A     1660         BCC NOTINP     It's not restore registers.
0328- A9 90     1670         LDA #ADDCHR-1  It is, replace it with
032A- 9D 03 01  1680         STA STACK+3,X  the address of the
032D- A9 03     1690         LDA /ADDCHR-1  ADDCHR routine to
032F- 9D 04 01  1700         STA STACK+4,X  bypass CAPTST.
                1710 *
                1720 *
                1730 * Here the accumulator and the X-register
                1740 * are restored and the program checks to see
                1750 * if a Control-A, which is used as a shift
                1760 * lock key, is pressed.  If not, the
                1770 * processing of the character continues.
                1780 * If a Control-A was entered, the program
                1790 * determines what mode it is currently
                1800 * in, switches to the other and sets the
                1810 * prompt to a flashing 'L' or 'U',
                1820 * depending on what the new mode is.
                1830 *
```

```
0332- A5 08    1840 NOTINP  LDA ASAVE      Restore accumulator.
0334- A6 09    1850         LDX XSAVE      Restore X-register.
0336- C9 81    1860         CMP #$81       Is it a Ctrl-A?
0338- D0 14    1870         BNE CONTIN     No, continue processing.
033A- A5 07    1880         LDA MODE       Yes, check current mode.
033C- F0 08    1890         BEQ SETMOD     Now upper case, make lower.
033E- C6 07    1900         DEC MODE       Now lower case, make upper.
0340- A9 55    1910         LDA #$55       Show upper case prompt.
0342- 91 28    1920         STA (BASL),Y
0344- D0 CC    1930         BNE START      Branch always, get new key.
0346- E6 07    1940 SETMOD  INC MODE       Set lower case mode.
0348- A9 4C    1950         LDA #$4C       Show lower case prompt.
034A- 91 28    1960         STA (BASL),Y
034C- D0 C4    1970         BNE START      Branch always, get new key.
               1980 *
               1990 *
               2000 * Here a check is made for caps lock mode.
               2010 * If in this mode, the keyboard acts as
               2020 * a normal Apple keyboard.  If not in
               2030 * caps lock, a check is made to see if
               2040 * the ESC key (used as a shift key) was
               2050 * pressed.  If so, cap flag is set and
               2060 * upper case prompt is set.  Otherwise
               2070 * the SHIFT key is checked.
               2080 *
034E- 48       2090 CONTIN  PHA            Save character.
034F- A5 07    2100         LDA MODE       Check input mode.
0351- F0 3C    2110         BEQ END        Caps lock, input as is.
0353- 68       2120         PLA            Lower case, restore character.
0354- C9 9B    2130         CMP #$9B       Is it the ESC key?
0356- D0 08    2140         BNE CHKSHFT    No, check shift key.
0358- E6 06    2150         INC FLAG       Yes, set caps flag.
035A- A9 55    2160         LDA #$55       Set upper case prompt.
035C- 91 28    2170         STA (BASL),Y
035E- D0 B2    2180         BNE START      Branch always, get new character.
               2190 *
               2200 *
               2210 * This section checks to see if a shift
               2220 * key, that has been modified by connecting
               2230 * it to pin 4 of the Game I/O connector,
               2240 * has been pressed.  If so, capitalize
               2250 * letter entered, otherwise see if ESC
               2260 * was used to shift case.  If so input
               2270 * cap, otherwise see if character is a
               2280 * letter and make lower case if it is.
               2290 *
0360- 48       2300 CHKSHFT PHA            Save character.
0361- AD 63 C0 2310         LDA SHIFT      Check shift key.
0364- C9 80    2320         CMP #$80       Is it down?
0366- 90 0D    2330         BCC CAP        Yes, handle it.
0368- A5 06    2340         LDA FLAG       No, was ESC used for shift?
036A- D0 09    2350         BNE CAP        Yes, handle it.
036C- 68       2360         PLA            No, restore character.
036D- C9 C0    2370         CMP #$C0       Is it a letter?
036F- 90 19    2380         BCC DONE       No, input as is.
0371- 09 20    2390         ORA #$20       Yes, make it lower case.
0373- D0 15    2400         BNE DONE       Input character.
               2410 *
               2420 *
               2430 * This is where Shift M,N and P are
               2440 * corrected to their real values.  Also
               2450 * the caps flag is reset so lower case
               2460 * will be entered.
               2470 *
0375- 68       2480 CAP     PLA
0376- C9 C0    2490         CMP #$C0       Is it a letter?
0378- 90 10    2500         BCC DONE       No, input as is.
037A- F0 0C    2510         BEQ CPTLP      Make it a capital-P.
037C- C9 DD    2520         CMP #$DD       Is it a shift-M?
037E- F0 04    2530         BEQ CPTLMN     Yes, make it a capital-M.
0380- C9 DE    2540         CMP #$DE       Is it a shift-N?
0382- D0 06    2550         BNE DONE       No, input as is.
0384- 29 EF    2560 CPTLMN  AND #$EF       Capitalize M and N.
```



```
0386- D0 02    2570         BNE DONE       Input it.
0388- A9 D0    2580 CPTLP   LDA #$D0       Get a capital-P.
038A- 48       2590 DONE    PHA            Save character.
038B- A9 00    2600         LDA #$0        Reset the caps
038D- 85 06    2610         STA FLAG       mode flag.
038F- 68       2620 END     PLA            Retrieve character.
0390- 60       2630         RTS            Input it.
               2640 *
               2650 *
               2660 * This routine substitutes for the code
               2670 * between $FD78 and $FD83 in the F8 ROM
               2680 * and permits the entry of lowercase characters.
               2690 *
0391- C9 95    2700 ADDCHR  CMP #$95       Is character a Ctrl-U?
0393- D0 04    2710         BNE GOADD      No, put it in input buffer.
0395- A4 24    2720         LDY CH         Yes, get the previous
0397- B1 28    2730         LDA (BASL),Y   character and add it
0399- 4C 84 FD 2740 GOADD   JMP ADDINP     to the input buffer.
```

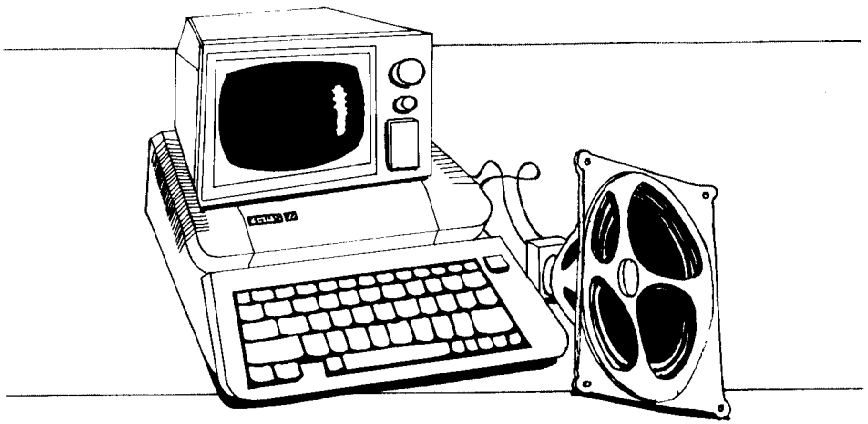
Chapter 6

USING SOUND IN YOUR PROGRAMS

One of the really nice things about the Apple computer is that it has a built-in speaker that can be controlled by software. While the speaker is small, and the quality of sound it produces can be less than high fidelity, nevertheless, it can be used for a wide variety of applications from generating warning signals by an application program to generating music. It can even be used to generate some fairly realistic sound effects for action games.

While the use of the internal speaker is limited only by your own imagination, a small sampling of useful routines and applications will be presented here. Some of these programs can be used by themselves, such as the KEYBOARD CLICKER, MORSE CODE GENERATOR and CASSETTE DUPLICATOR, while others can be used with BASIC or machine language programs to produce desired sound effects. On the sound effects programs, feel free to vary the parameters and see what effect the change has on the sound. You might just come up with that elusive sound you've been looking for.

A major portion of the sound programs in this chapter are made available through the kind permission of Bob Sander-Cederlof, who puts out a monthly



publication called Apple Assembly Line. Those programs, and some of the explanatory text, come from the February 1981 issue of Apple Assembly Line.

The speaker hardware in the Apple is very simple. A flip-flop, which is a device that repeatedly alternates between two states (e.g. ON and OFF), controls the current that is supplied to the coil of the speaker. The flip-flop is connected in such a way, that it reverses the flow of current through the speaker's coil. This is important, because the direction of the current flow determines whether the cone of the speaker is pulled in or out. If we "toggle" the flip-flop and cause it to continuously reverse the flow of current through the speaker, we can cause the speaker to produce audible sound. The rate at which we toggle, determines the frequency of the sound. And, by changing the toggling of the speaker dynamically, it is possible to produce some very complex sounds. The toggling of the speaker is accomplished by accessing location $C030 with any of the load, store or BIT instructions.

How to generate a simple tone

To generate a simple tone, it is only necessary to toggle the speaker at a rate that is low enough so that it falls within the range of 20 to 20,000 Hertz (cycles per second), which is the range of signals that the human ear can detect. The program SIMPLE TONE ROUTINE generates a tone burst of 128 cycles (this is equal to 256 half-cycles). Each half cycle here consists of 1288 Apple clock pulses. Since the internal Apple clock frequency is about 1 MHz, the frequency of sound that is produced is about 338 Hz.

The program starts out by setting the Y-register, which is used as a half-cycle counter, to zero (line 1210). In line 1220, the X-register, which is used as a delay counter, is also set to zero. The sound producing section of code starts with LOOP1 on line 1230, where the speaker is toggled. After toggling the speaker, the program waits, while LOOP2 is executed (lines 1240 and 1250). LOOP2 is only used to produce a time delay, which will be equal to the amount of time it takes to decrement the X-register to zero. After the delay, the Y-register is decremented (line 1260) and the speaker is toggled once more (line 1270). This continues until 256 half-cycles (128 cycles) are completed.

Figuring out the frequency

At this point, you might be curious how the frequency is figured out. This is done by determining the time taken up by each half cycle, doubling it and then taking its inverse. Let's go through a sample calculation.

To start with, we have to add up all of the 6502 cycles for each instruction in the sound producing loop (lines 1230 to 1270). To avoid confusion between the 6502 cycles and the cycles of the sound producing loop, each half cycle will be referred to as a pulse. Thus, there are two pulses per sound producing cycle (Hz). The LDA instruction in line 1230 takes 4 cycles. The DEX instruction on the next line, takes

```
                1000 ************************************
                1010 ***                              ***
                1020 ***      SIMPLE TONE ROUTINE     ***
                1030 ***                              ***
                1040 ***       REPRINTED FROM THE     ***
                1050 ***     FEBRUARY  1981 ISSUE OF  ***
                1060 ***       APPLE ASSEMBLY LINE    ***
                1070 ***      COPYRIGHT (C) 1981 BY   ***
                1080 ***           S-C SOFTWARE       ***
                1090 ***       ALL RIGHTS RESERVED    ***
                1100 ***                              ***
                1110 ************************************
                1120 *
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
C030-           1180 SPEAKER .EQ $C030
                1190 *
                1200 *
0800- A0 00     1210         LDY #$0       Cycle counter
0802- A2 00     1220         LDX #$0       Delay counter
0804- AD 30 C0  1230 LOOP1   LDA SPEAKER   Toggle speaker
0807- CA        1240 LOOP2   DEX           Decrement delay counter.
0808- D0 FD     1250         BNE LOOP2
080A- 88        1260         DEY           Do 128 cycles.
080B- D0 F7     1270         BNE LOOP1
080D- 60        1280         RTS
```

2 cycles and the BNE instruction in line 1250 takes 3 cycles when it branches and only 2 cycles when it does not branch. The DEX-BNE pair in lines 1240 and 1250 are executed 256 times for each pulse. The last time through this loop the BNE instruction does not branch, and thus only 2 cycles are used. The DEY-BNE pair will branch once for each pulse, so 5 cycles are used here. Now let's total up the number of cycles used:

Operation	Cycles
Toggle speaker	4
Delay loop (5 × 255) = ...	1275
End of delay loop	4
DEY-BNE Pair	5
Total Number of Cycles	1288

Since the Apple's internal clock works at roughly 1 MHz, each cycle is equal to 1 microsecond. So for each pulse, or each half of a sound generating cycle, 1288 microseconds are required. Doubling this, to get 2576 microseconds, gives us the time required for each sound cycle and using the formula:

$$\text{Frequency} = 1/\text{Time}$$

This formula assumes time is measured in seconds. If it is measured in microseconds, as it is here, the formula becomes:

$$\text{Frequency} = \frac{1{,}000{,}000}{\text{Time}}$$

Thus, we can calculate the frequency as being equal to:

$$\text{Frequency} = \frac{1{,}000{,}000}{2576} = 388\,\text{Hz}$$

As you can see, by increasing or decreasing the amount of delay within the sound-producing loop, it is possible to change the frequency of the sound that is generated.

Examining the Apple BELL routine

The preceding program is good for producing a sound of 388 Hz. But if you wanted to change the frequency, you'd have to change the program to increase or decrease the delay. It would be much more convenient to have a program that can be entered with a variable related to the frequency so that the same routine could be used to generate a whole range of frequencies. This is what Apple Computer did with the BELL routine inside the F8 ROM, at location $FBE2.

Unlike the the previous program, the APPLE BELL ROUTINE uses another monitor routine WAIT ($FCA8) to produce the delay that determines the width of the generated pulse. Thus, if the Y-register and the accumulator are loaded with data and this program is entered at line 1210, the user has full control over the frequency and duration of the pulse.

The APPLE BELL ROUTINE starts on line 1190 where the Y-register is preset for 192 ($C0) sound cycles. This is simply used to determine how long the sound will be played. On line 1200, the accumulator is loaded with a value that is used by the WAIT routine to generate a time delay. The delay can be determined by the following formula:

```
                1000 ************************************
                1010 ***                              ***
                1020 ***      APPLE BELL ROUTINE      ***
                1030 ***                              ***
                1040 ***       REPRINTED FROM THE     ***
                1050 ***     FEBRUARY  1981 ISSUE OF  ***
                1060 ***       APPLE ASSEMBLY LINE    ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130 * EQUATES
                1140 *
C030-           1150 SPEAKER .EQ $C030
FCA8-           1160 WAIT    .EQ $FCA8
                1170 *
                1180 *
0800- A0 C0     1190         LDY #$C0      Number of half-cycles
0802- A9 0C     1200 BELL2   LDA #$0C      Set delay to 500 microseconds,
0804- 20 A8 FC  1210         JSR WAIT      the half cycle of 1000 Hz.
0807- AD 30 C0  1220         LDA SPEAKER   Toggle the speaker.
080A- 88        1230         DEY           Count the half cycle.
080B- D0 F5     1240         BNE BELL2     Not finished.
080D- 60        1250         RTS           Finished return to caller.
```

$$\text{Delay} = (13 + 13.5A + 2.5A^2) \times 1.023 \text{ microseconds}$$

where A is the number that is in the accumulator. In the BELL routine, the accumulator is loaded with 12 ($0C) to produce a time delay of about 500 microseconds per half cycle. This means the frequency of the sound generated would be about 1000 Hz. After the delay in the WAIT subroutine (line 1210), the program comes back to toggle the speaker (line 1220). Next, the number of half cycles that are left to be played is decreased by one (line 1230) and a check is made to see if all of the half cycles have been played (line 1240). If not, the program goes back to line 1200 to play another half cycle.

Let your keyboard tell you what's happening

For those of you who are light-of-touch, and aren't always sure that the key you pressed on the keyboard went down far enough to register, the next program is for you. Called the KEYBOARD CLICKER, this program provides you with audio feedback that tells you when a key has been pressed. The program uses the techniques we learned in the last chapter to steal control away from the normal input routines and channel it to a new input routine (lines 1270 to 1350). The new program checks to see if a key has been pressed, and if so, generates a short click through the Apple's internal speaker.

The new input routine, which starts at line 1430, temporarily stores the accumulator and the Y-register on the stack (lines 1430 to 1450), so that they can be restored to their original values after the speaker has clicked. With the contents of the Y-register safely stored, a new value of 10 ($A) is placed in the register (line 1460) and a subroutine jump is made to the BELL2 entry point of the Apple's BELL routine (line 1470). What this does, is to generate a frequency of 1000 Hz (we haven't changed the amount of time spent in the WAIT loop) that consists only of 5 cycles (10 half cycles). The result is a nicely audible click. If the sound is too pronounced, you can reduce it by loading the Y-register in line 1460 with a 2. If you want it more pronounced, you can load in larger numbers up to 255 ($FF).

After the click has been generated, the Y-register and accumulator are restored (lines 1480 to 1500) and the next key press is gotten.

RAT-A-TAT-TAT here's the Apple machine gun

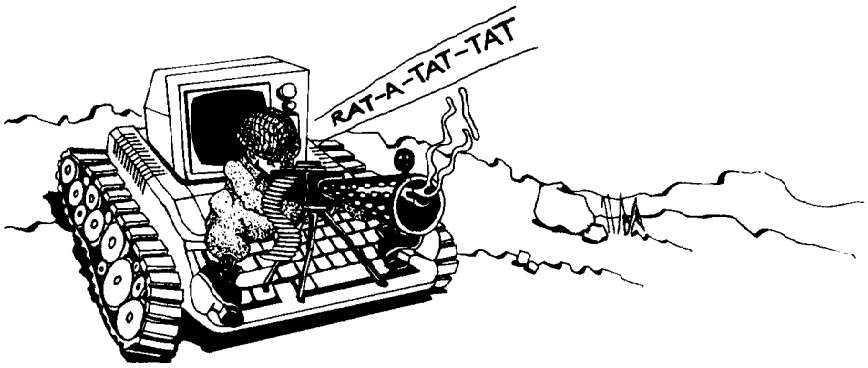
The next four programs are going to show you how it's possible to produce sound effects on the Apple's internal speaker. The first effect will be that of a machine gun. The sound of a machine gun is not composed of tones, but instead is made up of noise, or random sounds. If we were to generate pulses with random widths, we'd generate noise that could sound just like machine-gun fire. That's what is done in the program MACHINE GUN NOISE.

The program starts out by setting up the X-register to determine how many pulses will be in the noise burst, or how long the burst of will last (line 1230). In lines 1240 and 1250, an additional one byte counter is set up to determine the


```
                1000 ************************************
                1010 ***                              ***
                1020 ***       KEYBOARD CLICKER       ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***       JULES H. GILDER        ***
                1060 ***     ALL RIGHTS RESERVED      ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130 * EQUATES
                1140 *
0038-           1150 KSWL     .EQ $38
03D0-           1160 WARMDOS  .EQ $3D0
03EA-           1170 CONNECT  .EQ $3EA
FBE4-           1180 BELL2    .EQ $FBE4
FD1B-           1190 KEYIN    .EQ $FD1B
                1200 *
                1210 *
                1220 * This section steals control of the
                1230 * input and passes all characters to
                1240 * be output to the routine beginning
                1250 * with START.
                1260 *
0800- A9 13     1270          LDA #START    Get the address of
0802- A0 08     1280          LDY /START    the start of the program.
0804- 85 38     1290          STA KSWL      Store it in the
0806- 84 39     1300          STY KSWL+1    input hooks.
0808- AD D0 03  1310          LDA WARMDOS   Is DOS present?
080B- C9 4C     1320          CMP #$4C
080D- D0 03     1330          BNE NODOS     No.
080F- 20 EA 03  1340          JSR CONNECT   Yes, connect to DOS.
0812- 60        1350 NODOS    RTS           Return.
                1360 *
                1370 *
                1380 * This routine replaces the normal
                1390 * input routine and causes a click to
                1400 * be generated each time a key on the
                1410 * keyboard is pressed.
                1420 *
0813- 48        1430 START    PHA           Save entered character.
0814- 98        1440          TYA           Save the Y-register.
0815- 48        1450          PHA
0816- A0 0A     1460          LDY #$A       Setup short a
0818- 20 E4 FB  1470          JSR BELL2     bell (click).
081B- 68        1480          PLA           Restore the Y-register.
081C- A8        1490          TAY
081D- 68        1500          PLA           Restore the character.
081E- 4C 1B FD  1510          JMP KEYIN     Get the next key press.
```

number of bursts that will be heard. Next, the speaker is toggled at line 1260. The width of each pulse produced by the speaker (each half cycle) is determined by line 1270. Here, the Y-register is loaded with a pseudo-random number that is used to determine the pulse width, which is caused by the delay generated in LOOP2 (lines 1280 and 1290). The particular area chosen was the first page of the F8 ROM. Since this is ROM, the data will stay the same from computer to computer and the noise is repeatable. But experiment a little. Use different addresses in line 1270 and listen to how the sound changes.

After the delay caused by LOOP2, the X-register, or pulse counter, is decremented and the next pulse is generated (lines 1300 and 1310). When all of the pulses of the burst have been generated, the program reduces the burst counter by one



Use your Apple as a machine gun.

(line 1320) and starts generating the next burst of pulses (line 1330). This goes on until 10 bursts have been generated.

Use swooping lasers for space games

Moving up from conventional handheld weaponry to the weapons of the future, the next sound we are going to learn how to create is the swoop of a laser gun. Laser swoops, or blasts, are a common feature in space games, and the addition of this sound effect makes those games appear that much more realistic.

To produce a laser blast sound, it is necessary for us to change the width of the pulse being generated from a wide one to a narrow one. This will produce a low tone that gradually slides higher and higher until it is beyond the range of the human ear (or the Apple's speaker).

The program starts out by setting up some parameters. In lines 1240 and 1250 the program sets up the routine for producing only one pulse at each width. Next, in line 1260, a maximum width is assigned to the pulse. The sound generation routine starts at line 1280 where the Y-register is loaded with the pulse count, in this case one. The speaker is toggled in line 1290 and the delay required to produce the desired pulse width is set up and executed in lines 1300 to 1320. In line 1330, the pulse counter (Y-register) is decremented until the count becomes zero. Once this occurs, the width of the pulse is reduced (lines 1350 and 1360) until the width becomes zero, at which point the program returns to its calling routine or mode (1370).

If you run the program at $800 you will hear one swoop which is not too impressive. However, if you use the additional Multi-Swooper routine, you will hear some very nice laser blasts. This simple routine, which starts at line 1420, merely calls the SWOOP program 10 times.

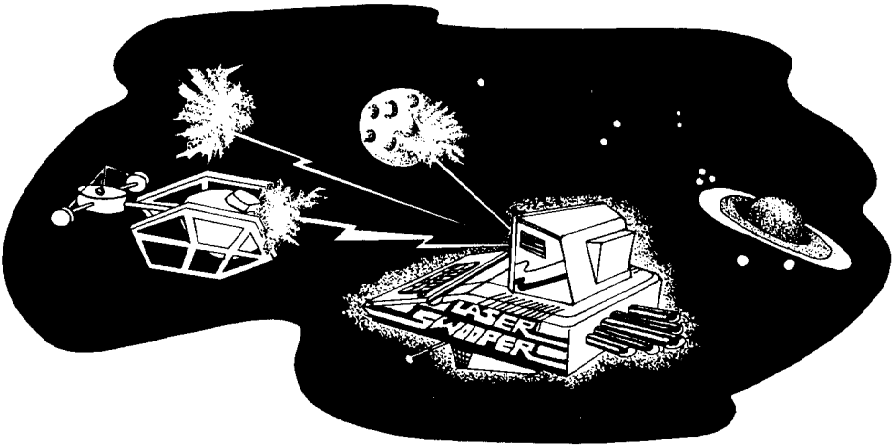

```
                1000 ************************************
                1010 ***                              ***
                1020 ***       MACHINE GUN NOISE      ***
                1030 ***                              ***
                1040 ***     REPRINTED FROM THE       ***
                1050 ***   FEBRUARY  1981 ISSUE OF    ***
                1060 ***     APPLE ASSEMBLY LINE      ***
                1070 ***    COPYRIGHT (C) 1981 BY     ***
                1080 ***         S-C SOFTWARE         ***
                1090 ***     ALL RIGHTS RESERVED      ***
                1100 ***                              ***
                1110 ************************************
                1120 *
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
0000-           1180 COUNTER .EQ $0
BA00-           1190 RANDOM  .EQ $F800
C030-           1200 SPEAKER .EQ $C030
                1210 *
                1220 *
0800- A2 40     1230         LDX #$40      Length of noise burst.
0802- A9 0A     1240         LDA #$0A      Number of noise bursts.
0804- 85 00     1250         STA COUNTER
0806- AD 30 C0  1260 LOOP1   LDA SPEAKER   Toggle speaker.
0809- BC 00 F8  1270         LDY RANDOM,X  Get pulse width pseudo-randomly.
080C- 88        1280 LOOP2   DEY           Delay loop for pulse width.
080D- D0 FD     1290         BNE LOOP2
080F- CA        1300         DEX           Get next pulse in burst.
0810- D0 F4     1310         BNE LOOP1
0812- C6 00     1320         DEC COUNTER   Get next noise burst.
0814- D0 F0     1330         BNE LOOP1
0816- 60        1340         RTS
```

If you're going to use a routine such as this one in a program, you're probably concerned about making the visual effects of the laser blast appear simultaneously with the sound. There is no possibility of doing two things at once on the Apple, but if you do things fast enough, they can be done one right after the other, and still appear to be simultaneous. That's the case here. Because this is a machine language routine and fairly fast, it is possible to generate the sound first and then the graphics and have it appear to occur simultaneously.

This program is a fairly versatile one and I encourage you to experiment a little. You can start out by changing the values in line 1260 from 160 to 128, 80 and 40 to see what effect these have on the sound produced. Next, try changing the number of pulses generated at each width (line 1240). Here you might want to try numbers such as 2, 5, 20 and 40. But don't limit yourself to these, experiment. Another thing you might want to try is running the pulse width in the opposite direction, from a narrow pulse to a wide one. You can do this by changing line 1350 to INC PULSWDH. Finally, try changing the number of swoops generated (line 1420).

Do your blasting with less memory

Another program to produce laser blasts, LASER SWOOP 2, can do a similar job with about half of the memory. This is done by integrating both the Multi-Swooper and the Swoop generator into one program.




