6502 Software Gourmet Guide & Cookbook

6502 SOFTWARE Gourmet Guide & COOKBOOK

Put Together
Your Own 6502 Programs
Using These
Time-Tested Recipes

— General Purpose Routines —
— Conversion Routines —
— Search and Sort Routines —
— Floating Point Routines —
— 6502 Instruction Set —
— And More —

SCELBI Publications

6502
SOFTWARE
Gourmet Guide &
COOKBOOK

By Robert Findley

Copyright © 1979
Scelbi Computer Consulting, Inc.
Elmwood, CT 06110

ALL RIGHTS RESERVED

IMPORTANT NOTICE

No part of this publication may be reproduced, transmitted, stored in a retrieval system, or otherwise duplicated in any form or by any means electronic, mechanical, photocopying, recording or otherwise, without the prior express written consent of the copyright owner.

The information in this manual has been carefully reviewed, and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies or for the success or failure of various applications to which the information contained herein might be applied.

Foreword

The "6502 Software Gourmet Guide & Cookbook" is written as an instructional publication for two audiences. First, it takes the BASIC language programmer into the realm of machine-language programming on the 6502. With the large number of computers on the market that use the 6502 as its central processor, one can find a new challenge by going one step closer to the inner workings of the CPU. There are many advantages to programming the 6502 at the machine-language level. This book presents these advantages in a way that a person with an introductory knowledge of computers will understand.

Second, the book is intended for the person with a knowledge of machine-language programming on a different CPU (i.e., 8080 or 6800) and wishes to become familiar with the 6502. The description of the 6502 structure and instruction set, along with the numerous applications discussed throughout the book, will quickly make an experienced programmer proficient with the 6502.

Robert Findley

November, 1979

ACKNOWLEDGEMENT

The author wishes to thank his wife, Barbara, and the staff at Scelbi for their invaluable help in preparing this book.

Contents

Introduction Page 7

1 6502 Instruction Set Page 9

2 6502 Programming Techniques Page 43

3 General Purpose Routines Page 49

4 Conversion Routines Page 71

5 Floating Point Routines Page 91

6 Decimal Arithmetic Routines Page 125

7 Input/Output Processing Page 139

8 Search and Sort Routines Page 169

Appendices Page 192

Index Page 204

Introduction

Have you tried cooking up a program lately on your 6502 microcomputer, and you just can't seem to get the right mixture of instructions? Or did that math recipe your friend gave you turn out to have too many bugs in it, and leave a sour taste in your mouth? Don't toss your computer in the sink and grind those bad listings up in the garbage disposal. Here's a book that will help take you from a novice that burns the bits to a gourmet chef that can make the sweetest APPLEcations program pie imaginable.

Before throwing together your favorite dish, a thorough knowledge of the basic ingredients, namely the 6502 instruction set, is essential. Every chef that's worth his salt knows exactly what each ingredient will do for him. Begin creating your masterpiece by mixing in a little of this routine and a little of that routine. Spice up the program with a few of your own special application routines, and before baking, add a personal touch by folding in the input/output driver routines for the peripherals in your system. Bake thoroughly with your assembler, and there you have it! Your programming masterpiece, ready to feed into your computer's memory for hours of tasty enjoyment.

Is your taste for math routines? Or manipulating data tables and character strings? Or maybe you wish to do some real time programming. Or set up your system to operate the peripherals under interrupt control. Whatever your requirements may be, there is certain to be some ideas, techniques, and routines in this book to aid you in programming for your specific application.

Chapter 1

The 6502 Instruction Set

The instruction set of the 6502 CPU provides considerable programming power to the machine language programmer. There are 56 basic instructions which, when all permutations are considered, provide 151 individual instructions. These instructions use from one to three bytes of memory depending on the function they perform.

There are several basic elements in the structure of the 6502 CPU with which the programmer must become thoroughly familiar. These elements include the Program Counter, the Accumulator, the two Index Registers, the Stack Pointer, Memory, and the Status Flags. Also, an understanding of certain concepts is important. For instance, with the 6502, input and output operations are performed using the same instructions which access the memory. The numerous addressing modes provide a versatility for very creative programming. One should be knowledgeable of these elements and concepts before attempting to write machine language programs.

The Internal Registers

The program counter is a sixteen-bit register which is used to direct the flow of a program from one instruction to another. Since the program counter is sixteen bits long, it can directly access instructions in any of the possible 64K bytes of memory. After an instruction is executed, the program counter is automatically incremented to the next location memory from which the next instruction to be executed will be taken. This automatic increment may be overridden if the current instruction directs the computer to a different memory location. In this case, the program counter is loaded with the new address, and a program execution continues with the instruction.

From the software point of view, the accumulator of the 6502 is the real workhorse element. All arithmetic and Boolean logic operations accumulate their results in this register. This eight-bit register, designated by the letter A, is also used for intermediate storage when transferring data from one memory location to another. A number of instructions for shift, rotate and compare also may be performed with the content of the accumulator. The condition of the status flags is affected by almost every operation of the accumulator.

The index registers, designated as X and Y, perform three important functions. First, as their name implies, they are used to form pointers which index into the memory for data storage, retrieval, manipulation and examination. The contents of the index register are added to a base address to allow selection of a successive group of memory locations. This is accomplished simply by incrementing the index register. Since these registers are only eight bits wide, it may appear that the range of the index register is limited to 256. However, as will be discussed in Chapter 2, there are programming techniques to extend this range. Their second function is that of an eight-bit counter register. By incrementing or decrementing these registers with the appropriate instructions, they may be used to count up, or down, keeping track of the number of occurrences of a specific event or, possibly the passage of time. The final function is, as general purpose registers, to transfer data between memory locations and between registers.

The stack pointer is an eight-bit register used to index into page one of the memory for storing and retrieving data on the stack. The stack is the storage area in which the 6502 CPU saves the return addresses of subroutine calls and the pertinent data that must be stored when an interrupt occurs. The data is stored and retrieved from the stack in a push-pull manner. This method is discussed in greater detail later.

External Memory Structure

The memory is the element in which the programs to be executed are stored. It also contains data that may be used by the programs. As mentioned earlier, the 6502 is capable of directly addressing up to 64K of memory. Each memory location consists of eight bits which together are referred to as a byte. The memory associated with any one individual system may vary. It may consist of a combination of ROM and/or PROM memories, which contain permanently stored programs or data. Or it could consist of a

RAM memory whose contents may be altered by the computer for storing various programs or data as needed.

The input/output structure of the 6502 allows the transfer of data to and from the peripheral interfaces by assigning memory addresses to the peripheral. By setting up memory locations as the channels through which data is transferred to and from the peripherals, it is possible to use any of the instructions that refer to the memory for transferring the I/O data. This affords the programmer great flexibility in testing the status and controlling the peripheral devices.

The Status Flags

In order to make decisions based on the contents of a register or memory location, or the results of an arithmetic or logical operation, the 6502 offers four status flags. They are set to one (for a true condition), or cleared to zero (for a false condition), in accordance with the results of an operation performed. Not all status flags are affected by the execution of each instruction. Only those flags that have relevance are affected by an instruction. These status flags are referred to as carry (C), overflow (V), negative (N), and zero (Z). The flag condition may be tested by several instructions. The instructions' operation will vary as a consequence of the flags particular status at the time it is tested.