```
                1000 ************************************
                1010 ***                              ***
                1020 ***         LASER SWOOP 1        ***
                1030 ***                              ***
                1040 ***       REPRINTED FROM THE     ***
                1050 ***     FEBRUARY  1981 ISSUE OF  ***
                1060 ***       APPLE ASSEMBLY LINE    ***
                1070 ***      COPYRIGHT (C) 1981 BY   ***
                1080 ***           S-C SOFTWARE       ***
                1090 ***       ALL RIGHTS RESERVED    ***
                1100 ***                              ***
                1110 ************************************
                1120 *
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
0000-           1180 PULSCNT .EQ $0
0001-           1190 PULSWDH .EQ $1
0002-           1200 SWOOPCT .EQ $2
C030-           1210 SPEAKER .EQ $C030
                1220 *
                1230 *
0800- A9 01     1240 SWOOP   LDA #$1       One pulse at each width.
0802- 85 00     1250         STA PULSCNT
0804- A9 A0     1260         LDA #$A0      Start with maximum width of 160.
0806- 85 01     1270         STA PULSWDH
0808- A4 00     1280 LOOP1   LDY PULSCNT
080A- AD 30 C0  1290 LOOP2   LDA SPEAKER   Toggle the speaker.
080D- A6 01     1300         LDX PULSWDH
080F- CA        1310 LOOP3   DEX           Delay loop for one pulse.
0810- D0 FD     1320         BNE LOOP3
0812- 88        1330         DEY           Loop for number of pulses
0813- D0 F5     1340         BNE LOOP2     at each pulse width.
0815- C6 01     1350         DEC PULSWDH   Shrink pulse width
0817- D0 EF     1360         BNE LOOP1     to limit of zero.
0819- 60        1370         RTS
                1380 *
                1390 *
                1400 * Multi-Swooper
                1410 *
081A- A9 0A     1420 SWOOP2  LDA #$A       Number swoops
081C- 85 02     1430         STA SWOOPCT   Save it.
081E- 20 00 08  1440 LOOP4   JSR SWOOP     Do swoop.
0821- C6 02     1450         DEC SWOOPCT   Decrement the swoop count.
0823- D0 F9     1460         BNE LOOP4     Not done, do more.
0825- 60        1470         RTS
```


The number of swoops, or shots generated is set up in line 1210 of the program. In the next line, 1220, the width of the first pulse is set. Next, this width is stored temporarily in the accumulator, while the delay is being implemented (lines 1230 to 1250). After the delay, the original value that was in the X-register is restored (line 1260) and the speaker is toggled (line 1270).

Once the speaker has been toggled, the width of the pulse is incremented by one (line 1280) and a comparison is made to see if the maximum pulse width has been reached (line 1290). If not, the program jumps back to line 1230 to generate the delay for the next pulse. If the maximum pulse width has been reached, the number of shots is reduced by one (line 1310) until all have been generated.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***         LASER SWOOP 2        ***
                1030 ***                              ***
                1040 ***       REPRINTED FROM THE     ***
                1050 ***     FEBRUARY  1981 ISSUE OF  ***
                1060 ***       APPLE ASSEMBLY LINE    ***
                1070 ***      COPYRIGHT (C) 1981 BY   ***
                1080 ***           S-C SOFTWARE       ***
                1090 ***       ALL RIGHTS RESERVED    ***
                1100 ***                              ***
                1110 ************************************
                1120 *
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
C030-           1180 SPEAKER .EQ $C030
                1190 *
                1200 *
0800- A0 0A     1210         LDY #$A       Number of shots.
0802- A2 40     1220 LOOP1   LDX #$40      Pulse width of first pulse.
0804- 8A        1230 LOOP2   TXA           Start a pulse within a shot.
0805- CA        1240 LOOP3   DEX           Delay for one pulse.
0806- D0 FD     1250         BNE LOOP3
0808- AA        1260         TAX
0809- AD 30 C0  1270         LDA SPEAKER   Toggle the speaker.
080C- E8        1280         INX
080D- E0 C0     1290         CPX #$C0      Width of last pulse.
080F- D0 F3     1300         BNE LOOP2
0811- 88        1310         DEY           Done shooting?
0812- D0 EE     1320         BNE LOOP1     No.
0814- 60        1330         RTS
```

As with the last program, you should do some experimenting with this one too. The first thing you ought to do is try changing the values used as the minimum and maximum pulse widths (lines 1220 and 1290). You might also want to try changing the number of swoops generated (line 1210). Finally, you could try changing the direction of the changing pulse from an increasing width to a decreasing with. This can be done by changing the INX in line 1280 to a DEX.

Fifteen bytes to an alarm signal

Another sound effect that can come in quite handy is a siren generating routine. It may be used as part of an alarm routine, or a simulation game where emergency vehicle sirens are required.

The SIREN program presented here is very short, only fifteen bytes long, but will produce a wailing siren sound that repeatedly starts at a low frequency and goes higher until an upper limit is reached. Then it starts all over again.

The program starts on line 1180, where the X-register is loaded with a random value from location TEMP ($2FF). Next the speaker is toggled (line 1190). You'll notice, that while we have generally used an LDA instruction to toggle the speaker, here the BIT instruction is used. The program executes a delay based on the the value stored in the X-register (lines 1200 and 1210). Because the time delay generated is constantly shrinking, the frequency is constantly increasing. After that, the value in TEMP is reduced by one (line 1220) and the program jumps back to the beginning using a relative branch (lines 1230 and 1240). This makes the program independent of memory position.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***          SIREN PROGRAM       ***
                1030 ***                              ***
                1040 ***      COPYRIGHT (C) 1982 BY   ***
                1050 ***         JULES H. GILDER      ***
                1060 ***       ALL RIGHTS RESERVED    ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 * EQUATES
                1130 *
02FF-           1140 TEMP     .EQ $2FF
C030-           1150 SPEAKER  .EQ $C030
                1160 *
                1170 *
0800- AE FF 02  1180 START    LDX TEMP      Initialize the X-register.
0803- 2C 30 C0  1190          BIT SPEAKER   Toggle the speaker.
0806- CA        1200 LOOP     DEX           Decrement the X-register
0807- D0 FD     1210          BNE LOOP      until it equals zero.
0809- CE FF 02  1220          DEC TEMP      Decrement TEMP.
080C- B8        1230          CLV           Go back to
080D- 50 F1     1240          BVC START     the beginning.
```

The Apple sound hardware, unlike the hardware in other computers, is only capable of generating one tone at a time. So, if you want to generate more than one tone at the same time, you can't. Nevertheless, I'm sure most of you have heard programs that produce what appears to be multi-tone sounds. This is done by playing the sounds one after the other in quick succession and repeating the sequence several times. This fools the ear into thinking that the sounds occurred simultaneously. By using this technique, and generating the proper frequencies, it is possible for us to write a program that will simulate the tones generated by a Touch-Tone keypad, such as those found on telephones.

Simulate a Touch-Tone generator with your Apple

The TOUCH-TONE SIMULATOR program begins on line 1350 with a routine called TWOTONE. While the quality of the tones produced is not good enough to use with the telephone system, it's good enough for demonstration purposes. This routine contains a loop that first plays the low tone and then the high tone and is repeated ten times. The number of repetitions is determined by the value stored in CHRDTIM in lines 1350 and 1360.

The particular two tones that are generated are determined by what the value in BUTTON is. In line 1370, the X-register is loaded with the value of the button pressed (from 0 to 9) and used as an index into two tables that contain the data for generating the low and high tones (lines 1380 and 1400). After the appropriate data are retrieved, the program jumps to a subroutine (lines 1390 and 1410) that actually generates and plays the tones through the speaker. After one pair of tones have been played through the speaker, the program loops back to line 1370 and plays them again until the process has been repeated ten times (lines 1420 to 1440).

A 12-key Touch-Tone keypad uses 7 basic frequencies that are combined in a variety of ways to produce twelve tone pairs. When the program retrieved tone data from the LOTONE and HITONE tables in lines 1380 and 1400, the information it retrieved was merely a number from 0 to 6 that determined which of the seven tones was to be generated. When the program jumps to the ONETONE routine from lines 1390 and 1410, this number from 0 to 6 is still in the accumulator. The first thing that ONETONE does is to store this value in the Y-register, where it will be used as an index into a second set of data tables that determine the time required for each half cycle (up time and down time) and the number of half cycles to be generated. Lines 1520 to 1570 pick up the variables from the three tables and stores them in three page zero locations for use later.

The next subroutine, PLAY, is the one that actually produces sound in the speaker. It contains two identical routines (lines 1580 to 1640 and lines 1650 to 1710). One handles the up time and one the down time. The purpose of having two routines, is to be able to more closely approximate the desired frequency. For example, if the loop count we ought to use to get the desired frequency is 104.5, we could use an up time of 104 and a down time of 105; this makes the total time for the full cycle correct. Line 1640 has a redundant BEQ instruction, because if it is eliminated, the program will still go to LOOP3. The reason for this redundant instruction is to make the loop times for UPTIME and DWNTIME exactly the same. In most cases, the up time and the down time half cycles will be the same. In fact you can see this is so by looking at the data tables for both in lines 1940 and 1950. You'll see that except for the fourth entry in the table (an entry consists of two digits) all the data are exactly the same.

The TOUCH-TONE SIMULATOR program should be called with the number of the button pressed in location BUTTON. When this program was originally developed, it was part of an Applesoft program that was used as a telephone demonstration. The screen showed a Touch-Tone pad and as the user pressed one of the digits on the keyboard, the corresponding button on the screen would light up (display in the inverse mode). Then the Applesoft program called this machine language program to produce the twin-tone sound that the telephone makes. Since the Applesoft program is not included here, a short routine to drive the TOUCH-

TONE SIMULATOR is included. This routine starts on line 1790 and is called PUSHALL, because it simulates the pushing of all of the telephone buttons, one after the other.

The routine starts out by storing a zero in location BUTTON and then doing a subroutine jump to TWOTONE, the main simulator routine (lines 1790 to 1810). Next, a short waiting period is set up to produce a slight delay between simulated button presses (lines 1820 and 1830) and then the value in BUTTON is incremented by one and checked to make sure it ia less than 10 (lines 1840 to 1860). This continues until all buttons have been pressed.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***     TOUCH-TONE SIMULATOR     ***
                1030 ***                              ***
                1040 ***      REPRINTED FROM THE      ***
                1050 ***    FEBRUARY  1981 ISSUE OF   ***
                1060 ***      APPLE ASSEMBLY LINE     ***
                1070 ***     COPYRIGHT (C) 1981 BY    ***
                1080 ***          S-C SOFTWARE        ***
                1090 ***      ALL RIGHTS RESERVED     ***
                1100 ***                              ***
                1110 ************************************
                1120 *
                1130 *
                1140 *
                1150 *
                1160 * EQUATES
                1170 *
009D-           1180 DWNTIME .EQ $9D
009E-           1190 UPTIME  .EQ $9E
009F-           1200 LENGTH  .EQ $9F
00A0-           1210 CHRDTIM .EQ $A0
00E7-           1220 BUTTON  .EQ $E7
C030-           1230 SPEAKER .EQ $C030
FCA8-           1240 WAIT    .EQ $FCA8
                1250 *
                1260 *
                1270 * This is the main program loop where
                1280 * the low and the high tones are played
                1290 * alternately 10 times to make it sound
                1300 * like both tones are being played
                1310 * simultaneously.  The particular set
                1320 * of tones played are determined by the
                1330 * value of BUTTON.
                1340 *
0800- A9 0A     1350 TWOTONE LDA #$A         Set up loop for 10 times.
0802- 85 A0     1360         STA CHRDTIM
0804- A6 E7     1370 LOOP1   LDX BUTTON      Get the digit pressed.
0806- BD 6E 08  1380         LDA LOTONES,X   Get the data for the low tone.
0809- 20 17 08  1390         JSR ONETONE     Play it.
080C- BD 78 08  1400         LDA HITONES,X   Get data for high tone.
080F- 20 17 08  1410         JSR ONETONE     Play it.
0812- C6 A0     1420         DEC CHRDTIM     Reduce count until the
0814- D0 EE     1430         BNE LOOP1       tone pair is played 10 times.
0816- 60        1440         RTS
                1450 *
                1460 *
                1470 * This routine toggles the speaker for
                1480 * LENGTH number of half-cycles which are
                1490 * controlled by UPTIME or DWNTIME.
                1500 *
0817- A8        1510 ONETONE TAY             Use LO/HITONE data as index.
0818- B9 59 08  1520         LDA DNTMTAB,Y   Get down-time data
081B- 85 9D     1530         STA DWNTIME     and store it.
081D- B9 60 08  1540         LDA UPTMTAB,Y   Get up time data
0820- 85 9E     1550         STA UPTIME      and store it.
```

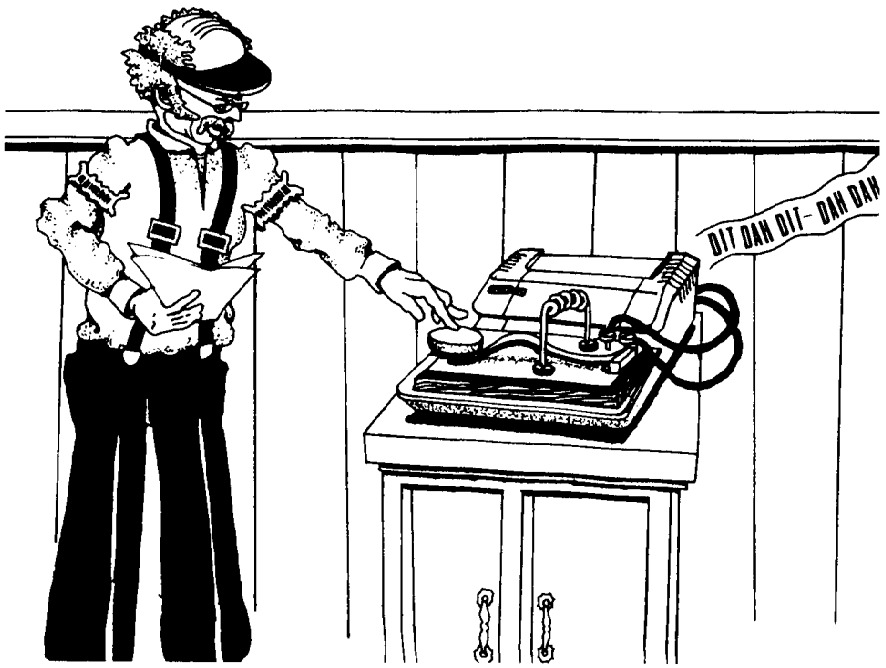


```
0822- B9 67 08  1560         LDA LENTABL,Y   Get number of half cycles
0825- 85 9F     1570         STA LENGTH      and store it.
0827- A4 9E     1580 PLAY    LDY UPTIME      Use UPTIME as counter.
0829- AD 30 C0  1590         LDA SPEAKER     Toggle the speaker.
082C- C6 9F     1600         DEC LENGTH      Reduce LENGTH until done.
082E- F0 13     1610         BEQ RETURN      Done, return to caller.
0830- 88        1620 LOOP2   DEY              Delay by UPTIME.
0831- D0 FD     1630         BNE LOOP2
0833- F0 00     1640         BEQ LOOP3       This is for timing symmetry.
0835- A4 9D     1650 LOOP3   LDY DWNTIME     Use DWNTIME as counter.
0837- AD 30 C0  1660         LDA SPEAKER     Toggle the speaker.
083A- C6 9F     1670         DEC LENGTH      Reduce LENGTH until done.
083C- F0 05     1680         BEQ RETURN      Done, return to caller.
083E- 88        1690 LOOP4   DEY              Delay by DWNTIME.
083F- D0 FD     1700         BNE LOOP4
0841- F0 E4     1710         BEQ PLAY        Play next half cycle.
0843- 60        1720 RETURN  RTS
                1730 *
                1740 *
                1750 * This routine automatically simulates
                1760 * the pushing of each of the buttons
                1770 * from 0 to 9.
                1780 *
0844- A9 00     1790 PUSHALL LDA #$0         Simulate button 0
0846- 85 E7     1800         STA BUTTON      being pressed.
0848- 20 00 08  1810 LOOP5   JSR TWOTONE     Generate the tone.
084B- A9 00     1820         LDA #$0         Delay between pressing
084D- 20 A8 FC  1830         JSR WAIT        of buttons.
0850- E6 E7     1840         INC BUTTON      Get ready for next button.
0852- A5 E7     1850         LDA BUTTON      Get next button pressed.
0854- C9 0A     1860         CMP #$A         Did we reach 10?
0856- 90 F0     1870         BCC LOOP5       No, generate tone.
0858- 60        1880         RTS             Yes, that's all!
                1890 *
                1900 *
                1910 * These are the various data tables that
                1920 * are required by this program.
                1930 *
0859- 8E 80 74
085C- 68 51 49
085F- 42        1940 DNTMTAB .HS 8E807468514942
0860- 8E 80 74
0863- 69 51 49
0866- 42        1950 UPTMTAB .HS 8E807469514942
0867- 14 12 10
086A- 0F 20 1D
086D- 1A        1960 LENTABL .HS 1412100F201D1A
086E- 03 00 00
0871- 00 01 01
0874- 01 02 02
0877- 02        1970 LOTONES .HS 03000000010101020202
0878- 05 04 05
087B- 06 04 05
087E- 06 04 05
0881- 06        1980 HITONES .HS 05040506040506040506
```

Let your computer send Morse code like a pro

For those of you who have an interest in Ham radio, this next program should be of considerable interest. Like so many of the programs in this chapter, it was written by Bob Sander-Cederlof. I've made one or two slight modifications and rearranged the source code a bit, but the bulk of the work was Bob's. This program works by stealing control away from the output and so, not surprisingly, the program starts out by setting up the output hooks to point to the appropriate part of this program (lines 1320 to 1400).

The replacement output program starts on line 1500, where the character that is



Send Morse Code like a pro with your Apple computer.

being output is tested to see if it is a letter or a number. The reason for this test is that not all of the ASCII set has been encoded for this program, just the letters and numbers. But if you wish to extend this to include the punctuation as well, after seeing how it's done here, you'll find it very easy to do. If you're going to do this, it will be necessary to change the $B0 in line 1500 to $A0 and add the extra codes to the code table (lines 2390 to 2530). If it is determined that the character to be printed is not a letter or a number, and thus not in the code table, the program branches to line 1550, where the character is output to the screen.

If the character is a letter or number, it is temporarily stored on the stack, while the program jumps to the SENDCHR routine in line 1620. After it comes back from that subroutine jump, the character is restored from the stack and then printed out to the screen (lines 1540 and 1550).

The heart of this program is the subroutine called SENDCHR which starts on line 1620. Since the X and Y registers are going to be used by this routine, their contents are saved at entry (lines 1620 and 1630) and will be restored before exiting. Next, the character that was entered is normalized by subtracting $B0 from it (line 1650). This allows the resulting number to be used as an index (line 1660) into the CODES table (line 1670) to retrieve the information needed to generate the appropriate sequences of dits and dahs (dots and dashes).

The number retrieved from the CODES table contains two pieces of information: the number of code elements (the total number of dots and dashes), which is stored in the three low-order bits, and the actual code elements (dots and dashes) themselves, which are stored in the five high order bits. The number of code elements is retrieved from the table data by ANDing the data with $7. If the result of this ANDing is zero, there is no code for the character in the table and the program branches to a routine (line 1700) that generates three character spaces and restores the previously saved registers. By the way, when we talk about character spaces and element spaces here, we're referring to spaces in time, or time delays between characters and elements.

If the element count is not zero, it is stored in a location called COUNT (line 1710). Once this is done, the contents of CODE, which contains the original data retrieved from the CODES table, is shifted left one bit, causing the high-order bit to be placed into the CARRY location of the 6502 microprocessor (line 1720). The dits and dahs (or dots and dashes) or Morse code, are represented here as bits of zero and one, respectively. So if the value in the CARRY bit is a zero (line 1730) a dit is generated. If it's a one however, a dah is generated by calling the dit routine three times in succession, thus producing a longer beep (lines 1740 to 1760). Whether a dit or a dah was generated, the next thing that happens is a space (time delay), equal to the time it takes to send a dit, is generated (line 1770). Since this is a space between elements, it is referred to as an 'element space'.

Now that the program has sent one element, the element count is reduced by one (line 1780) and the program loops back to handle the remaining elements (line 1790). When all of the elements of a character have been sent, a space, equal to three element spaces — or the amount of time required to send a dash — is generated (line 1800). Finally, the X and Y registers are restored and control is returned to the calling program (lines 1810 to 1830).

The next subroutine SPACE1 on line 1980, generates the required spacing between elements of a character (the dits and dahs) and between characters too. Element spacing is handled by SPACE2, while the space between characters, which is three times longer than the space between elements, is done by SPACE1, which simply does a subroutine jump to SPACE2 twice (lines 1980 and 1990), and falls into it for the third time.

SPACE2 starts on line 2000 where it loads the Y-register with the value for the speed of transmission. Next, the X-register is loaded with the value of the pitch (line 2010). Since this routine is not supposed to produce any sound, a dummy location (the keyboard) is toggled instead of the speaker (line 2020). Next, the pitch delay loop is executed (lines 2030 and 2040). After that, the speed constant is decremented until it reaches zero, when an RTS instruction is executed (lines 2050 to 2070).

The DIT routine (lines 2150 to 2220) is identical to the SPACE2 routine except the speaker is toggled (line 2170) instead of the dummy location in the previous routine.

If the code being transmitted is too fast, you can slow it down by increasing the value of either SPEED or PITCH or both.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***     MORSE CODE GENERATOR     ***
                1030 ***                              ***
                1040 ***      REPRINTED FROM THE      ***
                1050 ***    FEBRUARY  1981 ISSUE OF   ***
                1060 ***      APPLE ASSEMBLY LINE     ***
                1070 ***     COPYRIGHT (C) 1981 BY    ***
                1080 ***          S-C SOFTWARE        ***
                1090 ***     ALL RIGHTS RESERVED      ***
                1100 ***                              ***
                1110 ***  MODIFIED BY JULES H. GILDER  ***
                1120 ***                              ***
                1130 ************************************
                1140 *
                1141 *
                1142 *
                1143 * CONSTANTS
                1144 *
004C-           1145 JUMP     .EQ $4C
0050-           1146 PITCH    .EQ $50
0078-           1147 SPEED    .EQ $78
                1150 *
                1160 *
                1170 * EQUATES
                1180 *
0036-           1190 CSWL     .EQ $36
03D0-           1200 WARMDOS  .EQ $3D0
03EA-           1210 CONNECT  .EQ $3EA
C000-           1220 DUMMY    .EQ $C000
C030-           1230 SPEAKER  .EQ $C030
FDF0-           1240 COUT1    .EQ $FDF0
                1250 *
                1260 *
                1270 * This subroutine steals control away
                1280 * from the normal output routine and
                1290 * directs all outputted characters to
                1300 * this program.
                1310 *
0800- A9 13     1320 SETUP    LDA #MORSE    Get the address of the
0802- A0 08     1330          LDY /MORSE    start of the program
0804- 85 36     1340          STA CSWL      and store it in
0806- 84 37     1350          STY CSWL+1    the output hooks.
0808- AD D0 03  1360          LDA WARMDOS   Check if DOS
080B- C9 4C     1370          CMP #JUMP     present.
080D- D0 03     1380          BNE NODOS     No, return.
080F- 20 EA 03  1390          JSR CONNECT   Yes, connect to DOS.
0812- 60        1400 NODOS    RTS           Return.
                1410 *
                1420 *
                1430 * This routine checks to see if the
                1440 * character being sent is a letter or a
                1450 * number.  If it isn't the character is
                1460 * just printed to the screen.  If it is
                1470 * the character is sent in Morse Code
                1480 * and then printed to the screen.
                1490 *
0813- C9 B0     1500 MORSE    CMP #$B0      Is it alphanumeric?
0815- 90 05     1510          BCC PRNTCHR   No, print it.
0817- 48        1520          PHA           Yes, save it.
0818- 20 1F 08  1530          JSR SENDCHR   Send it in Morse.
081B- 68        1540          PLA           Retrieve the character.
081C- 4C F0 FD  1550 PRNTCHR  JMP COUT1     Print it.
                1560 *
                1570 *
                1580 * This is the routine that converts the
                1590 * character to Morse Code and drives
                1600 * the speaker.
                1610 *
081F- 8E 81 08  1620 SENDCHR  STX SAVEX     Save the X and Y
0822- 8C 82 08  1630          STY SAVEY     registers.
0825- 38        1640          SEC           Normalize by
0826- E9 B0     1650          SBC #$B0      subtracting $B0.
```



```
0828- AA        1660          TAX           Use the result as an
0829- BD 85 08  1670          LDA CODES,X   index into CODES.
082C- 8D 84 08  1680          STA CODE      Get the
082F- 29 07     1690          AND #$7       element count.
0831- F0 23     1700          BEQ THRESPC   No code.
0833- 8D 83 08  1710          STA COUNT     Save the count.
0836- 0E 84 08  1720 LOOP1    ASL CODE      Put next element into carry bit.
0839- 90 06     1730          BCC MAKEDIT   Make a dit.
083B- 20 73 08  1740          JSR DIT       Make a dah (from
083E- 20 73 08  1750          JSR DIT       3 dits).
0841- 20 73 08  1760 MAKEDIT  JSR DIT       Make a dit.
0844- 20 65 08  1770          JSR SPACE2    Element space.
0847- CE 83 08  1780          DEC COUNT     Decrement element count.
084A- D0 EA     1790          BNE LOOP1     Next element.
084C- 20 5F 08  1800 LOOP2    JSR SPACE1    Character space.
084F- AE 81 08  1810          LDX SAVEX     Restore X and
0852- AC 82 08  1820          LDY SAVEY     Y registers.
0855- 60        1830          RTS           Return.
0856- 20 5F 08  1840 THRESPC  JSR SPACE1    Send character space.
0859- 20 5F 08  1850          JSR SPACE1    Send character space.
085C- 4C 4C 08  1860          JMP LOOP2     Send character space and exit.
                1870 *
                1880 *
                1890 * This subroutine generates the
                1900 * required spacing between the elements
                1910 * of a character (dits and dahs) and
                1920 * also between characters.  Element
                1930 * spacing is handled by SPACE2 while
                1940 * the space between characters (which
                1950 * is 3 times longer than the space
                1960 * between elements) is done by SPACE1.
                1970 *
085F- 20 65 08  1980 SPACE1   JSR SPACE2    Do an element space.
0862- 20 65 08  1990          JSR SPACE2    Do an element space.
0865- A0 78     2000 SPACE2   LDY #SPEED    Make believe it's
0867- A2 50     2010 GTPITCH  LDX #PITCH    sending Morse
0869- AD 00 C0  2020          LDA DUMMY     But toggle the keyboard
086C- CA        2030 LOOP3    DEX           instead of the speaker.
086D- D0 FD     2040          BNE LOOP3
086F- 88        2050          DEY
0870- D0 F5     2060          BNE GTPITCH
0872- 60        2070          RTS
                2080 *
                2090 *
                2100 * This subroutine is identical to the
                2110 * previous one, but instead of toggling
                2120 * the keyboard, it toggles the speaker and
                2130 * generates a sound.
                2140 *
0873- A0 78     2150 DIT      LDY #SPEED    Get the speed.
0875- A2 50     2160 LOOP4    LDX #PITCH    Get the pitch.
0877- AD 30 C0  2170          LDA SPEAKER   Toggle the speaker.
087A- CA        2180 LOOP5    DEX           Decrement the pitch
087B- D0 FD     2190          BNE LOOP5     delay.
087D- 88        2200          DEY           Decrement the speed
087E- D0 F5     2210          BNE LOOP4     delay.
0880- 60        2220          RTS
                2230 *
                2240 *
                2250 * These are temporary storage locations
                2260 * used by the program.
                2270 *
0881- 00        2280 SAVEX    BRK
0882- 00        2290 SAVEY    BRK
0883- 00        2300 COUNT    BRK
0884- 00        2310 CODE     BRK
                2320 *
                2330 *
                2340 * These are the code tables used to
                2350 * convert the letters into Morse Code.
                2360 *
                2370 * 0, 1 through 9
                2380 *
```

```
0885- FD 7D 3D
0888- 1D 0D 05
088B- 85 C5 E5
088E- F5        2390 CODES   .HS FD7D3D1D0D0585C5E5F5
088F- 00 00 00
0892- 00 00 00 2400          .HS 000000000000
               2410 *
               2420 *
               2430 * @, A through M
               2440 *
0895- 00 42 84
0898- A4 83 01
089B- 24 C3 04
089E- 02        2450         .HS 004284A4830124C30402
089F- 74 A3 44
08A2- C2        2460         .HS 74A344C2
               2470 *
               2480 *
               2490 * N through Z
               2500 *
08A3- 82 E3 64
08A6- D4 43 03
08A9- 81 23 14
08AC- 63        2510         .HS 82E364D4430381231463
08AD- 94 B4 C4 2520          .HS 94B4C4
08B0- 00 00 00
08B3- 00 00 00 2530          .HS 000000000000
```

How to copy any cassette program

Even though most Apple owners have at least one disk drive, occasionally the need arises to duplicate an Apple cassette program. If the program is unprotected (yes there are cassette protection schemes too) and only one program is on the cassette, it's a relatively simple matter to load the program into memory and then save it out again on a fresh cassette. However, if even one of these conditions is not true (e.g. there is more than one program one the cassette — possibly a mixture of BASIC and machine language programs — and/or the program is protected) then it is much easier to use the CASSETTE DUPLICATOR program to copy the cassette.

The program starts out by printing out the title of the program (lines 1270 to 1330) and then branches to line 1450 where it scans the keyboard to see if a key has been pressed. Whether or not a key has been pressed, it takes the value it got from the keyboard location and tests it to see if it is equal to the value generated by the ESCape key (line 1460). If it is, the program toggles the keyboard strobe and returns to the caller (line 1470).

If the ESCape key has not been pressed, the program does a subroutine jump to a monitor routine called RDBIT ($FCFD) which reads one bit of data off the tape (line 1480). Next, the speaker is toggled (line 1490) so the user gets some audible feedback on what's happening and then the cassette output port is toggled so the data is written out to the new cassette. Finally, the program jumps back to the beginning to check the keyboard again and then read the next bit off the tape (line 1510).

This program requires the use of two tape recorders, one to read the data from and one to write the data to. Even though the Apple was only designed for use with

one recorder, the two can be used as long as the input lead is connected to one recorder and the output lead to the other. Cassettes produced in this manner are as good as the original and do not suffer any multiple generation degradation (e.g. the copy is worse than the original and a copy of a copy is even worse). The reason for this is the data are being read into the computer and a new version of the same data is being written out, just as though it was an original.

The monitoring capability through the Apple's internal speaker is very important because it lets you hear what's on the tape you're duplicating so that you will know when you have reached the end of the record data. When that happens, all you have to do is press the ESCape key to exit the program.

```
               1000 ************************************
               1010 ***                              ***
               1020 ***      CASSETTE DUPLICATOR     ***
               1030 ***                              ***
               1040 ***    COPYRIGHT (C) 1982 BY     ***
               1050 ***       JULES H. GILDER        ***
               1060 ***     ALL RIGHTS RESERVED      ***
               1070 ***                              ***
               1080 ************************************
               1090 *
               1100 *
               1110 *
               1120 * EQUATES
               1130 *
C000-          1140 KEYBD    .EQ $C000
C010-          1150 KBDSTRB  .EQ $C010
C020-          1160 CASSOUT  .EQ $C020
C030-          1170 SPEAKER  .EQ $C030
FC58-          1180 HOME     .EQ $FC58
FCFD-          1190 RDBIT    .EQ $FCFD
FD0C-          1200 RDKEY    .EQ $FD0C
FDED-          1210 COUT     .EQ $FDED
               1220 *
               1230 *
               1240 * This section prints out the title
               1250 * and copyright notice.
               1260 *
0800- 20 58 FC 1270          JSR HOME      Clear the screen.
0803- A0 00    1280          LDY #$0       Initialze character pointer.
0805- B9 27 08 1290 LOOP     LDA TEXT,Y    Get a character.
0808- F0 06    1300          BEQ START     Done, run program.
080A- 20 ED FD 1310          JSR COUT      Print character.
080D- C8       1320          INY           Increment pointer.
080E- D0 F5    1330          BNE LOOP      Get next character.
               1340 *
               1350 * This section constantly monitors the
               1360 * keyboard to see if the ESCape key is
               1370 * being pressed.  If not it reads in
               1380 * data from a cassette on one tape
               1390 * recorder and writes it out to another
               1400 * tape recorder.  At the same time, it
               1410 * also toggles the speaker so that you
               1420 * can listen to the tape as it is being copied
               1430 * and will know when it is done.
               1440 *
0810- AD 00 C0 1450 START    LDA KEYBD     Read the keyboard.
0813- C9 9B    1460          CMP #$9B      Is it ESCape?
0815- F0 0C    1470          BEQ QUIT      Yes, quit.
0817- 20 FD FC 1480          JSR RDBIT     No, read tape.
081A- AD 30 C0 1490          LDA SPEAKER   Toggle speaker.
081D- AD 20 C0 1500          LDA CASSOUT   Toggle cassette.
0820- 4C 10 08 1510          JMP START     Read next bit from tape.
0823- 2C 10 C0 1520 QUIT     BIT KBDSTRB   Clear keyboard strobe.
0826- 60       1530          RTS           Return to caller.
```

```
                1540 *
                1550 *
                1560 * This is the text printed out by
                1570 * the program.
                1580 *
0827- C3 C1 D3
082A- D3 C5 D4
082D- D4 C5 A0
0830- C4 D5 D0
0833- CC C9 C3
0836- C1 D4 CF
0839- D2        1590 TEXT     .AS -"CASSETTE DUPLICATOR"
083A- 8D 8D     1600          .HS 8D8D
083C- C2 D9 A0
083F- CA D5 CC
0842- C5 D3 A0
0845- C8 AE A0
0848- C7 C9 CC
084B- C4 C5 D2  1610          .AS -"BY JULES H. GILDER"
084E- 8D        1620          .HS 8D
084F- C3 CF D0
0852- D9 D2 C9
0855- C7 C8 D4
0858- A0 A8 C3
085B- A9 A0 B1
085E- B9 B8 B2  1630          .AS -"COPYRIGHT (C) 1982"
0861- 8D        1640          .HS 8D
0862- C1 CC CC
0865- A0 D2 C9
0868- C7 C8 D4
086B- D3 A0 D2
086E- C5 D3 C5
0871- D2 D6 C5
0874- C4        1650          .AS -"ALL RIGHTS RESERVED"
0875- 8D 8D 8D
0878- 8D 00     1660          .HS 8D8D8D8D00
```

Chapter 7

LEARNING TO USE THE AMPERSAND

While it's possible to do anything in machine language that you can do in Applesoft, it may not always be advisable. Sometimes it may be faster and easier to develop most of your program in Applesoft, and only use a machine language routine to speed up time-critical sections of the program. This is frequently done with business software so that the user can customize it to his own needs by just modifying the Applesoft program, but still have the speed he needs in, for example, sorting routines.

Because the designers of the Applesoft language foresaw the probable need to couple Applesoft with machine language routines, they provided several ways of doing it, including CALL, USR (X) and &. It is this last method, using the ampersand (&), that we are going to discuss in this chapter.

One of the tokens, or reserved words, in Applesoft is not a word but a single character, the ampersand (&), also known as the 'and' sign. This Applesoft command works just like the PRINT command or any other Applesoft command. When the Applesoft interpreter sees an '&', it jumps to the routine that handles it. The big difference between this command and most of the other Applesoft commands is that the address the computer jumps to for this command is not in any of the Applesoft ROMs, as the others are, but is in page three, specifically at address $3F5. There is no machine language code to process the command, only three reserved locations which can be used to store a command to jump to some other location in memory to the desired subroutine. Thus, what happens when an '&' is encountered is the program jumps to $3F5, where it expects to find another jump command.

Data can be passed with the ampersand too

While the primary purpose of the ampersand is to transfer control to a machine language program, it should be noted that it is also possible to transfer data, with the ampersand command. There is a short routine, called CHRGET, in page zero that starts at location $B1 that is used to interpret the lines of an Applesoft program. We'll go into a deeper discussion of this later on in the book, but suffice it to say, that as each command is encountered, a text pointer is advanced to interpret each token or character. One other feature of this routine is that it ignores spaces.

After the routine has interpreted a character or token — such as the ampersand — it leaves the text pointer pointing to the character that follows it if there is one and

loads that character into the accumulator. By taking advantage of this fact, and using some of the routines built into the Applesoft ROMs, it is possible to pass a wide variety of data to machine languages programs that are called via the ampersand. The first program we are going to look at — HEX/DECIMAL/HEX CONVERTER — passes both a symbol (a dollar sign) and a number.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***   HEX/DECIMAL/HEX CONVERTER   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $300
                1130 *
                1140 *
                1150 * EQUATES
                1160 *
003E-           1170 A2L      .EQ $3E
0050-           1180 LINNUM   .EQ $50
00B1-           1190 CHRGET   .EQ $B1
0200-           1200 IN       .EQ $200
03F5-           1210 AMPERSD  .EQ $3F5
DD67-           1220 FRMNUM   .EQ $DD67
E199-           1230 IQERR    .EQ $E199
E752-           1240 GETADR   .EQ $E752
ED24-           1250 LINPRT   .EQ $ED24
FDDA-           1260 PRBYTE   .EQ $FDDA
FDED-           1270 COUT     .EQ $FDED
FFA7-           1280 GETNUM   .EQ $FFA7
                1290 *
                1300 *
                1310 *
                1320 * This is where the ampersand (&) vector
                1330 * jump is set up.
                1340 *
0300- A2 4C     1350          LDX #$4C        Get JMP op code and
0302- A9 10     1360          LDA #START      the low and high bytes
0304- A0 03     1370          LDY /START      of START's address and
0306- 8E F5 03  1380          STX AMPERSD     store them in locations
0309- 8D F6 03  1390          STA AMPERSD+1   $3F5, $3F6 and $3F7.
030C- 8C F7 03  1400          STY AMPERSD+2
030F- 60        1410          RTS
                1420 *
                1430 *
                1440 * This part of the program checks to
                1450 * see if the character immediately following
                1460 * the ampersand (&) was a dollar sign.
                1470 * If it was, control is passed to the
                1480 * routine that converts from hexadecimal
                1490 * to decimal.  Otherwise the number is
                1500 * decimal and converted to hexadecimal.
                1510 *
0310- C9 24     1520 START    CMP #$24        Is it a dollar sign ($)?
0312- F0 17     1530          BEQ HEXIN       Yes, convert hex to decimal.
0314- 20 67 DD  1540          JSR FRMNUM      No, evaluate number or formula.
0317- 20 52 E7  1550          JSR GETADR      Convert to integer form.
031A- A9 A4     1560          LDA #$A4        Output a dollar sign ($).
031C- 20 ED FD  1570          JSR COUT
031F- A5 51     1580          LDA LINNUM+1    Get high byte.
0321- F0 03     1590          BEQ PRINTLO     If zero, get low byte.
0323- 20 DA FD  1600          JSR PRBYTE      Otherwise print high byte.
```



```
0326- A5 50     1610 PRINTLO LDA LINNUM      Get low byte.
0328- 4C DA FD  1620         JMP PRBYTE      Print it.
                1630 *
                1640 *
                1650 * This routine handles the hexadecimal
                1660 * to decimal conversion.
                1670 *
032B- A0 00     1680 HEXIN    LDY #$0         Zero offset index.
032D- 20 B1 00  1690 HEXIN2   JSR CHRGET      Get the next character.
0330- F0 08     1700          BEQ PUTBUF      Store in buffer and convert.
0332- 49 80     1710          EOR #$80        Set high bit.
0334- 99 00 02  1720          STA IN,Y        Store in input buffer.
0337- C8        1730          INY             Increment offset index.
0338- D0 F3     1740          BNE HEXIN2      Get next character.
033A- 99 00 02  1750 PUTBUF   STA IN,Y        Store zero in buffer.
033D- A8        1760          TAY             Zero offset index.
033E- 20 A7 FF  1770          JSR GETNUM      Convert ASCII to hex.
0341- A6 3E     1780          LDX A2L         Store low byte in X-register.
0343- A5 3F     1790          LDA A2L+1       Store high byte in Y-register.
0345- C0 06     1800          CPY #$6         Check if number too large.
0347- 90 03     1810          BCC INRANGE     No, it's okay.
0349- 4C 99 E1  1820          JMP IQERR       Yes, print error message.
034C- C0 03     1830 INRANGE  CPY #$3         Converting only 1 byte?
034E- B0 02     1840          BCS PRINTIT     No, do both.
0350- A9 00     1850          LDA #$0         Yes, do just one.
0352- 4C 24 ED  1860 PRINTIT  JMP LINPRT      Convert and print number.
```

Converting between decimal and hexadecimal

As you work more and more with machine-language programs and write routines that can be used with Applesoft, you will frequently find the need to convert numbers from decimal to hexadecimal and vice versa. Perhaps you've written a program that starts at memory location $9400 and you want to know what the decimal equivalent is so that you can call it from a BASIC program. Or perhaps one of the functions your BASIC program does is display a section of memory with its contents in hexadecimal notation.

You can do all of this in BASIC if you choose to, but it will significantly slow down your program. The alternative is to use a machine-language program to do the conversions for you. That's where the HEX/DECIMAL/HEX CONVERTER program comes in. This program will allow you to convert numbers in either direction. An added advantage of the program is that it does not have to be used in an Applesoft program only, but can also be used in the immediate mode.

The program starts with a short routine (line 1350), whose only purpose is to activate the '&' jump locations. These jump locations are also referred to as jump 'vectors'. When the program is BRUN or a CALL 768 is issued, the program loads locations $3F5, $3F6 and $3F7 with a jump op code and the address of the converter part of the program, which starts on line 1520.

Since we know that the accumulator contains the character following the ampersand, if there is one, the first thing that our program does is to test the contents of the accumulator and see if the character there is a dollar sign. If it is, that's a sign to the program that the number that follows is a hexadecimal number and it is to be converted to decimal. Thus, the program branches (line 1530) to the routine that handles the hexadecimal-to-decimal conversion.

Using the Applesoft ROM routines

If there is no dollar sign in the accumulator, the assumption is made that whatever follows the ampersand has a decimal value and it is to be converted to a hexadecimal number. To do this, we use some of the routines in the Applesoft ROMs. The first one is called FRMNUM and is located at $DD67. This routine assumes that the text pointer from the CHRGET routine at $B1 is pointing to the first character of the number, variable or formula. FRMNUM takes this number, variable or formula and converts its value into a special format called floating point and stores this information in six special locations on page zero called the 'floating point accumulator', which is often abbreviated to the three letters FAC. The FAC is in locations $9D to $A2.

Getting the number into the FAC is only the first step, and the number is not useful to us in this form. The reason is these six locations contain an exponent, four mantissa bytes and a sign byte. In addition, the data are stored in a form known as 'excess $80'. This means that it has $80 added to it. The actual mathematics can be a little confusing, but you shouldn't worry about it, because the folks that wrote the Applesoft language did all the hard work for us. All we have to do is use the routines that they wrote.

Converting floating point to integer

Once the number is in the floating point accumulator, we can use another ROM routine to convert it into an integer number that is represented by two hexadecimal bytes. This routine is called GETADR and is located at $E752 in the ROMs. It is designed to take any number in the -65535 to +65535 range, that is currently stored in the FAC, and convert it into a two-byte integer. These two bytes will be stored in a memory location called LINNUM and LINNUM + 1, which are also located on page zero at $50 and $51.

GETADR is one of several routines in the ROMs that converts numbers in FAC to integers. Another routine called QINT at $EBF2 could also be used, but it is limited to positive numbers only.

After the program does a JSR to GETADR (line 1550), a dollar sign is printed out (lines 1560 and 1570) to indicate that the number being printed is a hexadecimal number. Then the most significant byte of the number, which is stored in LINNUM + 1, is loaded into the accumulator and tested to see if it's zero (lines 1580 and 1590). If it is, it's discarded and the least significant, or low-order byte is put into the accumulator and printed (lines 1610 and 1620). If the high-order byte was not zero, it is printed (line 1600) and then the low-order byte is printed. This subroutine returns to the caller via the RTS instruction in the PRBYTE routine.

Doing the hex to decimal conversion

Earlier, we said that if the first character following the ampersand is a dollar sign, we knew that the number following it would be hexadecimal, and thus the desired conversion was hexadecimal-to-decimal. The routine that does this conversion starts on line 1680, where the Y-register, which is going to be used as an offset into the input buffer, is set to zero. Then the next character after the dollar sign is retrieved by using the CHRGET routine at $B1 (line 1690). This routine increments the text pointer and loads the next character into the accumulator. Once there, the character is checked to see if it is a zero (line 1700). If it is, CHRGET knows that it has reached the end of the program (or input) line from which text is being read, and the zero is stored as-is in the input buffer (line 1750). If the character was not a zero, it is exclusive-ORed with $80 to set the high bit (bit number 7) and then stored in the input buffer with the high bit set (lines 1710 and 1720). By doing this, we're making it appear as if the text were entered from the keyboard. Next the Y-register is incremented and the next character is retrieved (lines 1730 and 1740). This process continues until all the text following the ampersand has been placed in the input buffer or until 256 characters have been processed.

Once the zero, which indicates the end of text, has been entered into the input buffer, the Y-register is reset to zero and a routine, GETNUM, in the Apples's F8 monitor ROM is called (lines 1760 and 1770). This is a routine that gets called when hex data are entered from the keyboard while in the monitor mode. This routine scans the input buffer and converts the ASCII data it finds there into a hexadecimal number. Its stores the two bytes of this number in two frequently used utility locations A2L and A2L + 1 (often referred to as A2H). The converted hex data, that are stored in A2L and A2L + 1, are loaded into the X-register and the accumulator (lines 1780 and 1790) in preparation for its conversion and printing as a decimal number.

The GETNUM routine does not check to see that only four hex digits are entered. Instead, it just converts the last four digits entered. In order for us to make sure that the number entered is not to large, we check the Y-register (line 1800), which is incremented, by GETNUM, for each character that is retrieved from the input buffer. We check to make sure that no more than 6 places have been used in the input buffer. Six was chosen because we need a maximum of four for the hex data, one for the zero and one more because the Y-register is incremented in GETNUM, before it returns.

If more than four hex digits were entered the program jumps to the routine in the Applesoft ROMs that prints the ILLEGAL QUANTITY error message. Otherwise, a test is made to see if only two digits, and hence one byte of data is being converted (line 1840). If it is only one byte, the accumulator is set to zero (line 1850), otherwise things are left as is. In any case, the program then jumps to still another Applesoft routine, LINPRT ($ED24). This is the routine that is called when a program is being listed and the line number has to be printed out. LINPRT does the actual hex to decimal conversion and also prints out the number.

As you can see from the preceding explanation and from the listing, we've made liberal use of the routines that exist in the Apple's ROMs to cut down on the amount of programming we would otherwise have had to do. Whenever possible, it is

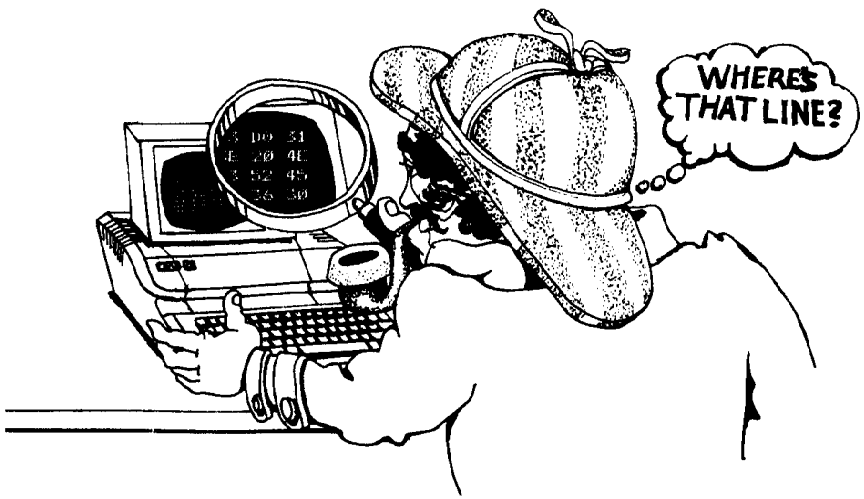
advisable to use the routines in the ROMs. This will speed up development time considerably. Just remember, when you do that, you tie yourself to those ROMs, so if Applesoft routines are being used, you have to ensure that they are in the machine and active.

Locate Applesoft program lines in memory

Did you ever see an Applesoft program with illegal line numbers (numbers greater than 63999) and wonder how they were entered? Or perhaps you've seen lines that have imbedded back spaces, so they look invisible when listed to the video display, and couldn't figure out how they were entered. Or maybe you want to use a character in your program that is not accessible from the Apple keyboard, such as the left square bracket ([).

If you've ever wondered just how these things are done, they're done by a process known as 'patching', which means the line is first entered with dummy legal characters to hold the place required for the illegal ones, and then the programmer goes into the monitor mode, finds the particular line he's trying to change, or patch, and then makes the required changes. Making the changes is easy. The difficult part is finding just where in memory the line you're interested in resides. Usually you have to start at the beginning of the Applesoft program and follow the next line pointers down until you find the line you desire. This can be a time consuming and cumbersome process, which is probably why more people don't patch programs. But, with the next program we are going to discuss, the task becomes trivial.

In Chapter 2 when we spoke about the Applesoft Program Line Counter, we had



a short discussion on the way a line of an Applesoft program is stored in memory. Without repeating that discussion in detail, let's just review a few pertinent facts. With ROM or language card Applesoft, program storage normally starts at location $801. The first two bytes of an Applesoft line contain a pointer to the location in memory of the next Applesoft line. The next two bytes are reserved for the hex representation of the line number. Then, the actual text of the line is stored with Applesoft keywords replaced by one-byte tokens. Finally, the line is terminated with a zero.

The program APPLESOFT LINE FINDER, takes a line number that is passed to it by the ampersand command and uses some of the routines in the Apple ROMs to first locate the position of the line in memory and then display the line in hex up to and including the terminating zero byte. The program then leaves you in the monitor mode so that you can make any changes desired in the line just displayed.

The program starts at location $2DA, which is the upper part of the input buffer. To use it, the program is loaded and then activated by a CALL 730. Since the program is located in pages 2 and 3 of memory, it can be loaded and run at any time during an Applesoft program's development, without affecting the Applesoft program.

The first part of the program, which starts on line 1360, clears the screen, prints out the program title and sets up the ampersand jump locations on page three to point to a routine that locates the Applesoft line. Immediately following this short routine, is the text that it prints out. As in some earlier programs, the reason the text is placed here up front, is that it is going to be used once, the first time the program is run, and thus is expendable. So we won't have to worry about part of our program, which is stored in the input buffer, being wiped out if a long line of text is entered.