The carry flag may be considered an extension of the eight-bit accumulator, or a memory location, used as the operand of an instruction. For addition and subtraction operations, the carry is considered the ninth bit and will indicate when an addition causes an overflow from bit seven, or a subtraction requires a borrow for bit seven. By functioning in this manner, the carry flag becomes a necessary link when performing multiple-precision operations. The carry flag is also considered an extension of a register or memory location in various rotate and shift operations. There are a number of instructions that set up the carry to a given condition. This function may be necessary when executing a group of instructions that require the carry to be set initially to a known state.

The negative flag indicates the condition of the most significant bit of a register or memory location following the last instruction that affects the negative flag. If the result leaves the most significant bit set to one, the negative flag will be set to one. If the most significant bit is zero, the negative flag also will be zero. For example, if the contents in a memory location are added to the contents of accumulator A, and this results in the most significant bit

in accumulator A being set to one, the negative flag will be set to one. Or, if a memory location is rotated once to the right, moving a zero into the most significant bit, the negative flag will be cleared to zero as a result of the operation.

The overflow flag provides an indication of a two's complement overflow as a result of an addition or subtraction. For addition, the two's complement overflow occurs when bit seven of both addends is the same value and bit seven of the sum is the opposite value (the addition of two negative numbers equaling a positive value). For subtraction, a two's complement overflow occurs when bit seven of the subtrahend and minuend are opposite, and bit seven of the result takes on the value of bit seven of the subtrahend (the subtraction of a negative from a positive number with a negative result).

The zero flag is set to one when the execution of an instruction results in an all-zero value. This may occur following an arithmetic or Boolean logic operation. It may also occur after an index register or memory location has been incremented or decremented to zero.

Condition Flags

In addition to the status flags, there are also three condition flags which are controlled either by execution of specific instructions or by certain hardware functions. These flags are designated the interrupt disable flag (I), the break flag (B) and the decimal mode flag (D).

The interrupt disable flag is used to indicate when the maskable interrupt input is disabled. When the flag is set to one, the maskable interrupt input is disabled. The CPU will not respond to an interrupt on this line. When this flag is cleared, an interrupt on the maskable interrupt line will be acknowledged by the CPU. This flag is set upon receipt of any one of the three interrupts. Upon returning from the interrupt, it is restored to its initial condition at the time the interrupt was received. It may also be set or cleared by the execution of two instructions that perform these specific functions.

The break flag is used to indicate the execution of a software interrupt. It is set when the break instruction is executed and reset after the status register is stored on the stack as a result of the BREAK instruction. The status register then may be examined to determine whether the interrupt was generated by hardware or software. A more detailed description of the BREAK instruction and flag will be presented later.

The decimal mode flag controls the type of arithmetic addition

or subtraction to be performed. These two types are decimal and binary addition and subtraction. The decimal mode assumes that the numbers to be added or subtracted are in BCD (binary coded decimal) form before the operation. The result is presented in BCD form. The binary mode assumes both values are in binary representation before and after the operation. This flag is set to one for decimal arithmetic and cleared to zero for binary arithmatic by two specific instructions. This flag allows one to write a single group of subroutines to perform both decimal and binary mathematic functions.

Combining All the Flags

These seven flags are arranged in an eight-bit register. The flags are combined so that they may be stored and retrieved easily for interrupt operations.

This register is called the STATUS register. The flags are assigned the following bit locations. One should note that the unused bit (bit 5) either may be set or cleared at any time and therefore should be ignored when working with the status register.

Status Register Bit Definition

Bit 0	–	Carry Flag	C
Bit 1	–	Zero Flag	Z
Bit 2	–	Interrupt Disable	I
Bit 3	–	Decimal Mode	D
Bit 4	–	Break Flag	B
Bit 5	–	Unused	
Bit 6	–	Overflow Flag	V
Bit 7	–	Negative Flag	N

How the Stack Operates

The stack is used to store and retrieve data in the memory locations on page one indicated by the stack pointer. The stack pointer operates in a push-pull manner. Its operation is the same whether the data being stored is (1) a return address from a subroutine call, (2) the return address and status register at the time of an interrupt, (3) the storage or retrieval of the contents of the accumulator. When data is stored in the stack, the data byte is stored in the memory location indicated by the stack pointer. The stack pointer then is automatically decremented. If more than one byte is to be stored, as in the storage of a return address, each additional byte is loaded into the memory. The stack pointer is decremented following each

byte storage. By automatically decrementing the stack pointer in this manner, it is positioned to store more data or read data stored in the stack when either a pull instruction or a return from subroutine, or interrupt, is executed. The following illustrates the method of storing the return address of a subroutine call in the stack. The return address to be stored is location $5E on page 02.

Before Subroutine Call

Stack Pointer	Memory Address of Stack	Stack Contents
$FF	$1FD	$00
	$1FE	$00
	$1FF	$00

After Subroutine Call

Stack Pointer	Memory Address of Stack	Stack Contents
$FD	$1FD	$00
	$1FE	$5E
	$1FF	$02

By performing a return or pull instruction when data is read from the stack, the reverse procedure is followed. That is, the stack pointer is automatically incremented and the data byte is read from the stack. The stack pointer is now positioned for the next stack operation, whether it be to read or write data in the stack.

The Format of Interrupt Operations

The 6502 CPU has provisions for three types of interrupts. Two interrupts are generated by hardware, the third is an interrupt created by a software instruction. The CPU responds to each of these interrupts by storing the return address and the status register in the stack and setting the interrupt disable flag. The CPU then selects the interrupt vector according to the type of interrupt received. This interrupt vector is actually a start address for an interrupt service routine. In most cases, this interrupt service routine begins in ROM memory with several short instructions that fetch another address set up in the RAM memory by the programmer. This second address would be the start of the actual interrupt service routine written to operate the devices associated with one's system.

The first of the hardware interrupts is called the nonmaskable interrupt. This interrupt, when received, will always be acknowl-

edged by the CPU. It is often used by high speed devices that have a very short time to transfer data. Or, it may be used by power-loss detect circuits to allow the CPU time to shutdown critical operations. Also, it can retain current operating status before the power falls to inoperable levels. The nomaskable interrupt is assigned its own interrupt vector.

The other hardware interrupt is called the maskable interrupt. The CPU responds to the receipt of a maskable interrupt by the setting of the interrupt disable flag. As previously discussed, when this flag is reset, the CPU will acknowledge a maskable interrupt. If this flag is set to one, the CPU will ignore this hardware interrupt. This allows the programmer to control when the program can and cannot respond to a maskable interrupt. This interrupt shares its vector with the software interrupt.

A software interrupt is generated by the execution of the software interrupt instruction. The 6502 reacts in the same fashion as it would to a nonmaskable interrupt. However, the software interrupt is not maskable by the interrupt disable flag. It will always vector to the interrupt service routine. Since the maskable and software interrupts share the same vector, it is necessary for the interrupt service routine to examine the contents of the status register stored in the stack to determine which type of interrupt was received. The break flag will be set for a software interrupt.

The use of interrupts in a microcomputer system allows a program to be performing one function while waiting for a peripheral device to complete its operation. For example, a mailing list program could be sorting out names of people living in a specific geographical area, while a printer device, operating under interrupt control, prints the selected names.

There is also a RESET interrupt which is generally used to direct the CPU to a start-up program. The reset is simply an overriding interrupt that halts execution of any program currently running and directs control to a program which may reinitialize the hardware to a known state. A separate vector is assigned for the reset interrupt.