The actual program that finds and displays Applesoft lines starts on line 1660, where an Applesoft routine called LINGET ($DA0C) is called. LINGET is the routine that is used to check get the line number of an Applesoft line that is being entered from the keyboard. It uses TXTPTR, which is the text pointer in the CHRGET routine, and reads the number that TXTPTR is pointing to. It takes this number, converts it to hexadecimal and stores it in LINNUM and LINNUM + 1 ($50 and $51). Because this routine is the same one that Applesoft uses to check line numbers, it has the same limitations, namely it is only good for line numbers up to and including 63999.

If you want to display lines greater than that, the JSR LINGET should be replaced by a JSR FRMNUM ($DD67), immediately followed by a JSR GETADR ($E752). You may recall, that we used these two routines in the previous program to input decimal numbers that were going to be converted to hexadecimal numbers.

Once the line number has been converted to hex and stored in LINNUM, another Applesoft ROM routine, FNDLIN ($D61A), is called (line 1670). FNDLIN will start at the beginning of the Applesoft program and search for the line number that is currently stored in LINNUM (and of course LINNUM + 1). If the line is found, its beginning address is stored in two page zero locations called LOWTR

and LOWTR + 1 ($9B and $9C). Also, if the number is found the carry bit is set. If the number is not found, the next highest line number, if there is one, is stored in LOWTR and the carry bit is cleared.

Upon returning from FNDLIN, the first thing the program does is to test the carry-bit to see if the line number was found (line 1680). If it was not found, the program branches to line 1940 where a message to the user is printed that rings the bell and tells him that no such line exists in the program. If the line does exist, the Y-register and memory location TEMP are both set to zero (lines 1690 and 1700) and the program jumps to a subroutine that prints out the two-byte address of the data that are going to be displayed on the next line of the video display (line 1710). This subroutine, which is called PRTADDR, starts on line 2050 and begins by printing a carriage return and than a space (lines 2050 to 2070). Next, the X-register is set up as a displayed byte counter (line 2080) and is used to permit the display of only eight bytes of data per line. Then the subroutine prints out the address that is stored in LOWTR and LOWTR + 1, high-order byte first (lines 2090 to 2120). Finally, a colon is printed out and the program returns to the caller via the RTS in the COUT routine (lines 2130 and 2140).

After printing out the starting memory address of the line of data to be displayed on the screen, a space is printed (lines 1720 and 1730) and eight bytes of data are printed. The byte to be printed is retrieved in line 1740 and checked to see if it is a zero in line 1750. If it is a zero, TEMP is tested to see if five or more bytes have already been printed (lines 1760 and 1770). The reason for this is that for line numbers below 255, the fourth byte, which is the high-order byte of the line number is set to zero. This is not the zero we wish to detect, but rather the zero that terminates the Applesoft program line.

If five or more bytes have been printed already, we know that this zero represents the end of the Applesoft line, so the program jumps to a routine (on line 1850), that prints out the zero, then prints out a carriage return (line 1870) and finally jumps to a routine in the F8 ROM called MONZ ($FF69) which leaves the user in the monitor mode (line 1880). If for some reason you wish to return to the program that called the APPLESOFT LINE FINDER instead of being left in the monitor, it is only necessary to replace the JMP MONZ in line 1880 with an RTS.

If less than five bytes have been printed, we know that this is not the end of the line and we print the zero out just as we would print any other byte (lines 1790 and 1800). Then the program jumps to a routine on line 2210 that increments the two-byte LOWTR pointer and also increments TEMP. After that, the X-register is decremented and tested to see if eight bytes have been printed already (lines 1820 and 1830). If not the program branches to line 1720 where the next byte is retrieved and printed. Otherwise, it branches to line 1710 where the address of the next byte to be displayed is printed. This process continues until the terminating zero of the Applesoft program line is encountered.

The subroutine located in lines 2300 to 2400 is our, by now familiar, message printing routine. Following this routine, on line 2480, is the text for the error message that says the line doesn't exist.


```
                1000 ************************************
                1010 ***                              ***
                1020 ***     APPLESOFT LINE FINDER    ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $2DA
                1130 *
                1140 *
                1150 * EQUATES
                1160 *
0008-           1170 TEMP     .EQ $8
0018-           1180 TXTPTR   .EQ $18
003C-           1190 A1L      .EQ $3C
009B-           1200 LOWTR    .EQ $9B
03F5-           1210 AMPERSD  .EQ $3F5
D61A-           1220 FNDLIN   .EQ $D61A
DA0C-           1230 LINGET   .EQ $DA0C
F941-           1240 PRNTAX   .EQ $F941
FC58-           1250 HOME     .EQ $FC58
FD8E-           1260 CROUT    .EQ $FD8E
FDDA-           1270 PRBYTE   .EQ $FDDA
FDED-           1280 COUT     .EQ $FDED
FF69-           1290 MONZ     .EQ $FF69
                1300 *
                1310 *
                1320 * This is where the program title is
                1330 * printed out and the ampersand (&) vector
                1340 * jump is set up.
                1350 *
02DA- 20 58 FC  1360          JSR HOME        Clear the screen.
02DD- A9 F4     1370          LDA #TEXT1      Get the address of the
02DF- A0 02     1380          LDY /TEXT1      text to be printed.
02E1- 20 A8 03  1390          JSR MSGPRT      Print it.
02E4- A2 4C     1400          LDX #$4C        Get a JMP op code and
02E6- A9 49     1410          LDA #START      the low and high bytes
02E8- A0 03     1420          LDY /START      of START's address and
02EA- 8E F5 03  1430          STX AMPERSD     store them in locations
02ED- 8D F6 03  1440          STA AMPERSD+1   $3F5, $3F6 and $3F7.
02F0- 8C F7 03  1450          STY AMPERSD+2
02F3- 60        1460          RTS
                1470 *
                1480 *
                1490 * This is the text for the title and
                1500 * copyright notice.
                1510 *
02F4- C1 D0 D0
02F7- CC C5 D3
02FA- CF C6 D4
02FD- A0 CC C9
0300- CE C5 A0
0303- C6 C9 CE
0306- C4 C5 D2  1520 TEXT1    .AS -"APPLESOFT LINE FINDER"
0309- 8D 8D     1530          .HS 8D8D
030B- C2 D9 A0
030E- CA D5 CC
0311- C5 D3 A0
0314- C8 AE A0
0317- C7 C9 CC
031A- C4 C5 D2  1540          .AS -"BY JULES H. GILDER"
031D- 8D        1550          .HS 8D
031E- C3 CF D0
0321- D9 D2 C9
0324- C7 C8 D4
0327- A0 A8 C3
032A- A9 A0 B1
032D- B9 B8 B2  1560          .AS  "COPYRIGHT (C) 1982"
```

```
0330- 8D          1570          .HS 8D
0331- C1 CC CC
0334- A0 D2 C9
0337- C7 C8 D4
033A- D3 A0 D2
033D- C5 D3 C5
0340- D2 D6 C5
0343- C4          1580          .AS -"ALL RIGHTS RESERVED"
0344- 8D 8D 8D
0347- 8D 00       1590          .HS 8D8D8D8D00
                  1600 *
                  1610 *
                  1620 * This part of the program is the main
                  1630 * loop.  It gets the line number, finds
                  1640 * it in memory and displays it in hex.
                  1650 *
0349- 20 0C DA    1660 START    JSR LINGET      Convert number after & to hex.
034C- 20 1A D6    1670          JSR FNDLIN      Put address of line in LOWTR.
034F- 90 2E       1680          BCC NOLINE      Line doesn't exist.
0351- A0 00       1690          LDY #$0         Zero the Y-register.
0353- 84 08       1700          STY TEMP        and TEMP.
0355- 20 86 03    1710 NXTLIN   JSR PRTADDR     Print address of line.
0358- A9 A0       1720 PRTSPC   LDA #$A0        Print a space.
035A- 20 ED FD    1730          JSR COUT
035D- B1 9B       1740          LDA (LOWTR),Y   Get the next byte in the line.
035F- D0 08       1750          BNE PRINTIT     If it's not zero, print it.
0361- A5 08       1760          LDA TEMP        It is zero, did we pass
0363- C9 05       1770          CMP #$5         the fifth byte?
0365- B0 0D       1780          BCS DONE        Yes, print it and end up.
0367- A9 00       1790          LDA #$0         No, print it and continue.
0369- 20 DA FD    1800 PRINTIT  JSR PRBYTE      Print byte in accumulator.
036C- 20 9F 03    1810          JSR INCR        Increment LOWTR and TEMP.
036F- CA          1820          DEX             Decrease X by one.
0370- F0 E3       1830          BEQ NXTLIN      X=0 start a new line.
0372- D0 E4       1840          BNE PRTSPC      Get and print next byte.
0374- A9 00       1850 DONE     LDA #$0         The last byte is a zero
0376- 20 DA FD    1860          JSR PRBYTE      so print it.
0379- 20 8E FD    1870          JSR CROUT       Print a carriage return.
037C- 4C 69 FF    1880          JMP MONZ        Jump to the monitor.
                  1890 *
                  1900 *
                  1910 * Tell the user the line he requested
                  1920 * does not exist.
                  1930 *
037F- A9 BE       1940 NOLINE   LDA #TEXT2      Point to text to be
0381- A0 03       1950          LDY /TEXT2      printed.
0383- 4C A8 03    1960          JMP MSGPRT      Print it.
                  1970 *
                  1980 *
                  1990 * This section of the program prints
                  2000 * out a carriage return, a space and then
                  2010 * the address in memory of the first byte
                  2020 * displayed on the line, followed by a
                  2030 * colon.
                  2040 *
0386- 20 8E FD    2050 PRTADDR  JSR CROUT       Print a carriage return.
0389- A9 A0       2060          LDA #$A0        Print out a space.
038B- 20 ED FD    2070          JSR COUT
038E- A2 08       2080          LDX #$8         Count 8 bytes per line.
0390- A5 9C       2090          LDA LOWTR+1     Print out the address of
0392- 20 DA FD    2100          JSR PRBYTE      the first byte on the
0395- A5 9B       2110          LDA LOWTR       line, high byte first.
0397- 20 DA FD    2120          JSR PRBYTE
039A- A9 BA       2130          LDA #$BA        Then print a colon.
039C- 4C ED FD    2140          JMP COUT
                  2150 *
                  2160 *
                  2170 * Here, the pointer to the contents of
                  2180 * the line is incremented.  Location
                  2190 * TEMP is incremented too.
                  2200 *
039F- E6 9B       2210 INCR     INC LOWTR
03A1- D0 02       2220          BNE INCTEMP
```



```
03A3- E6 9C       2230          INC LOWTR+1
03A5- E6 08       2240 INCTEMP  INC TEMP
03A7- 60          2250          RTS
                  2260 *
                  2270 *
                  2280 * This is the message printing routine.
                  2290 *
03A8- 85 18       2300 MSGPRT   STA TXTPTR
03AA- 84 19       2310          STY TXTPTR+1
03AC- A0 00       2320          LDY #$0
03AE- B1 18       2330 LOOP     LDA (TXTPTR),Y
03B0- F0 0B       2340          BEQ ENDPRT
03B2- 20 ED FD    2350          JSR COUT
03B5- E6 18       2360          INC TXTPTR
03B7- D0 F5       2370          BNE LOOP
03B9- E6 19       2380          INC TXTPTR+1
03BB- D0 F1       2390          BNE LOOP
03BD- 60          2400 ENDPRT   RTS
                  2410 *
                  2420 *
                  2430 * This is the text that tells the user
                  2440 * that the requested line doesn't exist
                  2450 * in the program.  A bell is also rung
                  2460 * to alert the user to the error.
                  2470 *
03BE- 8D          2480 TEXT2    .HS 8D
03BF- CE CF A0
03C2- D3 D5 C3
03C5- C8 A0 CC
03C8- C9 CE C5    2490          .AS -"NO SUCH LINE"
03CB- 87 8D 00    2500          .HS 878D00
```

Unlike most programs that use the ampersand, this one is meant to be used primarily from the immediate mode rather than being called from a running program. However, as I indicated earlier, if you want it to return to a program that called it, the change that has to be made is trivial.

To use APPLESOFT LINE FINDER, just type in an ampersand, followed by the line number like this, &10. This will cause line 10 of the current Applesoft program to be displayed on the screen in hexadecimal form and leave you in the monitor mode so that changes can be made to it. Since a colon is used to separate the address from the displayed data, it is only necessary to move your cursor up to the line that is going to be changed and copy everything with the right arrow key except those items that are going to be modified. It couldn't be simpler.

Appending two Applesoft programs together

Anyone who has done a considerable amount of BASIC programming has at one time or another had the need to combine two programs, or program segments together into one. There are several ways that this can be done. You can write one segment out as a text file and then EXEC it in after the second program has been loaded into memory, or you can use Apple's Renumber and Append program that comes of the DOS System Master diskette.

There are some disadvantages to these approaches. First, they both require that the user have a disk drive. Since most Apple owners have at least one drive that's not too bad, but it still leaves a small group of people without any way of combining two programs together. Another disadvantage of the EXEC approach, is that

you have to write a separate program to convert your Applesoft program into a text file. In addition, if it's a long program, the append operation becomes very time consuming. A disadvantage of using Apple's program, is that it must be loaded into memory first, because if you try to run it after your program is in memory, it will erase your program.

The answer to these problems is APPLESOFT APPEND, a short machine language program that sits in pages two and three and can be loaded and run at any time and does not require the presence of a disk drive.

The program starts off with a routine on lines 1390 to 1490 that prints out the title page and a 'READY.' message. It also sets up the ampersand jump locations to cause the program to jump to line 1700 when the ampersand key is pressed.

At line 1700, the program checks the character following the ampersand to see if it is an 'H'. If it is, the program branches to the routine that puts the first program on hold (line 1710), otherwise it checks to see if the character is an 'M'. It checks for the 'M' (lines 1730 and 1740) a little differently than it checked for the 'H', although we could have used the same technique. However, I wanted to introduce you to another useful routine in the Applesoft ROMs, called SYNCHR which is located at $DEC0.

SYNCHR is Applesoft's syntax character checking routine. What it does is check to see that TXTPTR, the pointer used by the CHRGET routine is pointing to the same character that is in the accumulator. If the characters match the routine returns without modifying the accumulator or TXTPTR and the program jumps to the routine that merges the two programs together (line 1740). However, if the two do not match, then it jumps to the routine in the ROMs that prints out the message SYNTAX ERROR and halts program execution.

The purpose of the BEGIN routine is to hide, or put on hold, the first Applesoft program. The first thing the routine does is to print out a message to the user telling him that the first program is on hold (lines 1790 to 1810). Now, after telling the user that it has already been done, the program goes about doing the things it must in order to put the program on hold and hide it from the Applesoft interpreter. The first thing it does is store the address of the beginning of the Applesoft program for retrieval later (lines 1820 to 1850).

To hide the program, it is necessary to adjust the two-byte start of program pointer, which is called TXTTAB, so that it points to the end of the program. This is exactly what is done in lines 1860 to 1930. As the first step in this process, the program jumps to a subroutine that checks to see how far past the program the end of program pointer is pointing.

There seems to be a bug in Applesoft that increases the end of program pointer by one if the program was written after a NEW was executed rather than after an FP was executed. To check this, try an experiment. While in the Applesoft mode type FP and press the carriage return. Now type CALL-151 and carriage return and finally type AF.B0. Your screen should look like this:

```
]FP
CALL-151

*AF.B0

00AF- 03
00B0- 08
```


Now get back into Applesoft by typing a Control-C, or 3D0G if DOS is active, and type NEW. Now type CALL-151 and AF.B0. This time, instead of getting 03 for AF, you get 04. This minor bug can cause havoc with an append program, as many people who have tried to write one have found out. The worst part is, if you don't know that it is caused by the differences between an FP and a NEW, it seems like a random bug and it is almost impossible to track down.

Now that we are aware of the bug, we can negate its affects. In line 1860, the program jumps to a subroutine called CHKEND (line 2050), that determines whether the program that is currently in memory (the first program) was entered after a NEW or an FP. The first thing this routine does is to set ENDBYTE equal to two, which is the value it should have if the program was entered after an FP (lines 2050 and 2060). Next the carry is set (line 2070) and a two-byte subtraction is performed, subtracting four from the end of program pointer (lines 2080 to 2130).

The value obtained from this subtraction is used as a pointer to pick up the next to the last byte of the last program line (line 2150). This byte should not be a zero. If the value retrieved is not zero, then the program was entered after an FP was executed and the value of ENDBYTE is okay. If it is zero, the end of program pointer was advanced by one, which means that the program was entered after a NEW. In this case, ENDBYTE must be incremented by one (line 2170), before returning to the calling routine.

Upon returning to the calling routine (back to line 1870) we now know how many bytes past the end of the program PRGEND is pointing, and thus know how much to subtract from it in order to get the correct starting location for the second program. In lines 1870 to 1930, the subtraction is performed, and the results are stored in the beginning of program pointer (TXTTAB). Finally, the program returns to the immediate mode either directly if DOS is not present, or via DOS if it is (lines 1940 to 1980).

There are still two routines we haven't gone over yet. One is the message printing routine (lines 2370 to 2450) which should be quite familiar by now and will not be discussed. The other is the RESET routine (lines 2250 to 2320). The RESET routine is reached only if an &M is entered (lines 1720 to 1740). This command should only be entered after the first program has been put on hold and the second program has been loaded. It can also be use to retrieve the first program if you change your mind and decide you don't want to append anything to it.

The first thing this routine does is to tell the user that the appending operation has been completed (lines 2250 to 2270). Then the start of program pointers for the first program, which were stored in TEMP and TEMP + 1, are restored to TXTTAB. The only thing left to do is to reset the line links (the next line pointers). This

is done by calling a routine in the Applesoft ROMs known as LINKSET ($D4F2). This routine does not return, but instead returns control to the immediate mode.

This program works very much like Apple's program. Once it is activated by loading it and doing a CALL 696, it is only necessary to load in your first program and then press &H to put it on hold. If you try to list the program at this point, it will have seemed to disappear. Don't get nervous it's still there and can be recalled by typing in &M. Try it and then type LIST. See? I told you there was nothing to worry about. If you did bring the program back by typing &M, let's type &H again to make it disappear once more. Next, load in your second program and type &M to merge the two programs together. That's all there is to it.

One thing you should pay attention to when using a program like APPLESOFT APPEND is that the line numbers of the second program are always greater than the line numbers of the first program. The only time you do not have to worry about this constraint is if there are no GOTOs, GOSUBs or IF... THEN <line number> statements in the second program. In that case the line numbers can overlap, but if they do, remember, you are not going to be able to edit or individually list the lines of the second program.

```
                      1000 ************************************
                      1010 ***                              ***
                      1020 ***       APPLESOFT APPEND       ***
                      1030 ***                              ***
                      1040 ***     COPYRIGHT (C) 1982 BY    ***
                      1050 ***        JULES H. GILDER       ***
                      1060 ***      ALL RIGHTS RESERVED     ***
                      1070 ***                              ***
                      1080 ************************************
                      1090 *
                      1100 *
                      1110 *
                      1120          .OR $2B8
                      1130 *
                      1140 *
                      1150 * EQUATES
                      1160 *
                      1170 *
0006-                 1180 TEMP     .EQ $6
0008-                 1190 TXTPTR   .EQ $8
0067-                 1200 TXTTAB   .EQ $67
00AF-                 1210 PRGEND   .EQ $AF
03D0-                 1220 WARMDOS  .EQ $3D0
03F5-                 1230 AMPERSD  .EQ $3F5
D4F2-                 1240 LINKSET  .EQ $D4F2
DEC0-                 1250 SYNCHR   .EQ $DEC0
E000-                 1260 BASIC    .EQ $E000
FC58-                 1270 HOME     .EQ $FC58
FDED-                 1280 COUT     .EQ $FDED
                      1290 *
                      1300 *
                      1310 * This section clears the screen and
                      1320 * prints out the title and copyright
                      1330 * notice.  It also notifies the user
                      1340 * that the first program has been put
                      1350 * on hold.  It then sets up the
                      1360 * ampersand jump vector to point to
                      1370 * this program.
                      1380 *
02B8- 20 58 FC        1390          JSR HOME        Clear the screen.
02BB- A9 D2           1400          LDA #TEXT1      Print out the title and
02BD- A0 02           1410          LDY /TEXT1      copyright notice.
02BF- 20 97 03        1420          JSR MSGPRT
```



```
02C2- A9 4C           1430          LDA #$4C        Set up the ampersand
02C4- 8D F5 03        1440          STA AMPERSD     jump vector to point
02C7- A9 31           1450          LDA #START      to START.
02C9- 8D F6 03        1460          STA AMPERSD+1
02CC- A9 03           1470          LDA /START
02CE- 8D F7 03        1480          STA AMPERSD+2
02D1- 60              1490          RTS
                      1500 *
                      1510 *
                      1520 * This is the title and copyright information
                      1530 * printed out the first time the
                      1540 * program is run.
                      1550 *
02D2- C1 D0 D0
02D5- CC C5 D3
02D8- CF C6 D4
02DB- A0 C1 D0
02DE- D0 C5 CE
02E1- C4 A0 D0
02E4- D2 CF C7
02E7- D2 C1 CD        1560 TEXT1    .AS -"APPLESOFT APPEND PROGRAM"
02EA- 8D 8D           1570          .HS 8D8D
02EC- C2 D9 A0
02EF- CA D5 CC
02F2- C5 D3 A0
02F5- C8 AE A0
02F8- C7 C9 CC
02FB- C4 C5 D2        1580          .AS -"BY JULES H. GILDER"
02FE- 8D              1590          .HS 8D
02FF- C3 CF D0
0302- D9 D2 C9
0305- C7 C8 D4
0308- A0 A8 C3
030B- A9 A0 B1
030E- B9 B8 B2        1600          .AS -"COPYRIGHT (C) 1982"
0311- 8D              1610          .HS 8D
0312- C1 CC CC
0315- A0 D2 C9
0318- C7 C8 D4
031B- D3 A0 D2
031E- C5 D3 C5
0321- D2 D6 C5
0324- C4              1620          .AS -"ALL RIGHTS RESERVED"
0325- 8D 8D 8D
0328- 8D              1630          .HS 8D8D8D8D
0329- D2 C5 C1
032C- C4 D9 AE        1640          .AS -"READY."
032F- 8D 00           1650          .HS 8D00
                      1660 *
                      1670 *
                      1680 *
                      1690 *
0331- C9 48           1700 START    CMP #$48        Is there an 'H' after &?
0333- F0 08           1710          BEQ BEGIN       Yes, put program on hold.
0335- A9 4D           1720          LDA #$4D        No, is it an 'M'?
0337- 20 C0 DE        1730          JSR SYNCHR
033A- 4C 85 03        1740          JMP RESET       Yes, merge the programs.
                      1750 *
                      1760 *
                      1770 * This section hides the first program.
                      1780 *
033D- A9 A8           1790 BEGIN    LDA #TEXT2      Tell the user a program
033F- A0 03           1800          LDY /TEXT2      is on hold.
0341- 20 97 03        1810          JSR MSGPRT
0344- A5 68           1820          LDA TXTTAB+1    Get the start of
0346- A4 67           1830          LDY TXTTAB      program pointers
0348- 85 07           1840          STA TEMP+1      and save them for
034A- 84 06           1850          STY TEMP        later.
034C- 20 69 03        1860          JSR CHKEND      Programmed after FP or NEW?
034F- A5 AF           1870          LDA PRGEND      Set the end of program
0351- 38              1880          SEC             pointer to the
0352- ED CF 03        1890          SBC ENDBYTE     beginning.
0355- 85 67           1900          STA TXTTAB
```

```
0357- A5 B0     1910          LDA PRGEND+1
0359- E9 00     1920          SBC #$0          Borrow if necessary.
035B- 85 68     1930          STA TXTTAB+1
035D- A9 D0     1940          LDA #WARMDOS     Return to a BASIC
035F- C9 4C     1950          CMP #$4C         warm start, through DOS
0361- D0 03     1960          BNE NODOS        if it is present.
0363- 4C D0 03  1970          JMP WARMDOS      DOS is present.
0366- 4C 03 E0  1980 NODOS    JMP BASIC+3      DOS is not present.
                1990 *
                2000 *
                2010 * This routine checks to see if the
                2020 * program was written after an FP
                2030 * (ENDBYTE=2) or after a NEW (ENDBYTE=3).
                2040 *
0369- A9 02     2050 CHKEND   LDA #$2          Set ENDBYTE to 2
036B- 8D CF 03  2060          STA ENDBYTE      for now.
036E- 38        2070          SEC              Prepare for subtraction
036F- A5 AF     2080          LDA PRGEND       Subtract 4 from the
0371- E9 04     2090          SBC #$4          end of program
0373- 85 67     2100          STA TXTTAB       pointer and save
0375- A5 B0     2110          LDA PRGEND+1     in TXTTAB.
0377- E9 00     2120          SBC #$0          Borrow if necessary.
0379- 85 68     2130          STA TXTTAB+1
037B- A0 00     2140          LDY #$0
037D- B1 67     2150          LDA (TXTTAB),Y   Get last byte of program.
037F- D0 03     2160          BNE CHKDONE      If not zero, ENDBYTE okay.
0381- EE CF 03  2170          INC ENDBYTE      Increase ENDBYTE by 1.
0384- 60        2180 CHKDONE  RTS              Return.
                2190 *
                2200 * This section resets the start of
                2210 * program pointers and then calls the
                2220 * line link fixing routine which then
                2230 * returns to Applesoft.
                2240 *
0385- A9 BB     2250 RESET    LDA #TEXT3       Point to APPEND
0387- A0 03     2260          LDY /TEXT3       COMPLETED message.
0389- 20 97 03  2270          JSR MSGPRT       Print it.
038C- A5 06     2280          LDA TEMP         Restore the low-order
038E- 85 67     2290          STA TXTTAB       byte of the pointer.
0390- A5 07     2300          LDA TEMP+1       Restore the high-order
0392- 85 68     2310          STA TXTTAB+1     byte.
0394- 4C F2 D4  2320          JMP LINKSET      Fix the line links.
                2330 *
                2340 *
                2350 * This is the message printing routine.
                2360 *
0397- 85 08     2370 MSGPRT   STA TXTPTR
0399- 84 09     2380          STY TXTPTR+1
039B- A0 00     2390          LDY #$0
039D- B1 08     2400 LOOP     LDA (TXTPTR),Y
039F- F0 06     2410          BEQ ENDPRT
03A1- 20 ED FD  2420          JSR COUT
03A4- C8        2430          INY
03A5- D0 F6     2440          BNE LOOP
03A7- 60        2450 ENDPRT   RTS
                2460 *
                2470 *
                2480 * These are the text messages printed out
                2490 * by the program.
                2500 *
03A8- 8D        2510 TEXT2    .HS 8D
03A9- D0 D2 CF
03AC- C7 D2 C1
03AF- CD A0 CF
03B2- CE A0 C8
03B5- CF CC C4
03B8- AE        2520          .AS -"PROGRAM ON HOLD."
03B9- 8D 00     2530          .HS 8D00
03BB- 8D        2540 TEXT3    .HS 8D
03BC- C1 D0 D0
03BF- C5 CE C4
03C2- A0 C3 CF
03C5- CD D0 CC
```



```
03C8- C5 D4 C5
03CB- C4 AE     2550          .AS -"APPEND COMPLETED."
03CD- 8D 00     2560          .HS 8D00
03CF- 00        2570 ENDBYTE  .HS 00
```

How to restore lost Applesoft programs

Has this ever happened to you? You spend three days working on that super-duper whiz-bang program then you accidentally hit reset and get thrown into the monitor mode. You type Control-B instead of the Control-C you intended and PUFF! Three days of work have vanished before your eyes. Or maybe you accidentally typed NEW before you saved the final version of your program. Again, long hours of work have disappeared. This can be frustrating for any programmer, but it doesn't have to be for you any more, because &RESTORE will bring back your vanished program as quickly as you can type in the command &RESTORE. And you don't have to worry about loading this program first. It sits in page three and can be loaded at any time, before or after you've lost your program. To use it just type CALL 768 or after it has been run once just type &RESTORE.

What makes a program like this possible is the fact that the designers of the Applesoft language wanted to have an efficient language and decided that it was not necessary to actually erase the contents of memory every time a NEW command was issued. Instead, they just changed the information stored in the end of program pointer and erased only two bytes of data. So the program is still in memory, it's just that Applesoft doesn't know where to look for it. By restoring the two bytes that were erased (the pointer to the second line of the Applesoft program), and searching through memory until the end of the program is found and restoring the PRGEND pointer, the program can be brought back to life, as if it were always there.

In this program, we will see how we can use the existing set of key words and give them new functions to perform. In this case, as you've already guessed, we're going to use the RESTORE command. This command will still perform its usual function without any problems. But, when it is preceded by another command, the ampersand (&), it takes on an entirely new task.

The &RESTORE program begins, on line 1410, by setting up the ampersand jump vector to point to START and after that jumps to START2 (line 1470), skipping the code that checks for the presence of the word RESTORE. At line 1620, which is the ampersand entry point, the program loads the token for the word RESTORE (which is $AE) into the accumulator and then jumps to the syntax character checking routine (SYNCHR) to see if that token matches the information following the ampersand. If it doesn't, the subroutine prints out SYNTAX ERROR and stops execution of the main program. If it matches, the main program falls into the START2 routine.

It is not at all necessary to use the RESTORE command, but I thought you'd like to see how to do it. If you prefer to use just the & as the command, simply eliminate lines 1470, 1620 and 1630 and rename the label on line 1640, START. Once at line

1640, the program clears the screen and prints out the program's title, copyright notice and the word READY, indicating to the user that the program has already been restored. While the program has not yet been restored, the task is accomplished so quickly, that the user never realizes it.

The actual program restoration begins on line 1780 where the start of program pointer, TXTTAB, is used to produce another pointer (lines 1780 to 1830), called POINTER, which will skip the first four bytes of the line (these consist of the next line pointer and the line number). The reason we want to skip these bytes is that ultimately we want to find the end of the first line which is terminated with a zero. However, any of the first four bytes can legitimately contain a zero, which could result in premature termination of this program.

After POINTER has been calculated and stored, the high-byte of the start of program pointer is still in the accumulator and it is stored as the high byte of the pointer to the second line in the program (line 1850).

Now that the high-byte of the next line link to the second line has been restored, we have to find out where the first line ends in memory so that we can restore the low-byte. The routine that does this, FINDEOL, begins in line 1870. In lines 1870 and 1880, the Y-register is incremented and the contents of the location pointed to by both POINTER and the offset of the Y-register, are checked to see if they are zero. If not, the process is repeated until they are. If they are, the Y-register is transferred to the accumulator (line 1900), the carry bit is cleared in preparation for adding two numbers (line 1910) and 5 is added to the accumulator (line 1920). The five includes the four bytes that were skipped at the beginning, plus an additional byte so that the pointer will point not to the last character of the Applesoft line, but one past it, where the next line actually begins. This number is stored in the low byte of the next line pointer (line 1940).

If the program were to stop at this point, you would be able to list the program and it would appear as if it had all been restored. It hasn't, because if you saved it out to tape or disk and then loaded it back in, you'd find you had nothing, even though you were able to list it, and also run it. The program can be saved at this point only by listing it to an EXEC file. The reason the program will not save out properly is that we have not adjusted the end of program pointer, PRGEND, to point to the end of the program. This is what is done, starting at line 2000, where TXTTAB, the start of program pointer, is loaded into POINTER (lines 2000 to 2030).

In lines 2040 and 2050, a flag called TESTBYT is set to zero. This is going to be used to help us determine when the end of the program has been reached. A loop to scan the program is set up starting at line 2060, where POINTER and the Y-register are used to determine the next location from which a byte will be loaded and tested to see if it is equal to zero. After the byte is loaded, and before the test is performed, the Y-register is incremented (line 2070) and a check is made to see if a memory page boundary has been crossed (e.g. did we go from an address in the $800 range to an addresses in the $900 range). If no page boundary was crossed (line 2080) the program branches to ZEROCHK, otherwise, the high-byte of POINTER is incremented by one.

ZEROCHK is where the byte in the accumulator is tested to see if it is a zero (line 2100). If it's not, the program branches back to line 2040 where TESTBYT is reset to zero, and then checks the next byte. If it is a zero, we have to determine if this is the end of an Applesoft line or the end of the program. To do this we check TESTBYT and see if it is equal to two (lines 2120 and 2130). If it is (line 2140), this is the end of the program and the program branches to a routine that stores all of the pointers. If it's not equal to two, we increment TESTBYT by 1 and go back to check the next byte. As you see, TESTBYT is used to determine how many consecutive zeros we have encountered. The end of the program is indicated by three consecutive zeros; one for the end of line marker and two instead of the next line pointer.

The EXIT routine is where all of the Applesoft pointers are adjusted. These include the end of program pointer (PRGEND), the start of variable storage (VARTAB), the start of array storage (ARYTAB) and the end of string storage (STREND). In line 2170, the Y-register is incremented by one because we want to point to one past the three consecutive zero bytes. The Y-register is then transferred to the accumulator (line 2180) and the high-byte of POINTER is incremented if a page boundary is crossed (lines 2190 and 2200). All of the appropriate zero page pointers are updated in lines 2210 to 2290.

```
                1000 ***************************************
                1010 ***                                 ***
                1020 ***            &RESTORE             ***
                1030 ***                                 ***
                1040 ***      COPYRIGHT (C) 1982 BY      ***
                1050 ***         JULES H. GILDER         ***
                1060 ***       ALL RIGHTS RESERVED       ***
                1070 ***                                 ***
                1080 ***************************************
                1090 *
                1100 *
                1110 *
                1120          .OR $300
                1130 *
                1140 *
                1150 * CONSTANTS
                1160 *
004C-           1170 JUMP     .EQ $4C           JMP op code
00AE-           1180 RESTORE  .EQ $AE           RESTORE token
                1190 *
                1200 *
                1210 * EQUATES
                1220 *
0006-           1230 POINTER  .EQ $6
0008-           1240 TESTBYT  .EQ $8
0067-           1250 TXTTAB   .EQ $67
0069-           1260 VARTAB   .EQ $69
006B-           1270 ARYTAB   .EQ $6B
006D-           1280 STREND   .EQ $6D
00AF-           1290 PRGEND   .EQ $AF
03F5-           1300 AMPERSD  .EQ $3F5
DEC0-           1310 SYNCHR   .EQ $DEC0
FC58-           1320 HOME     .EQ $FC58
FDED-           1330 COUT     .EQ $FDED
                1340 *
                1350 *
                1360 * This is where the ampersand jump
                1370 * vector is set up.  After set-up,
                1380 * a relative jump is made to the
                1390 * second entry point of the program.
```

```
                    1400 *
0300- A9 4C         1410          LDA #JUMP       Get the JMP
0302- 8D F5 03      1420          STA AMPERSD     op-code & store
0305- A9 12         1430          LDA #START      it and the
0307- 8D F6 03      1440          STA AMPERSD+1   address of the
030A- A9 03         1450          LDA /START      start of this
030C- 8D F7 03      1460          STA AMPERSD+2   program.
030F- 4C 17 03      1470          JMP START2      Go to START2.
                    1480 *
                    1490 *
                    1500 * There are two entry points to this
                    1510 * program.  One is via the &RESTORE
                    1520 * command (START) and one by a CALL 768
                    1530 * (START2).  At START, the program
                    1540 * looks at the information that follows
                    1550 * the & to see if it is the RESTORE
                    1560 * token.  This is done by SYNCHR. If
                    1570 * not RESTORE a  syntax error is
                    1580 * generated. Once  syntax has been
                    1590 * checked, the program title is
                    1600 * printed out.
                    1610 *
0312- A9 AE         1620 START    LDA #RESTORE    Does the RESTORE
0314- 20 C0 DE      1630          JSR SYNCHR      token follow the &?
0317- A0 00         1640 START2   LDY #$0         Yes, zero character pointer.
0319- 20 58 FC      1650          JSR HOME        Clear the screen.
031C- B9 7E 03      1660 LOOP1    LDA TEXT,Y      Get a character.
031F- F0 06         1670          BEQ NEXT        If done go to NEXT.
0321- 20 ED FD      1680          JSR COUT        Print a character.
0324- C8            1690          INY             Increment the pointer.
0325- D0 F5         1700          BNE LOOP1       Get more characters.
                    1710 *
                    1720 *
                    1730 * This section of program resets the
                    1740 * start of program pointers that are
                    1750 * wiped out when a NEW or Control-B
                    1760 * are entered.
                    1770 *
0327- A5 67         1780 NEXT     LDA TXTTAB      Get program start
0329- 18            1790          CLC             low byte. Calculate
032A- 69 03         1800          ADC #$3         and save the starting
032C- 85 06         1810          STA POINTER     line's low byte.
032E- A5 68         1820          LDA TXTTAB+1    Get program start
0330- 85 07         1830          STA POINTER+1   high byte and save it.
0332- A0 01         1840          LDY #$1         Save 2nd line's
0334- 91 67         1850          STA (TXTTAB),Y  high byte.
0336- 88            1860          DEY             Zero the Y-register.
0337- C8            1870 FINDEOL  INY             Look for the end
0338- B1 06         1880          LDA (POINTER),Y of line marker
033A- D0 FB         1890          BNE FINDEOL     Keep looking.
033C- 98            1900          TYA             Found end of line,
033D- 18            1910          CLC             find value of
033E- 69 05         1920          ADC #$5         program start
0340- A0 00         1930          LDY #$0         low byte and
0342- 91 67         1940          STA (TXTTAB),Y  restore it.
                    1950 *
                    1960 *
                    1970 * This part of the program resets the
                    1980 * end of program pointers.
                    1990 *
0344- A5 67         2000          LDA TXTTAB      Store start of
0346- 85 06         2010          STA POINTER     program pointers
0348- A5 68         2020          LDA TXTTAB+1    in POINTER for
034A- 85 07         2030          STA POINTER+1   future use.
034C- A9 00         2040 LOOP2    LDA #$0         Initialize end of
034E- 85 08         2050          STA TESTBYT     program test byte.
0350- B1 06         2060 LOOP3    LDA (POINTER),Y Start scanning
0352- C8            2070          INY             the program.
0353- D0 02         2080          BNE ZEROCHK     Page boundary?
0355- E6 07         2090          INC POINTER+1   Yes, increment the byte.
0357- C9 00         2100 ZEROCHK  CMP #$0         Does the accumulator 0?
0359- D0 F1         2110          BNE LOOP2       No, keep scanning.
035B- A5 08         2120          LDA TESTBYT     Yes, is it the
```



```
035D- C9 02         2130          CMP #2          end of the program?
035F- F0 04         2140          BEQ EXIT        Yes, finish up.
0361- E6 08         2150          INC TESTBYT     No, increment the test byte.
0363- D0 EB         2160          BNE LOOP3       Get the next byte.
0365- C8            2170 EXIT     INY             Adjust the byte
0366- 98            2180          TYA             count & see if
0367- D0 02         2190          BNE STORPTR     we have to increment
0369- E6 07         2200          INC POINTER+1   the high byte too.
036B- 85 69         2210 STORPTR  STA VARTAB       Store the low byte
036D- 85 6B         2220          STA ARYTAB      of the end of the program
036F- 85 6D         2230          STA STREND      in the appropriate
0371- 85 AF         2240          STA PRGEND      zero page locations.
0373- A5 07         2250          LDA POINTER+1   Store the high byte
0375- 85 6A         2260          STA VARTAB+1    of the end of the program
0377- 85 6C         2270          STA ARYTAB+1    in the appropriate
0379- 85 6E         2280          STA STREND+1    zero page
037B- 85 B0         2290          STA PRGEND+1    locations.
037D- 60            2300          RTS             Return.
                    2310 *
                    2320 *
                    2330 * This is where text for program title
                    2340 * and copyright notice are stored.
                    2350 *
037E- A6 D2 C5
0381- D3 D4 CF
0384- D2 C5         2360 TEXT     .AS -"&RESTORE"
0386- 8D 8D         2370          .HS 8D8D
0388- C2 D9 A0
038B- CA D5 CC
038E- C5 D3 A0
0391- C8 AE A0
0394- C7 C9 CC
0397- C4 C5 D2      2380          .AS -"BY JULES H. GILDER"
039A- 8D            2390          .HS 8D
039B- C3 CF D0
039E- D9 D2 C9
03A1- C7 C8 D4
03A4- A0 A8 C3
03A7- A9 A0 B1
03AA- B9 B8 B2      2400          .AS -"COPYRIGHT (C) 1982"
03AD- 8D            2410          .HS 8D
03AE- C1 CC CC
03B1- A0 D2 C9
03B4- C7 C8 D4
03B7- D3 A0 D2
03BA- C5 D3 C5
03BD- D2 D6 C5
03C0- C4            2420          .AS -"ALL RIGHTS RESERVED"
03C1- 8D 8D 8D      2430          .HS 8D8D8D
03C4- D2 C5 C1
03C7- C4 D9 AE      2440          .AS -"READY."
03CA- 8D 00         2450          .HS 8D00
```

Chapter 8

EXPANDING APPLESOFT BASIC

In the last chapter we saw how it was possible to use the ampersand (&), an Applesoft key word, to jump to a machine language program whenever we wanted to, even in the middle of an Applesoft program. Sometimes, however, it is desirable to not only jump to a machine language routine, but to take some variables along for the ride. This is particularly important if you want to expand the capabilities of the Applesoft language.

Just as there are several ways to jump from Applesoft to a machine language program, there are also several ways to pass variables. Single byte variables can be POKEd into place before the jump. So can double byte variables, but that starts to become a little cumbersome. The two major ways of passing variables, however, are by letting them follow the ampersand, or by using the USR command in Applesoft. In this chapter, we are going to look at both of these methods and go over two programs for each approach. In addition, we'll learn how to add special function keys to Applesoft.

The programs in this chapter are unusual in that they are designed to expand the capabilities of the Applesoft language. One of the nice features of Integer BASIC which never made it into Applesoft, was the ability to have computed commands.

For instance, wouldn't it be nice to be able to write GOTO ENTERINFO when you transfer control to a portion of the program that handles the input of data, instead of writing GOTO 200. Or perhaps you want to determine where to branch to depending on the data entered. Wouldn't it be nice to be able to write GOTO N*100, where N is the data entered. The convenience of computed commands can be had for the GOSUB and LIST keywords as well. Just how to implement these commands is demonstrated in the first program entitled, COMPUTED GOTO, GOSUB AND LIST.

Adding new commands to Applesoft

In this program, we're going to use the ampersand to define three new commands: &GOTO, &GOSUB and &LIST. To start off, the program sets up the ampersand jump locations on page three, so that they point to the part of the program that handles the computed functions (lines 1520 to 1570). After that, the program title is printed out along with the word READY to indicate to the user that the program is properly installed, and ready to use (lines 1580 to 1610).

When the Applesoft interpreter encounters an ampersand, it transfers control to the subroutine labelled START on line 1710. Here the program tests the character that follows the ampersand to see if it represents the GOTO, GOSUB or LIST tokens (lines 1710 to 1760). If it's none of these, the last test, which is performed by using the SYNCHR routine in the Applesoft ROMs, will return control to the immediate mode after printing a SYNTAX ERROR message. If it is either GOTO or GOSUB a branch is made to the appropriate routine. If it's LIST, the program returns from the SYNCHR routine and falls into the routine that handles the computed LIST statement.

The first thing that the routine for the computed LIST does, is a jump to a short routine that is the heart of this entire program (line 1820). This routine is generally used as a replacement for the LINGET ($DA0C) routine that is normally found in the listings for these commands. The LINGET routine, expects to find a digit following these commands and also expects that the number these digits form is no greater than 63999. If the number is larger, a syntax error is generated.

The two routines that are jumped to from EVLNM2 (line 2370), however, expect only to find a number, variable or expression and will accept any value between +65535 and -65535. The FRMEVL routine evaluates whatever is found after the token and puts the value that it calculates into the floating point accumulator (FAC), which is located on page zero from $9D to $A2. In this form, the number is not very useful, so we use another Applesoft ROM routine to convert the number in the FAC to a two-byte integer number that is stored in LINNUM ($50).

Once we have the number as a two-byte integer, we call another Applesoft routine (line 1830), known as FNDLIN ($D61A), to find out where the Applesoft program line is located in memory. When FNDLIN locates the line, it puts the address of the line in a two-byte pointer on page zero called LOWTR (which is at $9B). The address is then taken from the LOWTR pointer and stored in another set

of page zero locations until it is needed later (lines 1840 to 1870).

With the address of the first line to be listed safely stored, the program then checks to see if the next character following the number, variable or formula, is a zero (lines 1880 and 1890). If it is, the return address that was pushed on the stack when this routine was entered, is popped off and a jump is made to an entry point in the middle of Applesoft's LIST routine. As a result, only the one line is listed. The reason for this is the zero that followed the expression was an end of text marker and told the program that a single line was to be listed, and not a range of lines.

Understanding one of Applesoft's quirks

At this point, it is easy to explain one of the quirks of the Applesoft LIST command. Those of you who start their programs with line number zero may have encountered some problems when you tried to list line zero by typing LIST 0. Instead of listing that one line, it lists the entire program. The reason for this is that LINGET, which is used in Applesoft's LIST routine, returns a zero in LINNUM if no line number follows the LIST command. This is the same result that is returned for a zero following the command. Since the program is set up to list the entire program when LINNUM is zero, it is not possible to list just line zero with the Applesoft LIST command.

The culprit in Applesoft, is a short routine located between $D6CE and and $D6D8. Here, both the high and low bytes of LINNUM are ORed together. If the result is zero, and that will only happen if LINNUM is zero, $FF is loaded into both the high and the low bytes of LINNUM, effectively telling the program that the last line to be listed is line 65535. Our program overcomes this shortcoming by branching past this code to an entry point called NXTLST ($D6DA). Thus an &LIST0, will list only line zero, if it exists, or nothing if it doesn't.

Getting back to our program, if the character following the expression was not a zero, a check is made to see if the only legitimate character that is permitted there, a comma, is present (line 1930). If the character was not a comma, an error condition exists and the subroutine returns immediately, generating a syntax error message in the process.

If the character was a comma, that is a sign to the program, that a range of lines is to be listed and not merely a single line. Knowing that, the program looks for the second expression that will yield a number that represents the last line to be listed (line 1950). That number is stored in both bytes of LINNUM and then a check is made to see if the end of the command has been reached (line 1960). If the next character that is retrieved is not a zero, a syntax error message is generated. If it is, the location of the first line to be listed is restored to LOWTR and the program jumps to LIST2 ($D6CC) where the rest of Applesoft's LIST routine is used.

GOTOs and GOSUBs can be computed too

If the character that follows the ampersand is not a LIST token, but a GOTO token, then the program branches to line 2070, where a very short subroutine is

executed. First the program jumps to the EVLNUM routine to get the line number, and then it jumps to a secondary entry point in the GOTO routine, past the section of code that uses LINGET to determine which line is to be jumped to, and executes the command.

The computed GOSUB routine, on line 2130, is not much more complex. The GOSUB routine normally pushes information on the stack so the first thing the routine does, is to check and see if there's room on the stack for the required data. Here another Applesoft ROM routine is used, CHKMEM ($D3D6). CHKMEM is entered with a number in the accumulator that represents the number of items to be stored on the stack. CHKMEM assumes that all of the items to be stored will be two-bytes in length and thus doubles the value in the accumulator and uses that number to ascertain if there is enough room on the stack. Once we find out that there is room, the current value of TXTPTR and CURLIN (each two-byte variables) is pushed on the stack (lines 2150 to 2220). In addition, it is necessary to push the GOSUB token ($B0) on the stack as well. This is done in lines 2230 and 2240. As you can see, only five bytes of stack storage were really needed and not the six that were checked for.

With the stack properly conditioned, the program now does a subroutine jump to the computed GOTO routine (line 2250) to find which line is required and where it is located in memory. Then the program jumps to another Applesoft routine, NEWSTT ($D7D2) where the GOSUB is actually executed.