The interrupt vectors are set up in the hardware at the highest addressable locations of the computer (FFFA to FFFF). As discussed, these vectors direct the CPU to specific memory locations when the respective interrupts occur. The page portion of the vector address is in the higher address, and the low portion of the vector is in the lower address of each vector. The vectors are arranged in memory as follows:

Address of Vector	Type of Interrupt
FFFF,FFFE	Software and maskable
FFFD,FFFC	RESET
FFFB,FFFA	Nonmaskable interrupt

Addressing Modes Add Variety

The 6502 instruction set makes extensive use of various ADRESSING modes. These different modes of addressing provide many instructions with up to eight ways of selecting the instructions operand. The addressing mode may refer to the location that contains (or is to receive) data for the instruction execution. Or, it may refer to the location of the next instruction to be executed. The instructions that use these different addressing modes require an additional one or two bytes of memory to be properly defined by the actual machine code.

The first byte of the instruction contains the machine code which indicates the instruction to be executed along with the addressing mode used for that instruction. The information contained in the additional bytes of the instruction would indicate either the actual data to be used as the operand, the location in memory where the data is (or will be) stored, or a relative address. These addressing modes are referred to as immediate, zero page, zero page indexed, absolute, absolute indexed, indexed indirect, indirect indexed and relative.

The source listing of the instructions that use these modes is separated into two fields. The first is called the operator field, and contains the mnemonic for the operation to be performed. The second field is the operand field which will indicate the addressing mode to be used for the instruction. As will be pointed out later, when the individual instructions are presented, the machine code for the same mnemonic will vary depending on the addressing mode selected.

Whenever a numeric value is designated as the operand of the source listing for an instruction, the value will be represented by hexadecimal digits. In order to conform with the generally accepted notation for representing hexadecimal values in the source listing, these values will be preceded by a dollar sign ($). For example, an instruction to load the accumulator from memory location 00A7 will appear as follows: LDA $00A7.

Immediate Addressing Mode

The immediate addressing mode selects the operand from the

memory location following the first byte of the instruction. The instructions that allow the immediate mode of addressing require two bytes. The first byte contains the machine code for the operation to be performed and the second byte contains the immediate data value that will be used. The listings contained in this text have the operand preceded by a pound sign (#) whenever the immediate addressing mode is used. The following example illustrates the execution of the instruction that loads the accumulator with the immediate value of ten (hexadecimal):

Before Execution
Contents of A = XX (don't care)

Instruction Executed
Source code LDA #$10   Machine code $A9 $10

After Execution
Contents of A = $10

Zero Page Addressing Mode

The zero page addressing mode selects the operand of the instruction from a memory location on page 00. This mode requires one additional byte to specify the location on page 00 to be used by the instruction. It is advantageous to use page 00 for the storage of frequently used data. This allows one to access the specific location on page 00 with a two-byte instruction, rather than using an additional byte to specify the page, as in the absolute mode.

The example below illustrates the execution to store the accumulator instruction using the zero-page addressing mode. The instruction in the example stores the contents of the accumulator in memory location 49 (hexadecimal).

Before Execution
Contents of A = $85
Contents of memory location $0049 = XX (don't care)

Instruction Executed
Source code STA $49   Machine code $85 $49

After Execution
Contents of A = $85
Contents of memory location $0049 = $85

Zero-Page Indexed Addressing Mode

The zero-page indexed addressing mode is similar to the zero-

page addressing mode in that the operand refers to a specific location on page 00. However, the actual memory location is selected by adding the contents of the X index register to the operand value. The X index register thus becomes an offset from the location indicated by the operand. One should note two points. First, the Y index is only valid in this mode when loading or storing the page 00. If the sum of the operand plus the index register exceeds $FF, the overflow is ignored and the instruction loops back to the beginning of page 00.

The following example illustrates the execution of ANDing the accumulator with the third entry in a table which begins on page 00, location $50.

Before Execution
Contents of A = $47
Contents of X = $02
Contents of memory location $0050 = $01
Contents of memory location $0051 = $02
Contents of memory location $0052 = $04
Contents of memory location $0053 = $08

Instruction Executed
Source code AND $50,X   Machine code $35 $50

After Execution
Contents of A = $04

Absolute Addressing Mode

The absolute addressing mode uses two additional bytes to define the address of the memory location used as the operand for the instruction. The first of these two bytes contains the lower portion of the memory address; the second contains the page portion. Thus, the absolute mode allows one to directly access any memory location in the system for use as the operand of the instruction. When instructions that allow both absolute and zero-page addressing modes are assembled, the distinction between the two is determined by the page number of the address. If the page number is zero, the zero-page addressing mode should be selected. If the page number is not zero, the absolute addressing mode must be used.

The following example illustrates the execution of the load,

the accumulator with the contents of a memory location using the absolute addressing mode. The contents of memory location $0280 are loaded into the A accumulator.

Before Execution
Contents of A = XX (don't care)
Contents of memory location $0280 = $67

Instruction Executed
Source code LDA $0280   Machine code $AD $80 $02

After Execution
Contents of A = $67
Contents of memory location $0180 = $67

Absolute Indexed Addressing Mode

The absolute indexed addressing mode uses the operand address stored in the two bytes following the machine code for the instruction, and adds the contents of the X or Y index register to determine the actual memory location used by the instruction. The operand is stored with the first byte containing the lower portion of the memory address and the second byte containing the page portion. Unlike the zero-page indexed mode, this mode will cross a page boundary if the sum of the low portion of the operand and the index register is greater than $FF. Note that the X and Y index registers may be used in most instructions that allow absolute indexed addressing.

The following example adds the contents of the memory location following location $0520 to the accumulator, using the Y index register.

Before Execution
Contents of A = $20
Contents of Y = $01
Contents of memory location $0520 = $15
Contents of memory location $0521 = $30
Contents of memory location $0522 = $45
Carry flag is reset

Instruction Executed
Source code ADC $0520,Y   Machine code $79 $20 $05

After Execution
Contents of A = $50

Indirect Addressing Mode

The next three addressing modes utilize a common form of addressing known as indirect addressing. It uses an intermediate storage area to store a pointer. This pointer indicates the actual memory location used with the instruction. The operand of the instruction calls out the location of the intermediate pointer. This indirect method of fetching an operand allows a fixed instruction sequence to operate on numerous memory locations by simply changing the intermediate pointer. These modes used in the 6502 use page zero for storing the intermediate pointer. Therefore, the indirect addressing instructions only require two memory locations: the first to store the machine code for the instruction, and the second to store the location on page zero at which the pointer will be found. The pointer is stored in two consecutive bytes with the low portion of the address stored in the first byte and the page position stored in the second byte.

The indirect addressing mode is used by the JUMP instruction to select the location of the next instruction to be executed. The address stored as the pointer on page zero is moved into the program counter and the program sequence shifts to the routine beginning at this new address.

Indexed Indirect Addressing Mode

The indexed indirect addressing mode uses the X index register to offset the instruction operand. The content of the index register is added to the instruction operand. This value then is used to fetch the pointer on page zero which is in turn used to indicate the memory location operated on by the instruction. This instruction allows one to set up a table of pointers on page zero and, by manipulating the X index register, the desired pointer will be selected. It should be noted that if the sum of the operand plus the X index register is greater than $FF, the result will wrap around to the beginning of page zero.

The following example illustrates the operation of the indexed indirect addressing mode. The accumulator is stored in a memory location which is indexed indirectly through a pointer on page zero.

Before Execution
Contents of A = $55

Contents of X = $02
Contents of memory location $0080 = $24
Contents of memory location $0081 = $05
Contents of memory location $0082 = $22
Contents of memory location $0083 = $05
Contents of memory location $0522 = $XX (don't care)
Contents of memory location $0524 = $XX (don't care)

Instruction Executed
Source code STA ($80,X)   Machine code $81 $80

After Execution
Contents of A = $55
Contents of memory location $0522 = $55
Contents of memory location $0524 = $XX (don't care)

Indirect Indexed Addressing Mode

The indirect indexed addressing mode offsets the value of the pointer selected from page zero by adding the Y index register to it. The instruction operand indicates the location of the pointer on page zero. The contents of the Y index register is added to this pointer to select the actual memory location to be operated on. Thus, a table of as many as 256 entries may be set up in any section of the memory with a pointer to its lowest address stored on page zero. By proper adjustment of the Y index register, any desired entry in the table may be selected. This method is illustrated below. This example loads the accumulator with the second entry of a table beginning at location $0400.

Before Execution
Contents of A = XX
Contents of Y = $01
Contents of memory location $0090 = $00
Contents of memory location $0091 = $04
Contents of memory location $0400 = $B1
Contents of memory location $0401 = $B2
Contents of memory location $0402 = $B3

Instruction Executed
Source code LDA ($90),Y   Machine code $B1 $90

After Execution
Contents of A = $B2

Relative Addressing Mode

The relative addressing mode references a memory location relative to the current value of the program counter +2. The relative addressing mode is used exclusively by the branch instructions. Two bytes are required to define the branch instruction. The first byte of the branch instruction calls out which conditional branch is to be executed. The second byte contains the relative displacement in two's complement form. Branching to a memory location is calculated by simply adding the second byte to the value of the program counter +2. If the most significant bit is a one, the branch will be to an address lower than the current program counter +2. A value of zero for the most significant bit indicates a branch to a higher address. The two's complement notation limits the branch instructions to a displacement of −128 to +127 locations from the value of the program counter +2.

If the zero flag is set, the following example illustrates a branch back to the instruction located $0E hexadecimal locations before the branch instruction.

Before Execution
Program counter = $0270
(Location of first machine code of branch)

Instruction Executed
Source code BEQ $F0 Machine code $F0 $F0

After Execution
Program counter = $0262

Described here are the various types of instructions available with the 6502 CPU and will provide the mnemonic name used for writing programs in symbolic language. The machine code for the instruction is given as two hexadecimal digits. In cases where the mnemonic allows more than one addressing mode, the additional machine codes are listed, followed by an indication of the addressing mode to which they relate. Appendix A contains a list of these mnemonics and machine codes in alphabetical order. These mnemonics are equivalent to those defined by MOSTEK. Information concerning the timing for the instructions is also included.

The use of mnemonics facilitates working with an assembler program when developing relatively large and complex programs.

Thus, the programmer is urged to concentrate on learning the mnemonics for the instructions, and not to memorize the machine codes. After a program has been written using the mnemonics, the programmer can use a lookup table for conversion to machine code if an assembler program is not available.

The following discussion of the 6502 instruction set is preceded by the mnemonics and machine code in either two or three columns. The first column contains the mnemonic representation of the instruction. The second column contains the machine code for that mnemonic. In cases where several addressing modes are possible, the third column indicates the addressing mode for the machine code.

The first group of instructions loads data from the accumulator to the memory, and vice versa. These instructions require one to three bytes of memory.

Load the Accumulator from Memory

LDA	#DATA	$A9	IMMEDIATE
LDA	ADDR	$A5	ZERO PAGE
LDA	ADDR,X	$B5	ZERO PAGE INDEXED
LDA	ADDR	$AD	ABSOLUTE
LDA	ADDR,X	$BD	ABSOLUTE INDEXED
LDA	ADDR,Y	$B9	ABSOLUTE INDEXED
LDA	(ADDR,X)	$A1	INDEXED INDIRECT
LDA	(ADDR),Y	$B1	INDIRECT INDEXED

This group of instructions loads the accumulator with the content of the memory location indicated by the addressing mode. For the immediate mode, the instruction requires two bytes and the data to be loaded into the accumulator is taken from the second byte of the instruction. For the zero-page modes, the instruction requires two bytes, with the second byte indicating the location onpage 00 from which the data is to be taken and loaded into the accumulator. The second and third bytes of the three-byte absolute mode instruction contain the low and page portion of the address from which the data to be loaded is taken. The indirect modes require two bytes. The second byte indicates the location on page zero containing the indirect pointer. The N and Z flags are affected as a result of these instructions. The C, I, D and V flags remain unchanged.

Store Accumulator in Memory

STA	ADDR	$85	ZERO PAGE
STA	ADDR,X	$95	ZERO PAGE INDEXED

STA	ADDR	$8D	ABSOLUTE
STA	ADDR,X	$9D	ABSOLUTE INDEXED
STA	ADDR,Y	$99	ABSOLUTE INDEXED
STA	(ADDR,X)	$81	INDEXED INDIRECT
STA	(ADDR),Y	$91	INDIRECT INDEXED

Storing data contained in the accumulator to a memory location is accomplished by the execution of one of these instructions. The exact location in memory is determined by the addressing mode used. The immediate mode is not valid for this instruction. The zero page and indirect modes require two bytes, and the absolute modes require three. The status flags are affected in a similar manner as loading the accumulator from the memory instructions.

PUSH the Accumulator onto the Stack

PHA $48

This instruction stores the contents of the accumulator into the memory location indicated by the stack pointer. After storing the data, the stack pointer is automatically decremented to the proper position for the next stack operation. This one-byte instruction provides a convenient method for temporarily storing the contents of the accumulator without designating a specific memory location for its storage. None of the status flags are affected.

PULL Data from the Stack into the Accumulator

PLA $68

Execution of this instruction first increments the stack pointer, and then transfers the data in the memory location indicated by the stack pointer to the designated accumulator. This instruction is used in conjunction with the push instruction to retrieve data pushed onto the stack. The status flags are not affected.

The next section contains instructions that deal with the loading, storing, and manipulation of the index registers contents and stack pointer. Proper manipulation of these registers is essential in programming the 6502 efficiently. The number of bytes required for this group of instructions varies from one to three.

Load the Index Registers

LDX	#DATA	$A2	IMMEDIATE
LDX	ADDR	$A6	ZERO PAGE

LDX	ADDR,Y	$B6	ZERO PAGE INDEXED
LDX	ADDR	$AE	ABSOLUTE
LDX	ADDR,Y	$BE	ABSOLUTE INDEXED
LDY	#DATA	$A0	IMMEDIATE
LDY	ADDR	$A4	ZERO PAGE
LDY	ADDR,X	$B4	ZERO PAGE INDEXED
LDY	ADDR	$AC	ABSOLUTE
LDY	ADDR,X	$BC	ABSOLUTE INDEXED

This group of instructions load the designated index register from the memory location defined by the respective addressing modes. The immediate and zero page instructions require two bytes of memory and the absolute addressing mode requires three bytes. An index register is not used to load itself when an indexed addressing mode is called out. The resultant contents of the index register affect the N and Z flags, and the C, I, D and V flags are left unchanged.