The remainder of the listing consists of the message printing routine (MSGPRT) and the text that it prints out.

```
                1000 ****************************************
                1010 ***                                  ***
                1020 *** COMPUTED GOTO, GOSUB AND LIST ***
                1030 ***                                  ***
                1040 ***     COPYRIGHT (C) 1982 BY        ***
                1050 ***        JULES H. GILDER           ***
                1060 ***      ALL RIGHTS RESERVED         ***
                1070 ***                                  ***
                1080 ****************************************
                1090 *
                1100 *
                1110          .OR $2D8
                1120 *
                1130 *
                1140 * CONSTANTS
                1150 *
002C-           1160 COMMA    .EQ $2C
004C-           1170 JUMP     .EQ $4C
00AB-           1180 GOTO     .EQ $AB
00B0-           1190 GOSUB    .EQ $B0
00BC-           1200 LIST     .EQ $BC
                1210 *
                1220 *
                1230 * EQUATES
                1240 *
0006-           1250 TEMP     .EQ $6
0008-           1260 PNTR     .EQ $8
0075-           1270 CURLIN   .EQ $75
0098-           1280 LOWTR    .EQ $9B
00B1-           1290 CHRGET   .EQ $B1
00B7-           1300 CHRGOT   .EQ $B7
00B8            1310 TXTPTR   .EQ $B8
03F5            1320 AMPER    .EQ $3F5
```

```
D3D6-                1330 CHKMEM .EQ $D3D6
D61A-                1340 FNDLIN .EQ $D61A
D6CC-                1350 LIST2  .EQ $D6CC
D6DA-                1360 NXTLST .EQ $D6DA
D7D2-                1370 NEWSTT .EQ $D7D2
D941-                1380 GOTO2  .EQ $D941
DD7B-                1390 FRMEVL .EQ $DD7B
DEC0-                1400 SYNCHR .EQ $DEC0
E752-                1410 GETADR .EQ $E752
FC58-                1420 HOME   .EQ $FC58
FDED-                1430 COUT   .EQ $FDED
                     1440 *
                     1450 *
                     1460 * This section of code sets up the ampersand
                     1470 * jump vector and prints out the program
                     1480 * title.  When the READY. prompt is
                     1490 * displayed, the user knows the program
                     1500 * is active and ready to use.
                     1510 *
02D8- A9 4C          1520         LDA #JUMP       Get a JMP op code and
02DA- 8D F5 03       1530         STA AMPER       store it at $3F5.
02DD- A9 F1          1540         LDA #START      Store the address of
02DF- 8D F6 03       1550         STA AMPER+1     the start of the
02E2- A9 02          1560         LDA /START      program at $3F6 and
02E4- 8D F7 03       1570         STA AMPER+2     $3F7.
02E7- 20 58 FC       1580         JSR HOME        Clear the screen.
02EA- A9 6C          1590         LDA #TEXT       Point to the text to
02EC- A0 03          1600         LDY /TEXT       be printed.
02EE- 4C 56 03       1610         JMP MSGPRT      Print it.
                     1620 *
                     1630 *
                     1640 * Here the character immediately following
                     1650 * the ampersand is checked to see if it
                     1660 * is the GOTO, GOSUB or LIST tokens.  If
                     1670 * it's none of these, a syntax error is
                     1680 * generated, otherwise control is turned
                     1690 * over to the appropriate subroutine.
                     1700 *
02F1- C9 AB          1710 START   CMP #GOTO       Is it GOTO?
02F3- F0 38          1720         BEQ CGOTO       Yes, do it.
02F5- C9 B0          1730         CMP #GOSUB      No, is it GOSUB?
02F7- F0 3A          1740         BEQ CGOSUB      Yes, do it.
02F9- A9 BC          1750         LDA #LIST       Is it LIST?
02FB- 20 C0 DE       1760         JSR SYNCHR      Syntax error if not.
                     1770 *
                     1780 *
                     1790 * This is the subroutine that handles the
                     1800 * computed LIST statement.
                     1810 *
02FE- 20 50 03       1820         JSR EVLNM2      Get the number after LIST.
0301- 20 1A D6       1830         JSR FNDLIN      Find first line in memory.
0304- A5 9B          1840         LDA LOWTR       Save location for
0306- 85 06          1850         STA TEMP        use later.  Get
0308- A5 9C          1860         LDA LOWTR+1     both high and low
030A- 85 07          1870         STA TEMP+1      bytes.
030C- 20 B7 00       1880         JSR CHRGOT      Get character after variable.
030F- D0 05          1890         BNE CHKCOM      Not zero, maybe comma.
0311- 68             1900         PLA             Pop return address
0312- 68             1910         PLA             off the stack.
0313- 4C DA D6       1920         JMP NXTLST      List just one line.
0316- C9 2C          1930 CHKCOM  CMP #COMMA      Was it a comma?
0318- D0 51          1940         BNE ENDPRT      No, return with error.
031A- 20 4D 03       1950         JSR EVLNUM      Yes, get next value.
031D- 20 B7 00       1960         JSR CHRGOT      Get character after it.
0320- D0 49          1970         BNE ENDPRT      Not zero, return with error.
0322- A5 06          1980         LDA TEMP        Restore location of
0324- 85 9B          1990         STA LOWTR       first line in listing
0326- A5 07          2000         LDA TEMP+1      range.
0328- 85 9C          2010         STA LOWTR+1
032A- 4C CC D6       2020         JMP LIST2       List range of lines.
                     2030 *
                     2040 *
                     2050 * This routine handles the computed GOTO.
```



```
                     2060 *
032D- 20 4D 03       2070 CGOTO   JSR EVLNUM      Get the line number.
0330- 4C 41 D9       2080         JMP GOTO2       Jump to it.
                     2090 *
                     2100 *
                     2110 * This routine handles the computed GOSUB.
                     2120 *
0333- A9 03          2130 CGOSUB  LDA #$3         Number of variables to stack.
0335- 20 D6 D3       2140         JSR CHKMEM      See if enough room on stack.
0338- A5 B9          2150         LDA TXTPTR+1    Save TXTPTR on the stack.
033A- 48             2160         PHA
033B- A5 B8          2170         LDA TXTPTR
033D- 48             2180         PHA
033E- A5 76          2190         LDA CURLIN+1    Save the current line
0340- 48             2200         PHA             number on the stack.
0341- A5 75          2210         LDA CURLIN
0343- 48             2220         PHA
0344- A9 B0          2230         LDA #GOSUB      Save the GOSUB token
0346- 48             2240         PHA             on the stack.
0347- 20 2D 03       2250         JSR CGOTO       Use CGOTO to find the line.
034A- 4C D2 D7       2260         JMP NEWSTT      Execute the line.
                     2270 *
                     2280 *
                     2290 * This routine is the heart of the program.
                     2300 * It evaluates the number, variable or
                     2310 * formula that follows the appropriate
                     2320 * token and converts the number into a
                     2330 * two-byte integer that is stored in
                     2340 * LINNUM and LINNUM+1 ($50 and $51).
                     2350 *
034D- 20 B1 00       2360 EVLNUM JSR CHRGET       Set TXTPTR to next character.
0350- 20 7B DD       2370 EVLNM2 JSR FRMEVL       Evaluate formula.
0353- 4C 52 E7       2380         JMP GETADR      Convert to integer and store.
                     2390 *
                     2400 *
                     2410 * This is the message printing subroutine.
                     2420 *
0356- 85 08          2430 MSGPRT STA PNTR
0358- 84 09          2440         STY PNTR+1
035A- A0 00          2450         LDY #$0
035C- B1 08          2460 LOOP    LDA (PNTR),Y
035E- F0 0B          2470         BEQ ENDPRT
0360- 20 ED FD       2480         JSR COUT
0363- E6 08          2490         INC PNTR
0365- D0 F5          2500         BNE LOOP
0367- E6 09          2510         INC PNTR+1
0369- D0 F1          2520         BNE LOOP
036B- 60             2530 ENDPRT RTS
                     2540 *
                     2550 *
                     2560 * This is the text printed out by this program.
                     2570 *
036C- C3 CF CD
036F- D0 D5 D4
0372- C5 C4 A0
0375- C7 CF D4
0378- CF AC A0
037B- C7 CF D3
037E- D5 C2 A0
0381- C1 CE C4
0384- A0 CC C9
0387- D3 D4          2580 TEXT    .AS -"COMPUTED GOTO, GOSUB AND LIST"
0389- 8D 8D          2590         .HS 8D8D
038B- C2 D9 A0
038E- CA D5 CC
0391- C5 D3 A0
0394- C8 AE A0
0397- C7 C9 CC
039A- C4 C5 D2       2600         .AS -"BY JULES H. GILDER"
039D- 8D             2610         .HS 8D
039E- C3 CF D0
03A1- D9 D2 C9
03A4- C7 C8 D4
```

```
03A7- A0 A8 C3
03AA- A9 A0 B1
03AD- B9 B8 B2 2620          .AS -"COPYRIGHT (C) 1982"
03B0- 8D       2630          .HS 8D
03B1- C1 CC CC
03B4- A0 D2 C9
03B7- C7 C8 D4
03BA- D3 A0 D2
03BD- C5 D3 C5
03C0- D2 D6 C5
03C3- C4       2640          .AS -"ALL RIGHTS RESERVED"
03C4- 8D 8D 8D 2650          .HS 8D8D8D
03C7- D2 C5 C1
03CA- C4 D9 AE 2660          .AS -"READY."
03CD- 8D 8D 00 2670          .HS 8D8D00
```

To use the program, it should be loaded and then activated with a CALL 728. The format to be used for the commands is: &LIST X, &LIST X,Y, &GOTO X and &GOSUB X. The presence or absence of spaces is irrelevant and X and Y can be numbers, variables or mathematical formulas.

POKEing two bytes at a time

If you've done some Applesoft programming, I'm sure you've had occasion to store information in memory that had a value larger than 255. Since Applesoft only allows you to POKE quantities up to 255 into any memory location, you generally have to convert the value to two bytes and POKE each in separately. The code to do the job would look something like this:

```
10 B = 32767
20 Y = INT (B/256)
30 X = B - Y * 256
40 POKE A,X : POKE A + 1,Y
```

where B is the number to be stored and X is the low byte and Y is the high byte. That's an awful lot of program code for one simple operation. Besides, how many of you would remember exactly what the formulas are for X and Y? And how much time would be lost while you tried to figure them out? Wouldn't it be a lot more convenient to write something like this:

```
10 B = 32767
20 &POKE A,B
```

to accomplish the same task? Sure it would, and that's what prompted me to write the DOUBLE BYTE POKE program.

This program starts out the same way most programs that use the ampersand do, it sets up the ampersand jump locations (lines 1410 to 1460) so the computer will jump directly to the appropriate place in the program when the ampersand is encountered. Then the screen is cleared and the program title is printed out (lines 1470 to 1500).

The start of the routine that actually does the processing is on line 1530, where

the POKE token is loaded into the accumulator in preparation for a subroutine jump to SYNCHR (line 1540). As was mentioned in earlier programs, SYNCHR compares the accumulator with what is at the TXTPTR. If they match, the program returns to the caller and proceeds. If not, a syntax error is generated and program execution is halted.

In line 1550, the expression immediately following the POKE token is evaluated, and the results are placed in the floating point accumulator. From there, GETADR (line 1560) converts the number to a two-byte integer and stores the result in LINNUM as well as another set of page zero locations, called PNTR (lines 1570 to 1600). The syntax for the POKE statement requires that a comma follow the first expression so a check for a comma is made in line 1610 by jumping to an Applesoft routine, CHKCOM, at $DEBE.

If the comma is not present, a syntax error is generated and program execution stops. Otherwise, we get to the heart of the program. Line 1610 once again uses the FRMEVL routine to get the value of the expression that follows the comma. And once again GETADR is used to put it into a form that we can use (line 1630). Now that we have two bytes that represent the first storage location, and two bytes that represent the data to be stored, it is a simple matter to store both bytes of data in their appropriate locations. This is done in lines 1640 to 1690.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***       DOUBLE BYTE POKE       ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110          .OR $300
                1120 *
                1130 *
                1140 * CONSTANTS
                1150 *
004C-           1160 JUMP     .EQ $4C
00B9-           1170 POKE     .EQ $B9
                1180 *
                1190 *
                1200 * EQUATES
                1210 *
0006-           1220 PNTR     .EQ $6
0050-           1230 LINNUM   .EQ $50
00B1-           1240 CHRGET   .EQ $B1
00B7-           1250 CHRGOT   .EQ $B7
03F5-           1260 AMPER    .EQ $3F5
DD7B-           1270 FRMEVL   .EQ $DD7B
DEBE-           1280 CHKCOM   .EQ $DEBE
DEC0-           1290 SYNCHR   .EQ $DEC0
E752-           1300 GETADR   .EQ $E752
FC58-           1310 HOME     .EQ $FC58
FDED-           1320 COUT     .EQ $FDED
                1330 *
                1340 *
                1350 * This section of code sets up the ampersand
                1360 * jump vector and prints out the program
                1370 * title.  When the READY. prompt is
                1380 * displayed, the user knows the program
```

```
                    1390 * is active and ready to use.
                    1400 *
0300- A9 4C         1410         LDA #JUMP        Get a JMP op code and
0302- 8D F5 03      1420         STA AMPER        store it at $3F5.
0305- A9 19         1430         LDA #START       Store the address of
0307- 8D F6 03      1440         STA AMPER+1      the start of the
030A- A9 03         1450         LDA /START       program at $3F6 and
030C- 8D F7 03      1460         STA AMPER+2      $3F7.
030F- 20 58 FC      1470         JSR HOME         Clear the screen.
0312- A9 58         1480         LDA #TEXT        Point to the text to
0314- A0 03         1490         LDY /TEXT        be printed.
0316- 4C 42 03      1500         JMP MSGPRT       Print it.
                    1510 *
                    1520 *
0319- A9 B9         1530 START   LDA #POKE        See if POKE follows
031B- 20 C0 DE      1540         JSR SYNCHR       the ampersand.
031E- 20 7B DD      1550         JSR FRMEVL       Evaluate formula.
0321- 20 52 E7      1560         JSR GETADR       Convert to integer
0324- A5 50         1570         LDA LINNUM       Store address of
0326- 85 06         1580         STA PNTR         POKE in PNTR.
0328- A5 51         1590         LDA LINNUM+1
032A- 85 07         1600         STA PNTR+1
032C- 20 BE DE      1610         JSR CHKCOM       Check for a comma.
032F- 20 7B DD      1620         JSR FRMEVL       Evaluate formula.
0332- 20 52 E7      1630         JSR GETADR       Convert to integer.
0335- A0 00         1640         LDY #$0          Zero offset.
0337- B9 50 00      1650 LOOP1   LDA LINNUM,Y     Get value to be POKEd.
033A- 91 06         1660         STA (PNTR),Y     POKE it.
033C- C8            1670         INY              Increment offset.
033D- C0 02         1680         CPY #$2          Done yet?
033F- D0 F6         1690         BNE LOOP1        No, get next value.
0341- 60            1700         RTS              Yes, return.
                    1710 *
                    1720 *
                    1730 * This is the message printing subroutine.
                    1740 *
0342- 85 06         1750 MSGPRT  STA PNTR
0344- 84 07         1760         STY PNTR+1
0346- A0 00         1770         LDY #$0
0348- B1 06         1780 LOOP2   LDA (PNTR),Y
034A- F0 0B         1790         BEQ ENDPRT
034C- 20 ED FD      1800         JSR COUT
034F- E6 06         1810         INC PNTR
0351- D0 F5         1820         BNE LOOP2
0353- E6 07         1830         INC PNTR+1
0355- D0 F1         1840         BNE LOOP2
0357- 60            1850 ENDPRT  RTS
                    1860 *
                    1870 *
                    1880 * This is the text printed out by this program.
                    1890 *
0358- C4 CF D5
035B- C2 CC C5
035E- A0 C2 D9
0361- D4 C5 A0
0364- D0 CF CB
0367- C5            1900 TEXT    .AS -"DOUBLE BYTE POKE"
0368- 8D 8D         1910         .HS 8D8D
036A- C2 D9 A0
036D- CA D5 CC
0370- C5 D3 A0
0373- C8 AE A0
0376- C7 C9 CC
0379- C4 C5 D2      1920         .AS -"BY JULES H. GILDER"
037C- 8D            1930         .HS 8D
037D- C3 CF D0
0380- D9 D2 C9
0383- C7 C8 D4
0386- A0 A8 C3
0389- A9 A0 B1
038C- B9 B8 B2      1940         .AS -"COPYRIGHT (C) 1982"
038F- 8D            1950         .HS 8D
0390- C1 CC CC
```



```
0393- A0 D2 C9
0396- C7 C8 D4
0399- D3 A0 D2
039C- C5 D3 C5
039F- D2 D6 C5
03A2- C4            1960         .AS -"ALL RIGHTS RESERVED"
03A3- 8D 8D 8D      1970         .HS 8D8D8D
03A6- D2 C5 C1
03A9- C4 D9 AE      1980         .AS -"READY."
03AC- 8D 8D 00      1990         .HS 8D8D00
```

Taking a double PEEK at memory

Now that we've learned how to store information in memory two bytes at a time, it sure would be handy to be able to retrieve it the same way. With this program, we are going to look at another way of passing data between machine language programs and Applesoft programs. We're going to use the USR (X) function of Applesoft.

Once again, the task we wish to perform — a double byte peek — can be done in Applesoft. The following line of Applesoft code shows you how:

```
10 X = PEEK (A) + 256 * PEEK (A + 1)
```

Now I'll admit that this is not as difficult or complicated as what was required for the POKE statement, but it sure would be a whole lot easier to simply write:

```
10 X = USR (A)
```

to get the same result. The USR function in Applesoft has three characteristics about it. First, when it is encountered, the value of the expression that is within the parentheses is placed in the floating point accumulator. Second, the computer automatically jumps to location $A on page zero. There, in locations $A through $C, Applesoft expects to find a jump op code and the address of the machine language program that will do the processing. Thus, the first thing that our program should do, is to set up the USR jump locations (lines 1330 to 1380). When this is done, the program title and READY prompt are printed out (lines 1390 to 1420). The third, and final, thing that the USR function does is it passes a numerical value back to Applesoft through the floating point accumulator.

The routine that does the processing, whose address is stored in locations $B and $C, begins on line 1490. Here the GETADR routine is used to retrieve the value that has already been stored in the floating point accumulator. GETADR puts the address of the location of interest into LINNUM as two consecutive bytes, low-byte first, so LINNUM can be used as a pointer to the data we need.

Lines 1500 to 1550 retrieve the data we're interested in and store it in the floating point accumulator. Line 1560 sets the Carry bit so the routines that process the data that is in the FAC will know that we are interested only in positive numbers. If this were not done, numbers greater than 32767 would come back as negative numbers.

Next, the exponent of the floating point number is set to 216 by loading a $90 into the X-register and a jump is made to FLOAT2 (lines 1570 and 1580), where the number is processed so that it can be handed back to Applesoft.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***       DOUBLE BYTE PEEK       ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110         .OR $300
                1120 *
                1130 *
                1140 * EQUATES
                1150 *
0006-           1160 PNTR    .EQ $6
0008-           1170 TEMP    .EQ $8
000A-           1180 USR     .EQ $A
0050-           1190 LINNUM  .EQ $50
009D-           1200 FAC     .EQ $9D
E752-           1210 GETADR  .EQ $E752
EBA0-           1220 FLOAT2  .EQ $EBA0
FC58-           1230 HOME    .EQ $FC58
FDED-           1240 COUT    .EQ $FDED
                1250 *
                1260 *
                1270 * This section of code sets up the USR
                1280 * jump vector and prints out the program
                1290 * title.  When the READY. prompt is
                1300 * displayed, the user knows the program
                1310 * is active and ready to use.
                1320 *
0300- A9 4C     1330         LDA #$4C        Get a JMP op code and
0302- 85 0A     1340         STA USR         store it at $A.
0304- A9 16     1350         LDA #START      Store the address of
0306- 85 0B     1360         STA USR+1       the start of the
0308- A9 03     1370         LDA /START      program at $B and $C.
030A- 85 0C     1380         STA USR+2
030C- 20 58 FC  1390         JSR HOME        Clear the screen.
030F- A9 40     1400         LDA #TEXT       Point to the text to
0311- A0 03     1410         LDY /TEXT       be printed.
0313- 4C 2A 03  1420         JMP MSGPRT      Print it.
                1430 *
                1440 *
                1450 * This is the routine that gets the
                1460 * address pair to be looked at and
                1470 * retrieves the information stored their.
                1480 *
0316- 20 52 E7  1490 START   JSR GETADR      Convert to integer
0319- A0 00     1500         LDY #$0         Zero offset.
031B- B1 50     1510         LDA (LINNUM),Y  Get low byte and
031D- 85 9F     1520         STA FAC+2       store it in FAC.
031F- C8        1530         INY
0320- B1 50     1540         LDA (LINNUM),Y  Get high byte and
0322- 85 9E     1550         STA FAC+1       store it in FAC.
0324- 38        1560         SEC             Set for positive numbers only.
0325- A2 90     1570         LDX #$90        Set up X for FLOAT2.
0327- 4C A0 EB  1580         JMP FLOAT2      Convert to floating point.
                1590 *
                1600 *
                1610 * This is the message printing subroutine.
                1620 *
032A- 85 06     1630 MSGPRT  STA PNTR
032C- 84 07     1640         STY PNTR+1
032E- A0 00     1650         LDY #$0
```



```
0330- B1 06     1660 LOOP2   LDA (PNTR),Y
0332- F0 0B     1670         BEQ ENDPRT
0334- 20 ED FD  1680         JSR COUT
0337- E6 06     1690         INC PNTR
0339- D0 F5     1700         BNE LOOP2
033B- E6 07     1710         INC PNTR+1
033D- D0 F1     1720         BNE LOOP2
033F- 60        1730 ENDPRT  RTS
                1740 *
                1750 *
                1760 * This is the text printed out by this program.
                1770 *
0340- C4 CF D5
0343- C2 CC C5
0346- A0 C2 D9
0349- D4 C5 A0
034C- D0 C5 C5
034F- CB        1780 TEXT    .AS -"DOUBLE BYTE PEEK"
0350- 8D 8D     1790         .HS 8D8D
0352- C2 D9 A0
0355- CA D5 CC
0358- C5 D3 A0
035B- C8 AE A0
035E- C7 C9 CC
0361- C4 C5 D2  1800         .AS -"BY JULES H. GILDER"
0364- 8D        1810         .HS 8D
0365- C3 CF D0
0368- D9 D2 C9
036B- C7 C8 D4
036E- A0 A8 C3
0371- A9 A0 B1
0374- B9 B8 B2  1820         .AS -"COPYRIGHT (C) 1982"
0377- 8D        1830         .HS 8D
0378- C1 CC CC
037B- A0 D2 C9
037E- C7 C8 D4
0381- D3 A0 D2
0384- C5 D3 C5
0387- D2 D6 C5
038A- C4        1840         .AS -"ALL RIGHTS RESERVED"
038B- 8D 8D 8D  1850         .HS 8D8D8D
038E- D2 C5 C1
0391- C4 D9 AE  1860         .AS -"READY."
0394- 8D 8D 00  1870         .HS 8D8D00
```

The DOUBLE BYTE PEEK program is activated by BLOADing it and then doing a CALL 768. This just sets up the appropriate pointers and returns control to the calling program or mode. The syntax for using this program, as was illustrated earlier, is: X = USR (A) or PRINT USR (A), where A can be a number, variable or mathematical formula.

Running two Applesoft programs in memory together

By combining the use of the ampersand and the USR function with a little knowledge about how Applesoft stores programs in memory, it is possible to write an assembly language program that will enable two Applesoft programs to be stored in memory at the same time, with direct access to either one. The programs can be treated as completely separate entities, or they can share variables between them. You can even have one program call and execute the other, leaving all variables intact.

When the APPLESOFT PROGRAM SHARER program is run, by BLOADing it and doing a CALL 37888, the first thing the program does is to protect itself from

being wiped out by the strings of a running Applesoft program. Because of the length of the program, it cannot sit on page three and must therefore be located in high memory. I have placed it just below DOS at $9400. Thus, the first thing that the program does is to lower HIMEM from its current value, which is assumed to be $9600, to $9400 (lines 1390 and 1400). The next thing that is done is to set up the jump locations for both the ampersand and the USR commands (lines 1410 to 1510). Finally, the program title and instructions telling the user to load the first program, are displayed (lines 1520 to 1570.

This program makes good use of the fact that where the computer jumps to when the ampersand is encountered, depends on what is in locations $3F6 and $3F7. After each phase of the program, it updates the values stored there and shows you a way of using the ampersand for multiple routines without the need of passing variables via the ampersand. When the user was told to load the first program, he was also told to press the '&' key when he finished. When that is done, the computer jumps to line 1650, where a little manipulation of Applesoft pointers takes place.

In lines 1650 to 1680, TXTTAB ($67 and $68) the pointer that indicates where an Applesoft program starts, is stored in a pair of page zero locations called BEGIN1, for use later on. Next we find the end of the program so that we can hide the program from the Applesoft interpreter. We do this by getting the end of program pointer (PRGEND), and storing it in the beginning of program pointer (TXTTAB). At the same time, this information is also stored in another pair of page zero locations called BEGIN2 (lines 1690 to 1740).

Once the pointers have been changed, the program sets the ampersand jump locations to point to the next section of code to be executed (lines 1750 to 1780). Then the user is told to load the second program and press the ampersand key when it has been done (lines 1790 to 1830). Before returning control to the user, so he can load in the second program, the program marks the beginning of the second Applesoft program by storing a zero in the location that immediately precedes the start of the second program. This location is found by subtracting one from PRGEND (lines 1840 to 1860), which points to the end of the first program and the beginning of the second program. Next, since the accumulator still contains a zero that was placed there in the message printing routine, we transfer the accumulator to the Y-register (line 1870) to set the offset to zero and then store the zero in the accumulator in the location that precedes the start of the program (line 1880).

Normally it is not necessary to do this, because this location will contain the last of the three zero bytes that mark the end of the first program. However, if this program is called after a NEW has been executed, as you will recall from our earlier discussions, the normal end of program pointer is incremented by one byte, and thus the byte we're interested is no longer a zero. So, instead of testing for a NEW or an FP, as we did with the Applesoft Append program, we just store the zero there all the time. After the zero has been stored, the program jumps to the Applesoft NEW routine ($D649) and sets up all of the Applesoft pointers and registers so that the new program can be loaded without problems (line 1890).

After the second program has been loaded and the ampersand key has been pressed, the program jumps to line 1970 where the user is told (lines 1970 to 2010) that everything has been done and that now all that he has to do to switch between the two Applesoft programs is to press the ampersand key. Next, the ampersand jump locations are updated once again so that they point to another routine, this time to SWITCH which begins on line 2120.

As the name implies, SWITCH is the routine that switches between the two Applesoft programs. Basically, it sets up the ampersand key as a toggle between the two programs, so that no matter what program is currently available, when the ampersand key is pressed, the other one becomes usable. Upon entering this routine, BEGIN1 always contains the starting address in memory, of the next program to be made available. In lines 2120 to 2170, the address in BEGIN1 is placed in Applesoft's start of program pointer (TXTTAB) and also stored on the stack temporarily. Next, in lines 2180 to 2210, the address that was in BEGIN2 is moved to BEGIN1 so that the program it represents will be the next one to be loaded. Then, the address of the first program is retrieved from the stack and stored in BEGIN2 (lines 2220 to 2250).

How the two programs interact

If you've been paying close attention, you may have noticed that while we've been doing a lot of work with the beginning of program pointers, we've done virtually nothing with the end of program pointers. A logical question that may arise in your mind is, "Since the end of program pointer is pointing to the end of the second program (you did realize that didn't you), when the beginning of program pointer points to the first program, won't we be able to list both programs together, as if they had been append to each other?"

That's a good question, and while the programs do follow each other in memory, they have not been appended to one another. However, if you save the program out to tape or disk while the first program is being pointed to, both programs will be saved, but still only one will list. The reason for this is simple. The save routines work only with the start and end of program pointers, which at this point in time point to the start of the first program and the end of the second program. The LIST and RUN commands, however, only use the start of program pointer. To find the end of the program for these commands, Applesoft does not rely on the end of program pointer, but rather uses an end of program marker. This marker consists of three consecutive memory locations that each contain a zero. This marker is made up of the zero that terminates the last line of the program, and two zeros that are stored where the next line pointer would normally be stored.

In the Applesoft Append program, we set the pointer to the beginning of the second program to the location where the two zeros are normally stored. In this program, the second program pointer, points past these zeros, leaving them intact, resulting in the separation of the two programs.

Letting one program call the other

If you want one program to call the other automatically, with the variables still intact, this can be done with the USR function. Simply use a statement like the following one:

10 X = USR (A)

where A is a number, variable or mathematical expression that represents the line number you want the second program to start executing with.

The routine that handles the USR interface starts on line 2330 of the program listing, and is called USRGOTO. This short routine first does a JSR to SWITCH, where the pointers are adjusted so that the second program becomes active. Then the line number at which execution is to begin is retrieved from the floating point accumulator and placed, as a two-byte integer, into LINNUM (line 2340). Finally, a jump is made to Applesoft's GOTO routine (line 2350), which goes to the line number stored in LINNUM. This causes the second program to start executing without resetting the values of the variables.