Store the Index Registers

STX	ADDR	$86	ZERO PAGE
STX	ADDR,Y	$96	ZERO PAGE INDEXED
STX	ADDR	$8E	ABSOLUTE
STY	ADDR	$84	ZERO PAGE
STY	ADDR,X	$94	ZERO PAGE INDEXED
STY	ADDR	$8C	ABSOLUTE

Storing the contents of the designated index register is accomplished by the execution of one of these instructions. The contents of the index register remain unchanged. The zero-page addressing modes require two bytes of memory and the absolute mode requires three bytes. The flags are affected in the same manner as with the load index register instructions.

Increment the Index Register

INX	$E8
INY	$C8

These one byte instructions increment the designated index register by one. By using the index registers as part of a pointer, via an indexed addressing mode, this instruction is used to advance the pointer from one location to the next. The N and Z flags are affected while the C, I, D and V flags remain unchanged.

Decrement the Index Registers

DEX	$CA
DEY	$88

These instructions perform the opposite function of increment instructions. The contents of the designated index register is decremented by one. The N and Z flags reflect the result of the operation while the C, I, D and V flags are unchanged.

Transfer from Accumulator to Index Register

TAX	$AA
TAY	$A8

The current contents of the accumulator are transferred to the designated index register. This is a convenient one byte instruction for the temporary storage of the accumulator. The N and Z flags are affected by these instructions while the C, I, D and V flags and the contents of the accumulator remain unchanged.

Transfer from Index Register to Accumulator

TXA	$8A
TYA	$98

The contents of the designated index register are transferred to the accumulator. This may be performed to allow arithmetic or logical operations on the contents of the index register which can only be executed in the accumulator. As in the previous transfer instructions only the N and Z flags are affected by these one byte instructions.

Transfer from Stack Pointer to the X Index Register

TSX	$BA

This one-byte instruction transfers the contents of the stack pointer to the X index register. The stack pointer maintains its initial contents following the execution. By loading the X index register with the address contained in the stack pointer, the absolute indexed addressing mode instructions may be used to store data on the stack while in a subroutine. Also, by incrementing the X index register following this instruction, an indexed pointer is set to examine and/or change the return address of a subroutine call. The N and Z flags are the only flags affected.

Transfer from X Index Register to Stack Pointer
TXS $9A

This instruction transfers the contents of the X index register into the stack pointer. This one byte instruction is used to initialize the stack pointer at the start of a program. It may also be used to move the stack pointer to a new location in the stack, with the intent, possibly, of skipping a return address or some data stored on the stack. None of the status flags are affected by this instruction.

These instructions listed above describe the transfer of data between internal CPU registers, and a CPU register and a memory location. Several instructions that allow the manipulation of data within the CPU registers have also been discussed. The 6502 provides a similar type manipulation of memory contents. These instructions utilize the zero page and absolute modes of addressing, and require two or three bytes of memory.

Increment the Memory Location

INC	ADDR	$E6	ZERO PAGE
INC	ADDR,X	$F6	ZERO PAGE INDEXED
INC	ADDR	$EE	ABSOLUTE
INC	ADDR,X	$FE	ABSOLUTE INDEXED

The designated memory location is incremented by one. The absolute addressing mode requires three bytes of memory, and the zero page requires two. This makes it convenient to set up a memory location as a pointer. Only the X index register is used in the indexed form. Only the N and Z flags are affected by the execution.

Decrement the Memory Location

DEC	ADDR	$C6	ZERO PAGE
DEC	ADDR,X	$D6	ZERO PAGE INDEXED
DEC	ADDR	$CE	ABSOLUTE
DEC	ADDR,X	$DE	ABSOLUTE INDEXED

These instructions decrement the contents of the designated memory location by one. The X index register is used exclusively by the indexed form. As in the increment memory instructions, the zero page addressing mode requires two bytes and the absolute mode requires three bytes. The N and Z flags are conditioned to indicate the result and the C, I, D and V flags are left unchanged.

The following group of instructions allows the programmer to

direct the computer to perform arithmetic operations between the accumulator and the designated memory location. Also, there is a pair of instructions that control whether the arithmetic assumes binnary or BCD digits in the accumulator and memory location. The instructions in this group that use the immediate, zero page and indirect addressing modes require two bytes, and the absolute requires three.

Set the Decimal Mode
SED $F8H

Addition and subtraction of two bytes in a computer normally assumes that the contents of the bytes are eight-bit binary values. However, this instruction allows one to store the data to be added or subtracted as BCD digits. A BCD digit is a four-bit binary number within the range of zero to nine. The six binary values above nine are invalid. This instruction sets the decimal mode flag. As long as this flag remains set, the execution of the addition and subtraction instructions assumes that the accumulator and memory location used contain two BCD digits. The result of the arithmetic operation leaves two BCD digits in the accumulator. This one-byte instruction affects only the decimal mode flag.

Clear the Decimal Mode
CLD $D8

All addition and subtraction instruction executed when the decimal mode flag is cleared assumes the data to be in binary form. Only the decimal mode flag is affected.

Add the Contents of Memory Plus the Carry Flag to the Accumulator

ADC	#DATA	$69	IMMEDIATE
ADC	ADDR	$65	ZERO PAGE
ADC	ADDR,X	$75	ZERO PAGE INDEXED
ADC	ADDR	$6D	ABSOLUTE
ADC	ADDR,X	$7D	ABSOLUTE INDEXED
ADC	ADDR,Y	$79	ABSOLUTE INDEXED
ADC	(ADDR,X)	$61	INDEXED INDIRECT
ADC	(ADDR),Y	$71	INDIRECT INDEXED

These instructions add the contents of the designated memory

location to the accumulator. The carry flag is also added to the least significant bit of the accumulator. The result of the addition is left in the accumulator in the format dictated by the decimal mode flag. The carry flag is the link between bytes when adding two multiple precision values. The N, Z and V flags are also updated to indicate the result of the addition. The contents of the memory location used are not changed.

Subtract the Memory Contents and the Carry Flag from the Accumulator

SBC	#DATA	$E9	IMMEDIATE
SBC	ADDR	$E5	ZERO PAGE
SBC	ADDR,X	$F5	ZERO PAGE INDEXED
SBC	ADDR	$ED	ABSOLUTE
SBC	ADDR,X	$FD	ABSOLUTE INDEXED
SBC	ADDR,Y	$F9	ABSOLUTE INDEXED
SBC	(ADDR,X)	$E1	INDEXED INDIRECT
SBC	(ADDR),Y	$F1	INDIRECT INDEXED

The contents of the memory location and the carry flag are subtracted from the accumulator. The result of the subtraction either in binary or BCD, is stored in the accumulator and the carry flag will be reset if a borrow was required for the subtraction of the most significant bits. The N, Z and V flags are also affected by these instructions.

There is a group of instructions that perform a subtraction operation without altering the contents of any CPU registers or memory locations. However, the results of the subtraction operation are indicated by the condition of several of the status flags. The purpose of these instructions is to allow the program to compare the contents of the accumulator or index register to a value in memory.

The following group of compare instructions is very powerful and somewhat unique. They direct the computer to compare the contents of the designated accumulator or index register against the contents of the memory, and set the status flags as a result of the compare operation. Essentially it is a subtraction operation, with the value in the memory being subtracted from the value in the accumulator or index register. The value in the accumulator or index register is not altered by the operation. However, the flags are set in the same manner as though an actual subtraction operation had occurred. Subsequently, by testing the status of the various flags after a compare instruction is executed, the program can determine

whether the compare operation resulted in a match or not. The flags will indicate the relative magnitude of the two values with respect to each other.

These various tests are accomplished by utilizing the conditional branch instructions (to be described later). Unlike the SBC instructions, the carry flag is not included in the subtraction.

Compare the Contents of the Memory to the Accumulator

CMP	#DATA	$C9	IMMEDIATE
CMP	ADDR	$C5	ZERO PAGE
CMP	ADDR,X	$D5	ZERO PAGE INDEXED
CMP	ADDR	$CD	ABSOLUTE
CMP	ADDR,X	$DD	ABSOLUTE INDEXED
CMP	ADDR,Y	$D9	ABSOLUTE INDEXED
CMP	(ADDR,X)	$C1	INDEXED INDIRECT
CMP	(ADDR),Y	$D1	INDIRECT INDEXED

This group of compare instructions compares the content of the designated memory location to the content of the accumulator and requires two bytes for the immediate, zero page and indirect addressing modes, and three bytes for the absolute mode. The C, N and Z flags are conditioned according to the results of the subtraction operation. The V flag is not changed.

Compare the Contents of the Memory to the Index Register

CPX	#DATA	$E0	IMMEDIATE
CPX	ADDR	$E4	ZERO PAGE
CPX	ADDR	$EC	ABSOLUTE
CPY	#DATA	$C0	IMMEDIATE
CPY	ADDR	$C4	ZERO PAGE
CPY	ADDR	$CC	ABSOLUTE

These instructions compare the contents of the designated index register with the memory location. The contents of the indicated memory location is subtracted from the index register. The C, N and Z flags are affected by the result of the subtraction. However, the V flag, memory location and index register remain unchanged. The immediate and zero-page addressing mode instructions require two bytes and the absolute mode instructions require three. These instructions are useful in testing for the end of a table pointed to by the index register.

There are several groups of instructions that allow Boolean logic

operations to be performed between the contents of locations in the memory and the accumulator. Boolean logic operations are valuable in a number of programming applications. The 6502 instruction set allows three basic Boolean operations to be performed. These are the logical AND, logical OR, and EXCLUSIVE OR operations. Each type of logic operation is performed on a bit-by-bit basis between the memory location and the accumulator specified by the instruction.

These instructions utilize four basic addressing modes to define the memory location to be used. For this entire group, the immediate, zero page and indirect mode instructions require two bytes of memory, while the absolute mode requires three.

"AND" the Accumulator

AND	#DATA	$29	IMMEDIATE
AND	ADDR	$25	ZERO PAGE
AND	ADDR,X	$35	ZERO PAGE INDEXED
AND	ADDR	$2D	ABSOLUTE
AND	ADDR,X	$3D	ABSOLUTE INDEXED
AND	ADDR,Y	$39	ABSOLUTE INDEXED
AND	(ADDR,X)	$21	INDEXED INDIRECT
AND	(ADDR),Y	$31	INDIRECT INDEXED

When the Boolean AND instruction is executed, each bit of the accumulator will be compared with the corresponding bit in the memory location specified by the instruction. As each bit is compared, a logic result will be placed in the accumulator. The logic result is determined as follows: If both the bit in the accumulator and the bit in the memory location with which the operation is being performed are a "1," the accumulator bit will be left as a "1." For other possible combinations (i.e., the accumulator bit = "0," and the memory location bit = "1," or if the accumulator bit = "1" and the memory contents bit = "0," or if both the accumulator and the memory contents have the particular bit = "0"), the accumulator bit will be set to "0." An example will illustrate the logical AND operation:

Initial State of the Accumulator:	10101010
Contents of Memory Location:	11001101
Final State of the Accumulator:	10001000

The eight logical AND instructions perform this type of logic

operation between the accumulator and memory location, the result of the operation is stored in the accumulator. The N and Z flags are affected by the results of the logical AND operation. C and V flags are not affected.

Logical "OR" the Accumulator

ORA	#DATA	$09	IMMEDIATE
ORA	ADDR	$05	ZERO PAGE
ORA	ADDR,X	$15	ZERO PAGE INDEXED
ORA	ADDR	$0D	ABSOLUTE
ORA	ADDR,X	$1D	ABSOLUTE INDEXED
ORA	ADDR,Y	$19	ABSOLUTE INDEXED
ORA	(ADDR,X)	$01	INDEXED INDIRECT
ORA	(ADDR),Y	$11	INDIRECT INDEXED

This group of Boolean logic instructions direct the computer to perform the logical OR operation on a bit-by-bit basis with the designated accumulator and contents of the memory location. The logical OR operation will result in the accumulator having a bit set to "1" if either the bit in the accumulator, or the corresponding bit in the memory location is a "1." Since the case where both the accumulator bit and the operand bit is a "1" also satisfies the relationship, that condition will also result in the accumulator bit being a "1." If neither accumulator nor memory location has a "1" in the bit position, the accumulator bit remains "0." An example illustrates the results of the logical OR operation:

Initial State of the Accumulator:	10101010
Contents of the Operand Register:	11001101
Final State of the Accumulator:	11101111

The logical OR instructions listed here perform this operation between the accumulator and memory location. The execution of these instructions leaves the result in the accumulator. The effect on the status flags is the same as for the logical AND instructions.

Logical "EXCLUSIVE OR" the Accumulator

EOR	#DATA	$49	IMMEDIATE
EOR	ADDR	$45	ZERO PAGE
EOR	ADDR,X	$55	ZERO PAGE INDEXED
EOR	ADDR	$4D	ABSOLUTE
EOR	ADDR,X	$5D	ABSOLUTE INDEXED

EOR	ADDR,Y	$59	ABSOLUTE INDEXED
EOR	(ADDR,X)	$41	INDEXED INDIRECT
EOR	(ADDR),Y	$51	INDIRECT INDEXED

This group of Boolean logic instructions is a variation of the logic OR. The variation is termed the logical EXCLUSIVE OR. The EXCLUSIVE OR operation is similar to the OR, except that when the corresponding bits in both accumulator and the operand register are a "1," the accumulator bit will be set to "0." Thus, the accumulator bit will be a "1" after the operation only if one of the registers has a "1" in the bit position. An example provides clarification:

Initial State of the Accumulator:	10101010
Contents of the Operand Register:	11001101
Final State of the Accumulator:	01100111

These logical EXCLUSIVE OR instructions, similar to those for the AND and OR, perform the operation between the accumulator and memory location with the results being stored in the accumulator. The status flags are also affected, or not affected, in the same manner as the logical AND instructions.

BIT Test Memory with the Accumulator

BIT	ADDR	$24	ZERO PAGE
BIT	ADDR	$2C	ABSOLUTE

The BIT test instruction tests one or more bits in a memory location without altering the contents of the memory location. This is accomplished by performing a logic AND between the accumulator and the memory location. Although neither alter their contents, the Z flag will indicate whether one or more common bit positions contain a "1." Testing for the condition of a particular bit is done by loading the accumulator with zeros in all bits except the one to be tested. This bit is loaded with a one. Executing BIT would set the Z flag to one if the bit in memory is zero, or clear the Z flag if it is one. The condition of bit 7 and bit 6 of the memory location is loaded directly into flags N and V respectively. This is done independently of the logic AND operation. Thus, one may test these two bits with the BIT instructions, without initializing the accumulator. The C flag is not affected. The zero page addressing mode requires two bytes to define the operation and the absolute mode requires three.