The subroutine ENDMSG, which begins on line 2420, is used several times to print out the phrase, "AND PRESS THE '&' KEY." A second entry point, ENDMSG2, is use to print out only a part of the phrase. This is done by setting the accumulator to the value of the starting address before jumping to it. It is used here to cut off the word "AND". By doing this, some memory space was saved, because extra text was not required.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    APPLESOFT PROGRAM SHARER   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***        JULES H. GILDER       ***
                1060 ***      ALL RIGHTS RESERVED     ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110          .OR $9400
                1120          .TA $800
                1130 *
                1140 * EQUATES
                1150 *
0006-           1160 BEGIN1   .EQ $6
0008-           1170 BEGIN2   .EQ $8
000A-           1180 USR      .EQ $A
0018-           1190 TXTPTR   .EQ $18
0050-           1200 LINNUM   .EQ $50
0067-           1210 TXTTAB   .EQ $67
0073-           1220 HIMEM    .EQ $73
00AF-           1230 PRGEND   .EQ $AF
03F5-           1240 AMPERSD  .EQ $3F5
D649-           1250 NEW      .EQ $D649
D944-           1260 GOTO     .EQ $D944
E752-           1270 GETADR   .EQ $E752
FC58-           1280 HOME     .EQ $FC58
FDED-           1290 COUT     .EQ $FDED
                1300 *
                1310 *
```



```
                1320 * This section initializes the program
                1330 * by lowering HIMEM to protect the program
                1340 * and setting up the ampersand (&) and
                1350 * USR jump vectors.  It then prints out
                1360 * the title and the user is told to load
                1370 * in the first program.
                1380 *
9400- A9 94     1390          LDA #$94        Lower HIMEM to $9400
9402- 85 74     1400          STA HIMEM+1     to protect this program.
9404- A9 4C     1410          LDA #$4C        Set up the & and USR
9406- A0 2C     1420          LDY #NXTPROG    jump vectors to
9408- A2 94     1430          LDX /NXTPROG    point to the appropriate
940A- 85 0A     1440          STA USR         places in memory.
940C- 8D F5 03  1450          STA AMPERSD
940F- 8C F6 03  1460          STY AMPERSD+1
9412- 8E F7 03  1470          STX AMPERSD+2
9415- A9 92     1480          LDA #USRGOTO
9417- A0 94     1490          LDY /USRGOTO
9419- 85 0B     1500          STA USR+1
941B- 84 0C     1510          STY USR+2
941D- 20 58 FC  1520          JSR HOME        Clear the screen.
9420- A9 BB     1530          LDA #TEXT1      Point to the message
9422- A0 94     1540          LDY /TEXT1      to be printed.
9424- 20 A5 94  1550          JSR MSGPRT      Print it.
9427- A9 B1     1560          LDA #$B1        Tell user which program.
9429- 4C 9B 94  1570          JMP ENDMSG      Finish message
                1580 *
                1590 *
                1600 * Here the first program that was loaded
                1610 * in is hidden and the user is told to
                1620 * load in the second program.  Then the
                1630 * ampersand vector is set to jump to LOADED.
                1640 *
942C- A5 67     1650 NXTPROG LDA TXTTAB       The starting address of
942E- A4 68     1660          LDY TXTTAB+1    program 1 is stored
9430- 85 06     1670          STA BEGIN1      for later use.
9432- 84 07     1680          STY BEGIN1+1
9434- A5 AF     1690          LDA PRGEND      The ending address of
9436- A4 B0     1700          LDY PRGEND+1    program 1 is made the
9438- 85 67     1710          STA TXTTAB      beginning address for
943A- 85 08     1720          STA BEGIN2      program 2 and is also
943C- 84 68     1730          STY TXTTAB+1    saved for later use.
943E- 84 09     1740          STY BEGIN2+1
9440- A9 62     1750          LDA #LOADED     Ampersand vector
9442- A0 94     1760          LDY /LOADED     is reset to point
9444- 8D F6 03  1770          STA AMPERSD+1   to LOADED.
9447- 8C F7 03  1780          STY AMPERSD+2
944A- A9 10     1790          LDA #TEXT2      User is told to
944C- A0 95     1800          LDY /TEXT2      load the second
944E- 20 A5 94  1810          JSR MSGPRT      program and then
9451- A9 B2     1820          LDA #$B2        to press the '&' key.
9453- 20 9B 94  1830          JSR ENDMSG
9456- C6 AF     1840          DEC PRGEND      Mark the start
9458- D0 02     1850          BNE MARKIT      of the second
945A- C6 B0     1860          DEC PRGEND+1    program by storing
945C- A8        1870 MARKIT   TAY              a zero in the
945D- 91 AF     1880          STA (PRGEND),Y  first location.
945F- 4C 49 D6  1890          JMP NEW         NEW before loading program 2.
                1900 *
                1910 *
                1920 * Both programs have now been loaded so
                1930 * tell the user how to switch between them
                1940 * and reset the ampersand vector to the
                1950 * program switching routine.
                1960 *
9462- A9 1F     1970 LOADED   LDA #TEXT3      Point to text to
9464- A0 95     1980          LDY /TEXT3      be printed.
9466- 20 A5 94  1990          JSR MSGPRT      Print it.
9469- A9 4D     2000          LDA #TEXT4+4    Point to last part
946B- 20 A0 94  2010          JSR ENDMSG2     and print that too.
946E- A9 79     2020          LDA #SWITCH     Set up the ampersand
9470- A0 94     2030          LDY /SWITCH     vector to jump to
9472- 8D F6 03  2040          STA AMPERSD+1   the SWITCH routine.
```

```
9475- 8C F7 03 2050          STY AMPERSD+2
9478- 60       2060          RTS
               2070 *
               2080 *
               2090 * This is the routine that switches the
               2100 * availability of the programs in memory.
               2110 *
9479- A5 06    2120 SWITCH   LDA BEGIN1      Get the address of
947B- 85 67    2130          STA TXTTAB      the inactive program
947D- 48       2140          PHA             and save it and also
947E- A5 07    2150          LDA BEGIN1+1    make it the active
9480- 85 68    2160          STA TXTTAB+1    program.
9482- 48       2170          PHA
9483- A5 08    2180          LDA BEGIN2      Transfer address of
9485- A4 09    2190          LDY BEGIN2+1    former active program
9487- 85 06    2200          STA BEGIN1      to inactive location.
9489- 84 07    2210          STY BEGIN1+1
948B- 68       2220          PLA             Retrieve address of
948C- 85 09    2230          STA BEGIN2+1    current active program
948E- 68       2240          PLA             and put it in inactive
948F- 85 08    2250          STA BEGIN2      location.
9491- 60       2260          RTS
               2270 *
               2280 *
               2290 * This is where the USR function is
               2300 * implemented.  The programs are first
               2310 * switched and then run by executing a GOTO.
               2320 *
9492- 20 79 94 2330 USRGOTO JSR SWITCH       Switch programs.
9495- 20 52 E7 2340          JSR GETADR      Get the GOTO line number
9498- 4C 44 D9 2350          JMP GOTO        and go to it.
               2360 *
               2370 *
               2380 * This routine prints out the program
               2390 * number being loaded and tell the user
               2400 * to press the '&' key.
               2410 *
949B- 20 ED FD 2420 ENDMSG   JSR COUT        Print number in accumulator.
949E- A9 49    2430          LDA #TEXT4      Point to rest of
94A0- A0 95    2440 ENDMSG2  LDY /TEXT4      text and print it.
94A2- 4C A5 94 2450          JMP MSGPRT
               2460 *
               2470 *
               2480 * This is the message printing routine.
               2490 *
94A5- 85 18    2500 MSGPRT   STA TXTPTR
94A7- 84 19    2510          STY TXTPTR+1
94A9- A0 00    2520          LDY #$0
94AB- B1 18    2530 LOOP     LDA (TXTPTR),Y
94AD- F0 0B    2540          BEQ ENDPRT
94AF- 20 ED FD 2550          JSR COUT
94B2- E6 18    2560          INC TXTPTR
94B4- D0 F5    2570          BNE LOOP
94B6- E6 19    2580          INC TXTPTR+1
94B8- D0 F1    2590          BNE LOOP
94BA- 60       2600 ENDPRT   RTS
               2610 *
               2620 *
               2630 * These are the various text messages
               2640 * printed out by this program.
               2650 *
94BB- C1 D0 D0
94BE- CC C5 D3
94C1- CF C6 D4
94C4- A0 D0 D2
94C7- CF C7 D2
94CA- C1 CD A0
94CD- D3 C8 C1
94D0- D2 C5 D2 2660 TEXT1    .AS -"APPLESOFT PROGRAM SHARER"
94D3- 8D 8D    2670          .HS 8D8D
94D5- C2 D9 A0
94D8- CA D5 CC
94DB- C5 D3 A0
```



```
94DE- C8 AE A0
94E1- C7 C9 CC
94E4- C4 C5 D2 2680          .AS -"BY JULES H. GILDER"
94E7- 8D       2690          .HS 8D
94E8- C3 CF D0
94EB- D9 D2 C9
94EE- C7 C8 D4
94F1- A0 A8 C3
94F4- A9 A0 B1
94F7- B9 B8 B2 2700          .AS -"COPYRIGHT (C) 1982"
94FA- 8D       2710          .HS 8D
94FB- C1 CC CC
94FE- A0 D2 C9
9501- C7 C8 D4
9504- D3 A0 D2
9507- C5 D3 C5
950A- D2 D6 C5
950D- C4       2720          .AS -"ALL RIGHTS RESERVED"
950E- 8D 8D    2730          .HS 8D8D
9510- 8D       2740 TEXT2    .HS 8D
9511- CC CF C1
9514- C4 A0 D0
9517- D2 CF C7
951A- D2 C1 CD
951D- A0       2750          .AS -"LOAD PROGRAM "
951E- 00       2760          .HS 00
951F- 8D 8D    2770 TEXT3    .HS 8D8D
9521- C4 CF CE
9524- C5 A0 AD
9527- A0 D4 CF
952A- A0 D3 D7
952D- C9 D4 C3
9530- C8 A0 C2
9533- C5 D4 D7
9536- C5 C5 CE 2780          .AS -"DONE - TO SWITCH BETWEEN"
9539- A0 D0 D2
953C- CF C7 D2
953F- C1 CD D3
9542- AC A0 CA
9545- D5 D3 D4 2790          .AS -" PROGRAMS, JUST"
9548- 00       2800          .HS 00
9549- A0 C1 CE
954C- C4 A0 D0
954F- D2 C5 D3
9552- D3 A0 D4
9555- C8 C5 A0
9558- A7 A6 A7
955B- A0 CB C5
955E- D9 AE    2810 TEXT4    .AS -" AND PRESS THE '&' KEY."
9560- 8D 8D 00 2820          .HS 8D8D00
```

When using the APPLESOFT PROGRAM SHARER, it is very important that once the second program has been loaded, no changes are made to the first program. If you make changes, you'll move the second program away from the location that has already been specified as the start of the program. If changes must be made, do it on the original version of the program and then reload it and the second program as was previously described.

For those of you who are looking for a challenge and something interesting to do, try modifying the program so that it can be switched, sequentially, between any number of programs (assuming there's room in memory for all of them). If you want an even tougher assignment, permit the switching between programs on a random basis.

Add Applesoft function keys to your computer

Applesoft is quite a versatile version of BASIC with commands for changing the display mode from normal to inverse and flashing. Applesoft also has two commands that provide the ability to place the cursor at a specific position. Instead of having these as separate commands, some people prefer the approach that Commodore has taken in their computers, where a single character can be printed out to determine the display mode and relative cursor movements (move up three and left two).

With the program, APPLESOFT FUNCTION KEYS, you can now add this capability to your Apple. The program steals control away from both the input and the output. It monitors the input and looks for control characters that permit the user to switch between three display modes. One is the fully OPERATIONAL mode, which is entered with a Control-O, where all of the codes that have been entered are implemented. The second is a VIEW mode, which is entered with a Control-V. In the viewing mode, all of the control characters that are entered are visible on the screen in inverse, so that when the program is listed, you can see exactly what is going to happen. The third mode is a QUIT, or normal mode where the control characters are invisible. In this mode, the Apple is restored to its normal output configuration. This mode is entered by typing a Control-Q.

Since this program is too long to reside in page three of memory, it is designed to operate in high memory starting at location $9400. As with the previous program, the first thing this one does when it is run, is to protect itself from Applesoft programs by lowering HIMEM to $9400 (lines 1620 and 1630). Next the title of the program is printed out along with the word READY, so the user knows the program has been activated (lines 1640 to 1670). The final phase of this initialization process begins on line 1680, where control is taken away from the normal input routines and given to a routine that starts on line 2890. This is the routine that permits the switching between display modes by checking for the Control-O, Control-V and Control-Q characters, and jumping to the appropriate display mode routine.

Upon returning from the initialization routine (line 1780), everything appears normal, and will remain that way until one of the three previously mentioned control keys is pressed. If a Control-O is pressed, to place the program in the OPERATIONAL mode, a jump is made to a subroutine called ACTIVE, that begins on line 1840. This is the function key interpreter, and is the routine that should be called whenever the functions that the control keys stand for are to be executed. Generally, this is only active while the program is running. If you try to list out a program that uses function keys while in this mode, you'll get some real wild results.

This program uses a page zero location called FLAG to determine what mode the character to be printed should be displayed in. The first thing that this routine does, is to set FLAG for the normal mode by storing an $FF in it (lines 1840 and 1850). Next, a flashing cursor is stored on the screen to mark the position of the next character to be displayed (lines 1860 and 1870). After that, the output hooks

are set so they point to a subroutine labelled START, which is the beginning of the OPERATIONAL display mode (lines 1880 to 1950). Finally, the accumulator is loaded with a blank (line 1960), which will be used as the cursor character, when the program returns to fetch the next character from the keyboard (line 1970). It might seem like we're using two different techniques to display the same cursor. The reason is, this last one becomes important when the SAVOUT and INPRTN entry points are used.

Now that START has been activated, all characters that are to be printed out must first be tested to see if they are one of the control characters that determine the display mode or cursor movement. A whole series of comparisons take place between lines 2100 and 2370 in an effort to identify one of the eleven assigned control keys. A table of the keys and their functions can be seen below.

KEY	HEX CODE	FUNCTION
Control-A	$81	Move Left
Control-F	$86	FLASH
Control-H	$88	Back Space
Control-I	$89	INVERSE
Control-M	$0D	Carriage Return
Control-M	$8D	Carriage Return
Control-N	$8E	NORMAL
Control-P	$90	HOME
Control-S	$93	Move Right
Control-W	$97	Move Up
Control-Z	$9A	Move Down

If you look carefully at the chart, you'll notice two things. First of all, you'll see Control-H defined as performing a back space and Control-M defined as performing a carriage return. These are their normal functions, so you might ask why bother to include it to begin with. Secondly, you might notice that there are two

entries for Control-M. To answer the first question, when the OPERATIONAL mode is active, and inverse or flashing characters are being entered, the control characters are printed as inverse letters and are not implemented, so if you typed a carriage return (Control-M) all you'd get is an inverse M on the screen and no carriage return would be generated. The same is true for the Control-H and its back space function. The reason for the two Control-M entries is that while the characters that are entered from the keyboard have the high bit set, those that are generated by the computer (such as when a program is listed), do not.

After all of the tests have been performed and it is determined that the character is not one of the eleven assigned ones, the character is temporarily saved on the stack (line 2380) and FLAG is tested to see if the character to be printed is supposed to be converted to flashing (lines 2390 to 2400). If it is, control is passed to the CONVERT routine (line 2410), otherwise the character is pulled off the stack and it is ANDed with FLAG to convert it to either the normal mode or the inverse mode (lines 2420 and 2430). Once this adjustment has taken place, the character is printed out to the video screen (line 2440) and control is returned to the caller via the RTS in the COUT1 routine.

Next, we have a series of three, 3-line routines, whose only purpose is to set the value of FLAG for the appropriate mode. In the INVERSE and NORMAL routines (lines 2560 and 2630 respectively), the value that is stored in FLAG is the value that is ANDed with the character to produce a letter in the desired mode. In the case of the FLASH routine (line 2490), the value in FLAG is used as an indicator that another routine has to be used to do the conversion (lines 2400 and 2410).

The routine that does the flashing conversion starts on line 2710, where the character is retrieved from the stack where it was stored earlier. Next, a test is performed to see if the character is in the $A0 to $BF range and is therefore a number or a symbol (lines 2720 to 2750). If it is within this range, the character in the accumulator is Exclusive-ORed with the value $C0 and the printed out to the video screen (lines 2760 and 2770).

If the character being tested in lines 2720 to 2750 is a letter, control is passed to line 2790, where the letter is Exclusive-ORed with $80, which is the value in FLAG and then output to the screen (lines 2780 and 2790).

As was mentioned earlier, NWKEYIN, the routine that starts on line 2890, is the new input routine that checks for Control-O, Control-V and Control-Q being input from the keyboard. The monitor KEYIN routine is used to get a key press from the keyboard (line 2900) and the character is then checked to see if it is a Control-O. If it is, the program jumps to the ACTIVE subroutine in line 1840, otherwise, a check is made for a Control-V. If it is a Control-V, the program branches to VIEW on line 3030, which is simply a subroutine that changes the output hooks so that they point to the VIEWCTL routine.

The last test made on a character passing through NWKEYIN is for a Control-Q. If it is a Control-Q, the output hooks are set to the video screen output subroutine COUT1. If a character has managed to pass through this routine, it is entered

unchanged.

VIEWCTL is the subroutine that is used to display the control characters that are being used as function keys. It is very similar to the program SHOW CONTROL that we discussed in Chapter 4 and has only been slightly modified for use here. The modification consists of two parts. The first, does not display the Control-M character and the second does not permit the display of the Control-@ character.

In line 3250 we check for the presence of a carriage return and if it is found, bypass the program and print it out. This is a little different than the SHOW CONTROL program where the inverse M was printed out first and then the carriage return function was implemented. Next, characters are tested to see if they are in the control character range of $81 to $9F. This is where the other difference with the SHOW CONTROL program crops up. In the original, the test was made from $80 to $9F, to include the Control-@ code as well. If the character is not a control character, it is printed unchanged, but if it is, the high bit is turned off (line 3330) (which is the equivalent of subtracting $80) and the character is output to the video display (line 3340).

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    APPLESOFT FUNCTION KEYS   ***
                1030 ***                              ***
                1040 ***     COPYRIGHT (C) 1982 BY    ***
                1050 ***       JULES H. GILDER        ***
                1060 ***     ALL RIGHTS RESERVED      ***
                1070 ***                              ***
                1080 ************************************
                1090 *
                1100 *
                1110 *
                1120 *
                1130          .OR $9400
                1140          .TA $800
                1150 *
                1160 *
                1170 *
                1180 * CONSTANTS
                1190 *
000D-           1200 CNTRLM   .EQ $D
004C-           1210 JUMP     .EQ $4C
0081-           1220 CTRLA    .EQ $81
0086-           1230 CTRLF    .EQ $86
0088-           1240 CTRLH    .EQ $88
0089-           1250 CTRLI    .EQ $89
008A-           1260 CTRLJ    .EQ $8A
008D-           1270 CTRLM    .EQ $8D
008E-           1280 CTRLN    .EQ $8E
008F-           1290 CTRLO    .EQ $8F
0090-           1300 CTRLP    .EQ $90
0091-           1310 CTRLQ    .EQ $91
0093-           1320 CTRLS    .EQ $93
0096-           1330 CTRLV    .EQ $96
0097-           1340 CTRLW    .EQ $97
009A-           1350 CTRLZ    .EQ $9A
                1360 *
                1370 *
                1380 * EQUATES
                1390 *
0006-           1400 FLAG     .EQ $6
0008-           1410 CURSOR   .EQ $8
0009-           1420 PRTFLG   .EQ $9
0018            1430 TXTPTR   .EQ $18
0028            1440 BASL     .EQ $28
```

```
0036-           1450 CSWL     .EQ $36
0038-           1460 KSWL     .EQ $38
0073-           1470 HIMEM    .EQ $73
03D0-           1480 WARMDOS  .EQ $3D0
03EA-           1490 CONNECT  .EQ $3EA
FBF4-           1500 ADVANCE  .EQ $FBF4
FC10-           1510 BS       .EQ $FC10
FC1A-           1520 UP       .EQ $FC1A
FC58-           1530 HOME     .EQ $FC58
FD1B-           1540 KEYIN    .EQ $FD1B
FDED-           1550 COUT     .EQ $FDED
FDF0-           1560 COUT1    .EQ $FDF0
                1570 *
                1580 *
                1590 * This section steals control of the
                1600 * input routine.
                1610 *
9400- A9 94     1620          LDA #$94        Lower HIMEM to $9400
9402- 85 74     1630          STA HIMEM+1     to protect this program.
9404- 20 58 FC  1640          JSR HOME        Clear the screen.
9407- A9 04     1650          LDA #TEXT       Point to the text
9409- A0 95     1660          LDY /TEXT       to be printed.
940B- 20 EE 94  1670          JSR MSGPRT      Print it.
940E- A9 B0     1680          LDA #NWKEYIN    Get the address of the
9410- A0 94     1690          LDY /NWKEYIN    new input routine
9412- 85 38     1700          STA KSWL        and store it in the
9414- 84 39     1710          STY KSWL+1      input hooks.
9416- 84 09     1720          STY PRTFLG      Make flag nonzero.
9418- AD D0 03  1730          LDA WARMDOS     Is DOS present?
941B- C9 4C     1740          CMP #JUMP
941D- D0 03     1750          BNE NODOS       No.
941F- 20 EA 03  1760          JSR CONNECT     Yes, connect to DOS.
9422- A9 80     1770 NODOS    LDA #$80        Return with null character.
9424- 60        1780          RTS
                1790 *
                1800 *
                1810 * This routine steals control away
                1820 * from the normal output routine.
                1830 *
9425- A9 FF     1840 ACTIVE   LDA #$FF        Set mode flag
9427- 85 06     1850          STA FLAG        for normal text.
9429- A9 60     1860          LDA #$60        Store flashing
942B- 91 28     1870          STA (BASL),Y    cursor on the screen.
942D- A9 44     1880          LDA #START      Get the address of the
942F- A0 94     1890          LDY /START      start of the program.
9431- 85 36     1900 SAVOUT   STA CSWL        Store it in the
9433- 84 37     1910          STY CSWL+1      output
9435- AD D0 03  1920 INPRTN   LDA WARMDOS     Check for presence
9438- C9 4C     1930          CMP #JUMP       of DOS.
943A- D0 03     1940          BNE NODOS2
943C- 20 EA 03  1950          JSR CONNECT     Connect through DOS.
943F- A9 A0     1960 NODOS2   LDA #$A0        Set blank as input prompt.
9441- 4C 1B FD  1970          JMP KEYIN       Get the next character.
                1980 *
                1990 *
                2000 * This routine replaces the normal
                2010 * ouput routine and checks to see
                2020 * if any of the Control characters that
                2030 * are used as function descriptors are
                2040 * being output.  If not, the character
                2050 * passes through this program
                2060 * unchanged and is printed.  If so,
                2070 * the program jumps to the appropriate
                2080 * routine to implement the function.
                2090 *
9444- C9 86     2100 START    CMP #CTRLF      Is it Ctrl-F?
9446- F0 46     2110          BEQ FLASH       Yes, flash it.
9448- C9 8D     2120          CMP #CTRLM      Is it a carriage return?
944A- F0 3F     2130          BEQ PRTRTN      Yes, print it.
944C- C9 0D     2140          CMP #CNTRLM     Is it a carriage return?
944E- F0 3B     2150          BEQ PRTRTN      Yes, print it.
9450- C9 88     2160          CMP #CTRLH      Is it Ctrl H?
9452- F0 37     2170          BEQ PRTRTN      Yes, print it.
```



```
9454- C9 89     2180          CMP #CTRLI      Is it Ctrl-I?
9456- F0 3B     2190          BEQ INVERSE     Yes, inverse it.
9458- C9 8E     2200          CMP #CTRLN      Is it Ctrl-N?
945A- F0 3C     2210          BEQ NORMAL      Yes, make it normal.
945C- C9 90     2220          CMP #CTRLP      Is it Ctrl-P?
945E- D0 03     2230          BNE CHKCTLW     No, check if Ctrl-W.
9460- 4C 58 FC  2240          JMP HOME        Clear screen.
9463- C9 97     2250 CHKCTLW  CMP #CTRLW      Is it Ctrl-W?
9465- D0 03     2260          BNE CHKCTLZ     No, check if Ctrl-Z.
9467- 4C 1A FC  2270          JMP UP          Move cursor up.
946A- C9 9A     2280 CHKCTLZ  CMP #CTRLZ      Is it Ctrl-Z?
946C- D0 05     2290          BNE CHKCTLA     No, check if Ctrl-A.
946E- A9 8A     2300          LDA #CTRLJ      Make it Ctrl-J.
9470- 4C F0 FD  2310          JMP COUT1
9473- C9 81     2320 CHKCTLA  CMP #CTRLA      Is it Ctrl-A?
9475- D0 03     2330          BNE CHKCTLS     No, check if Ctrl-S.
9477- 4C 10 FC  2340          JMP BS          Back space.
947A- C9 93     2350 CHKCTLS  CMP #CTRLS      Is it Ctrl-S?
947C- D0 03     2360          BNE CONTIN      No, continue processing.
947E- 4C F4 FB  2370          JMP ADVANCE     Yes, move cursor right.
9481- 48        2380 CONTIN   PHA             Save character.
9482- A5 06     2390          LDA FLAG        Get flag and check
9484- C9 80     2400          CMP #$80        for flashing.
9486- F0 15     2410          BEQ CONVERT     Yes, check for numbers.
9488- 68        2420          PLA             No flashing.
9489- 25 06     2430          AND FLAG        Adjust for mode.
948B- 4C F0 FD  2440 PRTRTN   JMP COUT1       Print character and return.
                2450 *
                2460 *
                2470 * Set FLAG for flashing mode.
                2480 *
948E- A9 80     2490 FLASH    LDA #$80        Set FLAG for
9490- 85 06     2500          STA FLAG        flash mode.  2
9492- 60        2510          RTS
                2520 *
                2530 *
                2540 * Set FLAG for inverse mode.
                2550 *
9493- A9 3F     2560 INVERSE  LDA #$3F        Set FLAG for
9495- 85 06     2570          STA FLAG        inverse mode.
9497- 60        2580          RTS
                2590 *
                2600 *
                2610 * Set FLAG for normal mode.
                2620 *
9498- A9 FF     2630 NORMAL   LDA #$FF        Set FLAG for
949A- 85 06     2640          STA FLAG        normal mode.
949C- 60        2650          RTS
                2660 *
                2670 *
                2680 * This routine converts the character
                2690 * being output to the flashing mode.
                2700 *
949D- 68        2710 CONVERT  PLA             Retrieve character.
949E- C9 A0     2720          CMP #$A0        Is it a number
94A0- 90 09     2730          BCC FIXLTR      or symbol (in the
94A2- C9 C0     2740          CMP #$C0        range of $A0 to $BF).
94A4- B0 05     2750          BCS FIXLTR      No, it's alpha.
94A6- 49 C0     2760          EOR #$C0        Fix number.
94A8- 4C F0 FD  2770          JMP COUT1       Print it out.
94AB- 45 06     2780 FIXLTR   EOR FLAG        Fix alpha.
94AD- 4C F0 FD  2790          JMP COUT1       Print it out.
                2800 *
                2810 *
                2820 * This is the replacement input routine
                2830 * which checks to see if any of the mode
                2840 * switching keys (Control-Q, Control-O
                2850 * or Control-V) are being pressed.  If
                2860 * so, control is passed to the appropriate
                2870 * subroutine.
                2880 *
94B0- 20 1B FD  2890 NWKEYIN  JSR KEYIN       Read the keyboard.
94B3- C9 8F     2900          CMP #CTRLO      Was it Ctrl O?
```

```
94B5- D0 03    2910          BNE CHKCTLV     No, see if Ctrl-V.
94B7- 4C 25 94 2920          JMP ACTIVE      Activate function interpreter.
94BA- C9 96    2930 CHKCTLV  CMP #CTRLV      Was it Ctrl-V?
94BC- F0 05    2940          BEQ VIEW        Yes, show control characters.
94BE- C9 91    2950          CMP #CTRLQ      Was it Ctrl-Q?
94C0- F0 0C    2960          BEQ QUIT        Yes, inactivate program.
94C2- 60       2970          RTS
               2980 *
               2990 *
               3000 * Here the output routine is set up to
               3010 * display control characters.
               3020 *
94C3- A9 60    3030 VIEW     LDA #$60        Store flashing
94C5- 91 28    3040          STA (BASL),Y    cursor on the screen.
94C7- A9 D9    3050          LDA #VIEWCTL    Get new address for
94C9- A0 94    3060          LDY /VIEWCTL    output routine.
94CB- 4C 31 94 3070          JMP SAVOUT      Store address in hooks.
               3080 *
               3090 *
               3100 * Here the output hooks are set to
               3110 * restore the output hooks to normal.
               3120 *
94CE- A9 60    3130 QUIT     LDA #$60        Store flashing
94D0- 91 28    3140          STA (BASL),Y    cursor on the screen.
94D2- A9 F0    3150          LDA #COUT1      Put screen address
94D4- A0 FD    3160          LDY /COUT1      in the output hooks.
94D6- 4C 31 94 3170          JMP SAVOUT
               3180 *
               3190 *
               3200 * This routine permits the viewing of
               3210 * control characters (except null,
               3220 * carriage return and backspace) as
               3230 * inverse on the screen.
               3240 *
94D9- C9 8D    3250 VIEWCTL  CMP #CTRLM      Is it carriage return?
94DB- F0 0E    3260          BEQ PRINTIT     Yes, print it.
94DD- C9 88    3270          CMP #CTRLH      Is it a backspace?
94DF- F0 0A    3280          BEQ PRINTIT     Yes, print it.
94E1- C9 81    3290          CMP #$81        Is it a Control character
94E3- 90 06    3300          BCC PRINTIT     in the range of $81 to $9F?
94E5- C9 9F    3310          CMP #$9F        If not print it.
94E7- B0 02    3320          BCS PRINTIT
94E9- 29 7F    3330          AND #$7F        Otherwise, inverse it.
94EB- 4C F0 FD 3340 PRINTIT  JMP COUT1       Print character.
               3350 *
               3360 *
               3370 * This is the message printing routine.
               3380 *
94EE- 85 18    3390 MSGPRT   STA TXTPTR
94F0- 84 19    3400          STY TXTPTR+1
94F2- A0 00    3410          LDY #$0
94F4- B1 18    3420 LOOP     LDA (TXTPTR),Y
94F6- F0 0B    3430          BEQ ENDPRT
94F8- 20 ED FD 3440          JSR COUT
94FB- E6 18    3450          INC TXTPTR
94FD- D0 F5    3460          BNE LOOP
94FF- E6 19    3470          INC TXTPTR+1
9501- D0 F1    3480          BNE LOOP
9503- 60       3490 ENDPRT   RTS
               3500 *
               3510 *
               3520 * This is the text printed out by
               3530 * this program.
9504- C1 D0 D0
9507- CC C5 D3
950A- CF C6 D4
950D- A0 C6 D5
9510- CE C3 D4
9513- C9 CF CE
9516- A0 CB C5
9519- D9 D3    3540 TEXT     .AS -"APPLESOFT FUNCTION KEYS"
951B- 8D 8D    3550          .HS 8D8D
951D- C2 D9 A0
```



```
9520- CA D5 CC
9523- C5 D3 A0
9526- C8 AE A0
9529- C7 C9 CC
952C- C4 C5 D2 3560          .AS -"BY JULES H. GILDER"
952F- 8D       3570          .HS 8D
9530- C3 CF D0
9533- D9 D2 C9
9536- C7 C8 D4
9539- A0 A8 C3
953C- A9 A0 B1
953F- B9 B8 B2 3580          .AS -"COPYRIGHT (C) 1982"
9542- 8D       3590          .HS 8D
9543- C1 CC CC
9546- A0 D2 C9
9549- C7 C8 D4
954C- D3 A0 D2
954F- C5 D3 C5
9552- D2 D6 C5
9555- C4       3600          .AS -"ALL RIGHTS RESERVED"
9556- 8D 8D 8D 3610          .HS 8D8D8D
9559- D2 C5 C1
955C- C4 D9 AE 3620          .AS -"READY."
955F- 8D 00    3630          .HS 8D00
```