The 6502 has a group of instructions that allow the programmer to condition several of the status flags individually. The status register also may be stored and retrieved from the stack. All of the instructions in this group require only one byte of memory. The instructions that refer to an individual flag affect only that flag. All other status flags remain in their initial condition.

Set the Carry Flag
SEC $38

This instruction sets the carry flag to a value of "1," and the "clear the carry" instruction presented next, provides a convenient method for conditioning the carry flag before an arithmetic or rotate instruction.

Clear the Carry Flag
CLC $18

This instruction clears the carry flag by loading a "0."

Set the Interrupt Flag
SEI $78

The interrupt flag is set to a "1" by this execution. It may be considered a disable interrupt instruction since the interrupt flag disables the CPU from accepting maskable interrupts while it is set to a "1."

Clear the Interrupt Flag
CLI $58

This instruction clears the interrupt flag to a "0" condition. Clearing the interrupt flag allows the CPU to accept interrupts from the maskable interrupt line.

Interrupt flag instructions provide the programmer with a means of control when the computer may accept interrupts on the maskable interrupt line. The function of these two instructions is performed automatically when an interrupt is received. The computer automatically sets the I flag. Then, upon execution of the "return-from-interrupt" instruction (to be presented later), the I flag is returned to its initial state. Also, there may be times in a program when an operation to be performed affects data critical to the exe-

cution of the interrupt service routine. Before performing this operation, the interrupt flag should be set so that a maskable interrupt will not be accepted while the data is being changed. Once the program has completed this operation, the flag may be cleared to allow interrupts to be received.

Clear the Overflow Flag
CLV $B8

This instruction clears the two's complement overflow flag to a "0" and is useful in performing signed binary arithmetic operations.

PUSH Status Register onto Stack
PHP $08

Occasionally, it is desired to save the current status flag settings. For example, a routine may determine that a value is negative. However, this information is not required by the program until other parameters are tested. This one-byte instruction may be used to store the status register on the stack. Then, when the program is ready to make a decision based on the sign of the aforementioned value, the status can be retrieved from the stack by the pull status instruction. Pushing the status register onto the stack stores the status in the location indicated by the stack pointer at the time of execution. Then the stack pointer is decremented. The contents of the status register is not affected.

PULL Status Register from Stack
PLP $28

This one byte instruction causes the stack pointer to be incremented and the data on the stack at this location to be loaded into the status register. This is one method of restoring the status to a previously determined condition. The PHP and PLP instructions are also a convenient method of storing and restoring the decimal mode flag when calling an arithmetic routine that may change its setting.

It is often desirable to be able to shift the contents of an accumulator or memory location either right or left. In a fixed length register, a simple shift operation would result in some information being shifted right out of the register! Therefore, instead of losing

this information, the carry flag is used as an extension of the accumulator or memory location. The carry will "catch" the bit being shifted out of either the LSB for a shift to the right, or the MSB for a shift to the left.

When performing these shift operations, the condition of the bit being shifted into the register must also be considered. Depending on the application of the shifting operation, it may be desired to shift a zero, or to shift the initial contents of the carry flag, into this bit. The shifting operation that shifts the carry around to the opposite end of the register is termed a "rotate" operation. The initial contents of the entire register and the carry are never lost. it is shifted out one end into the carry, and from the carry back into the other end of the register.

The 6502 CPU provides four various shifting and rotating operations that may use either the accumulator or a memory location as the register to be shifted. A description of the shift and rotate operations available are presented here. Those designating an accumulator require one byte, those using the zero page addressing mode require two bytes, and those indicating the absolute addressing mode require three bytes. Only the X index register is valid for the indexed addressing modes.

Arithmetic Shift Left

ASL	A	$0A	
ASL	ADDR	$06	ZERO PAGE
ASL	ADDR,X	$16	ZERO PAGE INDEXED
ASL	ADDR	$0E	ABSOLUTE
ASL	ADDR,X	$1E	ABSOLUTE INDEXED

The arithmetic shift left operation shifts either the designated accumulator or memory location to the left one bit. The MSB is shifted into the carry and a zero is shifted into the LSB. This operation multiplies the initial contents of the register by two. For multiple precision operations, this instruction may be used to shift the least significant byte, and the successive bytes may be shifted by using the rotate left instruction, to be described shortly. By starting with this instruction, initially it is not necessary to clear the carry flag. The C, N and Z flags are affected by this operation. The V flag is not.

Logical Shift Right

LSR	A	$4A

LSR	ADDR	$46	ZERO PAGE
LSR	ADDR,X	$56	ZERO PAGE INDEXED
LSR	ADDR	$4E	ABSOLUTE
LSR	ADDR,X	$5E	ABSOLUTE INDEXED

The logical shift right instruction shifts the designated register to the right one bit. Bit zero is loaded into the carry flag, and bit seven is loaded with a zero. This instruction is used to divide the contents of the register by two when the MSB is assumed to be part of the value and not the sign of the value. The C, N and Z flags are affected by the result of this operation but the V flag is not.

Rotate Left

ROL	A	$2A	
ROL	ADDR	$26	ZERO PAGE
ROL	ADDR,X	$36	ZERO PAGE INDEXED
ROL	ADDR	$2E	ABSOLUTE
ROL	ADDR,X	$3E	ABSOLUTE INDEXED

The designated accumulator or memory location is rotated one bit to the left by the execution of this instruction. The MSB is rotated into the carry, and the initial content of the carry is rotated into the LSB. Since this instruction forms a closed loop, it does not lose the contents of any of the bits. It may, therefore, be used to calculate the parity of the value in the register by rotating each bit into the carry and adding up the number of ones contained in the register. Rotating a multiple precision value to the left may be accomplished by initially clearing the carry and then, beginning with the least significant byte, rotating each byte once to the left. In doing so, it is essential that the instructions in between each rotate do not affect the carry flag. The status flags are affected in the same manner as with the arithmetic shift left.

Rotate Right

ROR	A	$6A	
ROR	ADDR	$66	ZERO PAGE
ROR	ADDR,X	$76	ZERO PAGE INDEXED
ROR	ADDR	$6E	ABSOLUTE
ROR	ADDR,X	$7E	ABSOLUTE INDEXED

The rotate right instruction rotates the designated accumulator

or memory location once to the right with the LSB rotated into the carry, and the initial contents of the carry rotated into the MSB. The parity of the register contents also may be checked by a series of rotate right instructions. Dividing a multiple precision value by two may be accomplished initially by clearing the carry and then, beginning with the most significant byte and working down to the least significant byte, each byte of the multiple precision value is rotated once to the right. Here again, the instructions in between rotate instructions must not affect the carry. The status flags are affected in the same manner as with the arithmetic shift left instruction.

The No Operation Instruction

NOP $EA

The no operation, or NOP, instruction directs the computer to consume time by executing a machine cycle that effectively does nothing except advance the program counter to the next memory address. None of the CPU registers are affected by the operation. The instruction is useful for creating time delays, or as a filler if patches to a program are required (or anticipated).