Appendix A

ASCII CODE CONVERSION TABLE

DECIMAL		0	16	32	48	64	80	96	112
	HEX	$0	$10	$20	$30	$40	$50	$60	$70
0	$0	NUL	DLE	SPACE	0	@	P	`	p
1	$1	SOH	DC1	!	1	A	Q	a	q
2	$2	STX	DC2	"	2	B	R	b	r
3	$3	ETX	DC3	#	3	C	S	c	s
4	$4	EOT	DC4	$	4	D	T	d	t
5	$5	ENQ	NAK	%	5	E	U	e	u
6	$6	ACK	SYN	&	6	F	V	f	v
7	$7	BEL	ETB	'	7	G	W	g	w
8	$8	BS	CAN	(	8	H	X	h	x
9	$9	HT	EM	)	9	I	Y	i	y
10	$A	LF	SUB	*	:	J	Z	j	z
11	$B	VT	ESC	+	;	K	[	k	{
12	$C	FF	FS	,	<	L	\	l	|
13	$D	CR	GS	-	=	M	]	m	}
14	$E	SO	RS	.	>	N	^	n	~
15	$F	SI	US	/	?	O	—	o	DEL

Appendix B

APPLESOFT TOKEN LIST

DECIMAL TOKEN	HEX TOKEN	APPLESOFT KEYWORD	DECIMAL TOKEN	HEX TOKEN	APPLESOFT KEYWORD
128	$80	END	160	$A0	COLOR =
129	$81	FOR	161	$A1	POP
130	$82	NEXT	162	$A2	VTAB
131	$83	DATA	163	$A3	HIMEM:
132	$84	INPUT	164	$A4	LOMEM:
133	$85	DEL	165	$A5	ONERR
134	$86	DIM	166	$A6	RESUME
135	$87	READ	167	$A7	RECALL
136	$88	GR	168	$A8	STORE
137	$89	TEXT	169	$A9	SPEED =
138	$8A	PR#	170	$AA	LET
139	$8B	IN#	171	$AB	GOTO
140	$8C	CALL	172	$AC	RUN
141	$8D	PLOT	173	$AD	IF
142	$8E	HLIN	174	$AE	RESTORE
143	$8F	VLIN	175	$AF	&
144	$90	HGR2	176	$B0	GOSUB
145	$91	HGR	177	$B1	RETURN
146	$92	HCOLOR =	178	$B2	REM
147	$93	HPLOT	179	$B3	STOP
148	$94	DRAW	180	$B4	ON
149	$95	XDRAW	181	$B5	WAIT
150	$96	HTAB	182	$B6	LOAD
151	$97	HOME	183	$B7	SAVE
152	$98	ROT =	184	$B8	DEF
153	$99	SCALE =	185	$B9	POKE
154	$9A	SHLOAD	186	$BA	PRINT
155	$9B	TRACE	187	$BB	CONT
156	$9C	NOTRACE	188	$BC	LIST
157	$9D	NORMAL	189	$BD	CLEAR
158	$9E	INVERSE	190	$BE	GET
159	$9F	FLASH	191	$BF	NEW

DECIMAL TOKEN	HEX TOKEN	APPLESOFT KEYWORD	DECIMAL TOKEN	HEX TOKEN	APPLESOFT KEYWORD
192	$C0	TAB(	214	$D6	FRE
193	$C1	TO	215	$D7	SCRN(
194	$C2	FN	216	$D8	PDL
195	$C3	SPC(	217	$D9	POS
196	$C4	THEN	218	$DA	SQR
197	$C5	AT	219	$DB	RND
198	$C6	NOT	220	$DC	LOG
199	$C7	STEP	221	$DD	EXP
200	$C8	+	222	$DE	COS
201	$C9	−	223	$DF	SIN
202	$CA	*	224	$E0	TAN
203	$CB	/	225	$E1	ATN
204	$CC	^	226	$E2	PEEK
205	$CD	AND	227	$E3	LEN
206	$CE	OR	228	$E4	STR$
207	$CF	>	229	$E5	VAL
208	$D0	=	230	$E6	ASC
209	$D1	<	231	$E7	CHR$
210	$D2	SGN	232	$E8	LEFT$
211	$D3	INT	233	$E9	RIGHT$
212	$D4	ABS	234	$EA	MID$
213	$D5	USR			

Appendix C

SHIFT KEY MODIFICATION FOR APPLE II AND II PLUS COMPUTERS

The Apple II and II Plus computers do not have any way of allowing the user to enter upper and lower case letters from the keyboard. While the computer keyboard does have a SHIFT key on it, it is only usable to get the alternate characters on the number, punctuation and M, N and P keys.

It is possible however, to make a very simple, one-wire, modification to your Apple computer that will allow you, with the aid of an appropriate program, to determine if the SHIFT key has been pressed and then retrieve the character from the keyboard and adjust it appropriately. We have already spoken about the software required to do the job in Chapter 5, here are the instructions to modify the hardware.

Over the years, Apple Computer has made some minor changes to its computers, and as a result, newer models of the Apple II Plus have a different keyboard than the older ones. As a result of this, there are two sets of instructions for the modification. One for older Apples that have marked on the circuit board REV 6 or earlier, and one for the later versions which are marked REV 7 or higher. In both cases, a wire is connected to the SHIFT key on one end and the Game I/O socket on the other. The software that supports this modification, then looks at the particular pin on the Game I/O socket that the wire is connected to and checks to see if the switch is closed.

NOTE: The following procedures may void your computer's warranty. Do not attempt the next modification (for REV 6 or earlier computers) unless you know how to solder. The REV 7 modification does not require soldering. If you are unsure or encounter any difficulty, ask your dealer to make the modification for you, otherwise proceed at your own risk. Also, be aware that Apple IIe and IIc keyboards do not have to be modified.

Modifying Revision 6 and earlier computers

This modification requires some physical changes to the Apple computer and should be done with care. Follow the steps below.

1. Shut off all power to the computer and remove all peripheral cards and anything plugged into the Game I/O socket.

2. Turn your Apple over so that the metal baseplate is facing up and remove the

four screws below the keyboard, the two screws along the sides and the two screws in the rear corners. DO NOT REMOVE THE BASEPLATE YET!

3. Carefully lift the baseplate an inch or two. You will see a cable going from the keyboard to the computer's main circuit board. Note the orientation of the plug on the ribbon cable and then carefully unplug it from the main computer board.

4. Now you can fully remove the baseplate, which holds the main circuit board, and place it aside.

5. Take an 18-inch piece of thin wire and strip off about 1/4-inch of insulation from both ends of it. With the computer case still upside down, and the front (keyboard) closest to you, look in the lower right-hand corner of the keyboard circuit board and locate the number 42. Next to it should be three solder pads, all in a straight row.

6. Solder one end of the 18-inch wire to the left-most pad, the one that has the empty hole. Near the solder connection, tape the wire to the keyboard circuit board and then point the wire towards the back of the computer.

7. Carefully reassemble the baseplate, not forgetting to plug the keyboard connector back into the main computer board.

8. Turn the Apple over (right side up) and insert the free end of the wire into pin 4 of the Game I/O socket. To orient you properly, when viewing the Game I/O socket from the front, pin 1 is in the lower right-hand corner of the socket, pin 2 is behind it, etc.

That's it. The modification has been completed and when used with the lowercase software in Chapter 5, will allow your Apple keyboard to behave more like a typewriter keyboard.

Modifying Revision 7 and later computers

This modification is a little simpler to perform than the one for the earlier model Apples because it does not require any soldering.

1. Turn off all power to the computer and remove anything that is plugged into the Game I/O socket.

2. Take a 12-inch piece of thin, solid wire and strip off 1/4-inch of insulation from one end and 3/4-inch of insulation from the other. On the end with the 3/4-inch of insulation missing, bend the wire into a hook shape.

3. Remove the top cover and locate the keyboard encoder circuit board, which is underneath the keyboard and connected to it by means of 25 long connector pins.

4. The SHIFT key is connected to the second pin from the right as you face the computer with the keyboard closest to you. Place the hooked wire you prepared in step 2, around this pin and squeeze it closed tight with a pair of thin pliers. Place insulating tape over this connection so that none of the other pins come in contact with this wire or connection.

5. Push the other end of the wire into pin 4 of the Game I/O socket. The modification is complete.

Don't take chances

While both of these modifications are very simple to perform, and are being used by thousands of people, some people are simply not mechanically inclined. If you're one of these people with two left hands, go to your dealer or some other technically knowledgeable person to have the modification done.

Appendix D

ADAPTING PROGRAMS TO WORK WITH PRODOS

Many of the programs in this book are designed to steal control away from the normal input or output routines during the course of their operation. Under DOS 3.3, to do this, all you had to do was place the address of the new input or output routine in either the input ($38 and $39) or output ($36 and $37) hook and then tell DOS that you did it by doing a subroutine jump to location $3EA. This technique will not work under ProDOS.

Installing a new input or output routine is a little more complicated under ProDOS, because you must place the address of the new routine in one of two places, depending on how the program is going to be run. If the program is either going to be activated from the immediate mode or activated by BRUNing it, then the address of the new routine goes in the same input and output hooks that were used for DOS 3.3, you just don't do the subroutine jump to $3EA. However, if the program is going to be activated by BLOADing it and then doing a CALL to the machine language routine when it is needed, the address of the new routine must be placed in ProDOS's global page (page $BE). The output routine address in the global page is stored at $BE30 and $BE31 and is normally set to $FDF0. The input address on the global page is stored at $BE32 and $BE33 and is normally set to $FD1B.

The reason for the two different places for storing the address of the I/O (input/output) routines is related to the way ProDOS initializes the values stored in the I/O hooks ($36 to $39). If you're BRUNing a program or executing it from the immediate mode, the first thing that ProDOS does is to initialize the values in the I/O hooks by copying the contents of $BE30 to $BE33 into $36 to $39. At that point it executes your command and runs your machine language program. If your program stores the new address for the I/O hooks in the global page ($BE30 to $BE33), it will remain there only as long as your program is running. As soon as your program returns control to whatever program or mode called it, ProDOS immediately copies the addresses in $36 through $39 back into the global page, immediately disconnecting the new I/O hooks your program just set up. However, if your machine language program had set up the new I/O addresses at S36 through $39, when ProDOS copied these values back to the global page, it would have made sure that the new I/O addresses remained connected.

The situation is a little different if you're going to operate your program by first BLOADing it and then doing a CALL to its starting address to activate it. In this case, things proceed as they did in the previous example except that the CALL

command which runs your program is not executed until after the I/O routine addresses are copied back to the global page. Thus, since the global page is where you want the adresses ultimately stored, your program must put them there itself. If it stored them in $36 through $39, they'd stay there and never get copied back to the global page.

If you want to avoid the problem of determining in advance how your program is going to be activated, you can store the addresses of the new I/O routines in both the zero page locations ($36 through $39) and the global page locations ($BE30 through $BE33). This will allow your program to work both ways, just as it did under DOS 3.3. To demonstrate how to do this, the SHOW CONTROL program which makes it possible to see the normally invisible control characters has been converted for use with ProDOS and is listed here. Compare this with the original program listed in Chapter 4. Notice that in lines 1330 to 1380 in the ProDOS version, that the address of the new output routine has been stored in both the zero page and the global page. You will also notice that the test for DOS has been eliminated as has the jump to $3EA which is required under DOS 3.3 to connect the I/O hooks. The rest of the program remains unchanged. As you can see, the change to accomodate ProDOS is really minimal.

```
                1000 ************************************
                1010 ***                              ***
                1020 ***    SHOW CONTROL CHARACTERS   ***
                1030 ***         PRODOS VERSION       ***
                1040 ***                              ***
                1050 ***     COPYRIGHT (C) 1984 BY    ***
                1060 ***        JULES H. GILDER       ***
                1070 ***      ALL RIGHTS RESERVED     ***
                1080 ***                              ***
                1090 ************************************
                1100 *
                1110 *
                1120 *
                1130          .OR $300
                1140 *
                1150 *
                1160 *
                1170 * EQUATES
                1180 *
0036-           1190 CSWL     .EQ $36
BE30-           1200 GPCSWL   .EQ $BE30
FDF0-           1210 COUT1    .EQ $FDF0
                1220 *
                1230 *
                1240 * This section of code sets up the
                1250 * output hooks at $36 and $37
                1260 * and on the global page so that
                1270 * any characters that are being output
                1280 * by the computer will first pass
                1290 * through this subroutine.  With this
                1300 * setup, it doesn't matter if the program
                1310 * is BRUN or BLOADed and then CALLed.
                1320 *
0300- A9 0F     1330          LDA #START      Get START low
0302- 85 36     1340          STA CSWL        byte & save it on
0304- 8D 30 BE  1350          STA GPCSWL      zero and global pages.
0307- A9 03     1360          LDA /START      Get START high
0309- 85 37     1370          STA CSWL+1      byte & save it on
030B- 8D 31 BE  1380          STA GPCSWL+1    zero and global pages.
030E- 60        1390          RTS
                1400 *
```



```
                1410 *
                1420 * This is the actual start of the
                1430 * control character display program.
                1440 * Here a check is made to see if the
                1450 * character is a Control-M (carriage
                1460 * return).  If it is, an inverse M is
                1470 * printed followed by a carriage
                1480 * return.  Otherwise control is passed
                1490 * to a routine that checks to see if
                1500 * the character is a control character.
                1510 *
030F- C9 8D     1520 START    CMP #$8D        Is it Cntrl-M?
0311- D0 08     1530          BNE CHKCTRL     No, inverse it.
0313- 48        1540          PHA             Yes, save it.
0314- 20 1B 03  1550          JSR CHKCTRL     To inverse.
0317- 68        1560          PLA             Restore it.
0318- 4C 25 03  1570          JMP PRINTIT     Print a carriage return.
                1580 *
                1590 *
                1600 * Here a check is made to see if the
                1610 * character in the accumulator is a
                1620 * control character.  If it's not, it
                1630 * is printed as is.  If it is, the
                1640 * character is converted to inverse and
                1650 * then printed.
                1660 *
031B- C9 80     1670 CHKCTRL CMP #$80         See if the accumulator
031D- 90 06     1680          BCC PRINTIT     contains a
031F- C9 9F     1690          CMP #$9F        control character.
0321- B0 02     1700          BCS PRINTIT     No, print it.
0323- 49 80     1710          EOR #$80        Yes, inverse it.
0325- 4C F0 FD  1720 PRINTIT JMP COUT1        Print character.
```

Finding space for long machine language programs

As with DOS 3.3, short machine language programs under ProDOS can be stored on page 3 of memory. Long machine language programs, however, are treated a little differently under ProDOS than they were under DOS 3.3. With DOS 3.3, these long programs were usually loaded under HIMEM, which was usually set at $9600, and then the value of HIMEM was lowered to the starting address of the machine language program. This protected the machine language program from being wiped out by strings used in Applesoft programs.

Something similar can be done with ProDOS, but there are some differences. To begin with, ProDOS doesn't like HIMEM to have just any old value, but insists that the value of HIMEM be a multiple of 256. This is not serious, because at the most you're only wasting a fraction of a page (256 bytes) of memory. Another problem is that as more files are opened, ProDOS takes away memory from the upper boundary and moves HIMEM down. Thus, while resetting HIMEM will result in an initially safe location for your machine language code, as more files are opened, the location where your program is stored is in danger of being over-written.

To overcome this problem, we can make use of a ProDOS subroutine known as GETBUFR, which is located at $BEF5. This is the subroutine that ProDOS uses to move HIMEM down and create a safe area to use as a file buffer, and there's no reason why we can't use it too. To use this routine, all you have to do is determine how many pages of memory you need and load this number into the accumulator. With the number of pages in the accumulator, all you have to do then is do a JSR to

GETBUFR. HIMEM is then moved down by that number of pages and a safe hole in high memory is now created. This safe area is located 4 pages above the new value of HIMEM. That's because ProDOS needs a 1K buffer immediately above HIMEM. Thus, if the value of HIMEM is at it's normal $9600 and you call the GETBUFR routine with a 3 in the accumulator (you want to reserve 768 bytes — three pages — of memory) the new value of HIMEM will be $9300 and your usable buffer area will begin at $9700.

If everything is okay when you return from the GETBUFR subroutine jump, the accumulator will contain the high byte of the address of the buffer (the low byte is always zero). If too many buffers have already been allocated, then on returning from GETBUFR the Carry bit should be set and the error code number is in the accumulator. Some programmers have reported a problem with this and found that in some instances even when an error occurs, the Carry bit is clear. I have not experienced this problem, but if reports in some magazines are accurate, you might. To overcome this, you can test the value of the byte in the accumulator to see if it is equal to $0C, which is the error code for "NO BUFFERS AVAILABLE".

One more ProDOS subroutine that you will find of interest is the one called FREBUFR, which is located at $BEF8. If you do a JSR to this routine, it will free all the buffers and reset the computer to it normal condition.

INDEX

Q

R

S

T

U

V

W

$19.95

Now That You Know APPLE ASSEMBLY LANGUAGE: What Can You Do With It?

Here is an easy-to-understand collection of programs to help you unlock the power and speed of assembly language programming on your Apple // computer. You've spent a great deal of time learning the various assembly language commands, what they do and how. Now you can put these commands together to produce fast, powerful programs that let you greatly expand the capabilities of your computer.

You'll find out how you can let your Apple do things it couldn't do before! With this book you'll discover how to:

- Hold two Applesoft BASIC programs in memory simultaneously and switch between them at will, even under program control.
- Reverse the way text is displayed on a video monitor to show black characters on a white background.
- Add new commands to Applesoft BASIC so that it will be more powerful and easier to use.
- Convert a section of the normal keyboard into a numeric keypad for super-fast entry of numerical data.
- Use a custom-developed form of shorthand that automatically types out one or more BASIC commands when you press just one or two keys.
- Permit older versions of the Apple computer to recognize lowercase letters, even in BASIC programs.
- Restore Applesoft programs that have been accidentally erased.

All this and much more is possible with the machine language programs described and listed here. And it's all done exclusively with software, no additional hardware or peripheral devices are required.

If you're worried about getting confused and not understanding the operation of these powerful programs, *don't.* All 55 programs in this book are fully documented with detailed descriptions in the text of how and why the programs work, and line-by-line descriptions of each step in every assembly language program listed.

You will find this to be an invaluable source-book of ideas, techniques and routines that can be incorporated into your own programs.

ABOUT THE AUTHOR

Jules H. Gilder is a pioneer in the use of personal computers. He was one of a handful of people to purchase and use the original Apple I computer, on which he taught himself 6502 assembly language programming. Since then he has taught hundreds of people both BASIC and assembly language programming in courses at New York University and private seminars. He has been editor-in-chief of *Personal Computing* magazine, vice president of software for Children's Television Workshop and editorial director of Hayden Software. He is the author of seven other books covering integrated software, and science and engineering programs in Pascal and BASIC for the Apple and IBM PC computers.

DataCraft, Inc.
2068 - 79th Street
Brooklyn, NY 11214

ISBN 0-933913-00-1