The instructions discussed so far have been direct action ones. The programmer arranged a sequence of these instructions in memory. When the program is started, the computer proceeds to execute the instructions in the order in which they are encountered. The computer automatically reads the contents of the memory location and executes the instruction it finds there. Then it automatically increments a special address register called a "program counter" to the next sequential memory location. Often it is desirable to perform a series of instructions located in one section of the memory and then skip over a group of memory locations to start executing instructions in another section. This action can be accomplished by a group of instructions that will cause the CPU to jump to a new section of the memory and continue executing instructions sequentially from the new memory location.

There are a series of conditional branch instructions available in this computer that add considerable power to the machine's capabilities. The computer can be directed to test the status of a particular flag. If the status of the flag is the desired one, a branch will be performed. If it is not, the computer will continue to execute the next instruction in the current sequence. This capability provides a means for the computer to make decisions, and to modify its op-

eration as a function of flag status.

All of the branch instructions use the relative addressing mode to define the memory location from which the next instruction to be executed is to be taken. This mode of addressing requires two bytes of memory to properly define the instruction. The first byte contains the machine code for the type of branch instruction to be executed. The second byte contains the relative displacement, in two's complement form, from the memory location following the second byte of the branch instruction. Refer to the beginning of this chapter to review the relative addressing mode if necessary. These branch instructions do not affect any of the status flags.

The following is a list of branch instructions. Each tests a single flag to determine whether to branch, or to fall through to the next sequential instruction. The first column contains the mnemonic representation for the instruction, the second column contains the machine code for the first byte of the instruction, and the final column indicates the flag tested and the condition that would cause the instruction to branch.

BCC	RELA	$90	C=0
BCS	RELA	$B0	C=1
BNE	RELA	$D0	Z=0
BEQ	RELA	$F0	Z=1
BPL	RELA	$10	N=0
BMI	RELA	$30	N=1
BVC	RELA	$50	V=0
BVS	RELA	$70	V=1

The Jump Instruction

JMP	ADDR	$4C	ABSOLUTE
JMP	(ADDR)	$6C	INDIRECT

The jump instruction always results in the computer going to the designated address rather than fetching the next instruction from the current sequence. However, the jump instruction is not limited to an area in the memory relative to its current location. Using either of the addressing modes indicated, the jump instruction can direct the computer to any location throughout its memory. For the three-byte absolute addressing mode instruction, the second byte contains the low portion, and the third byte contains the page portion of the address to which the computer is to jump. For the indirect mode, the operand points to the location where the

actual address to jump may be found. The indirect pointer points to the low portion of the address and the next successive memory location contains the page portion. These jump instructions do not affect any of the status flags.

Quite often, when a programmer is developing computer programs, he will find that a particular algorithm can be used many times in different parts of the program. Rather than entering the same sequence of instructions at different locations in the memory (which would not only consume the time of the programmer but would also result in a lot of memory being used to perform the same function), it is desirable to be able to put an often used sequence of commands in one section of the memory. Then, whenever this particular algorithm is required, it would be convenient to jump to the section that contained it and perform the sequence of instructions, before returning to the main part of the program. This is a standard practice in computer operations. The algorithm can be designated as a subroutine. A special instruction allows the programmer to call a subroutine. A second type of instruction is used to terminate the sequence of instructions. This special terminator will cause the program operation to revert back to the next sequential location in the memory.

When a jump-to-subroutine instruction is executed, the CPU will save the address of the last byte of the instruction call by storing it in the stack. The address in the program counter is advanced to the last byte of the subroutine call instruction. The low portion of this address then is stored in the stack indicated by the stack pointer. The stack pointer is decremented by one, and the page portion of the address is stored in the stack. Finally, the stack pointer is then decremented once more to position it for the next operation.

The return instruction that terminates a subroutine requires only one byte. When the CPU encounters a return instruction, it causes the address stored in the stack to be pulled off into the program counter. The program counter is then incremented and the instruction following the jump to subroutine is executed. The low and then page portions of the address are each pulled from the stack in the same manner that a value is pulled from the stack and loaded into an accumulator.

Jump to Subroutine

JSR	ADDR	$20	ABSOLUTE

This three-byte instruction directs program execution to the

address indicated by the operand. The second byte contains the low portion and the third byte contains the page portion of the subroutine's start address. This instruction does not affect the status flags.

Return from Subroutine
RTS $60

This one byte instruction returns program execution from a subroutine to the calling program. The return address is pulled from the stack and loaded into the program counter. The program counter is then incremented and the next instruction in the initial program sequence is executed. Since no status flags are affected by the return, the result of the subroutine's operation may be passed to the calling program through the flags.

This final group of instructions deals with the software portion of interrupt operations. These instructions, along with the interrupt flag set and clear instructions presented previously, provide the 6502 with the necessary software capability to operate under interrupt control.

Break – a Software Interrupt
BRK $00

Execution of this one-byte instruction causes the 6502 to respond in a manner similar to the receipt of a hardware interrupt. The address of the BRK instruction plus two is pushed into the stack, followed by the status register. The break flag, bit four of the status register, will be set when it is stored as an indication to the interrupt handler that the interrupt is software generated. The program counter then is loaded with the interrupt vector at locations $FFFF and $FFFE. This vector is the start address of the maskable interrupt routine. At this time, any hardware interrupts that occur on the maskable interrupt line will be ignored since the interrupt disable flag is also set by this instruction.

Return from Interrupt
RTI $40

This one-byte instruction is used at the completion of an interrupt service routine to automatically restore the flags in the status register to their initial values at the time the interrupt was received. The return address is then pulled from the stack. Execution of the

program resumes at the instruction following the last one executed before the interrupt.

Information on Instruction Execution Times

When programming for real-time applications, it is important to know how much time each instruction requires for execution. With this information, the programmer can develop timing loops or determine with substantial accuracy how much time it takes to perform a particular series of instructions. This information is important when dealing with programs that control the operation of external devices which require events to occur at specific times. Along with the list of mnemonics and machine codes, Appendix A provides the nominal instruction execution time for each instruction used in a 6502 system. The table shows the number of cycle states required by the instruction. Since the nominal cycle time for a 6502 microcomputer is one microsecond, the number of cycle states translates directly into the execution time for each instruction. In some cases, however, the cycle time of a 6502 system may be slowed down to allow the use of slower memory. To calculate the execution time of an instruction, multiply the number of cycle states for the instruction by the time required for one cycle. Knowing the exact time required by the CPU to execute each instruction allows algorithms to have specific events occur at precisely timed intervals. This concept is discussed in greater detail in Chapter Three on programmed time delays.

Chapter 2

6502 Programming Techniques

Creativity in programming is what makes the difference between so-so programmers and efficient programmers. Proper selection of the instructions available to perform a given task can be of substantial importance. When memory size and execution times are critical, the technique used must be concise and free of extraneous operations.

The flexibility inherent in the 6502 instruction set allows one to become extremely creative. The structure of the 6502 instruction set provides a variety of techniques to accomplish a given task. Different methods for storing and retrieving data, altering instruction execution sequences and controlling various other functions of the CPU are among its many attributes. Proper selection and utilization of these techniques can shorten both memory requirements and execution time for a given program.

If Page Zero Were Only Bigger!

Although the 6502 is capable of directly addressing 64K of memory, the instruction set places special significance on the lowest 256 bytes. This portion of the memory is referred to as page zero. The reason for the significance of using page zero is the presence of the zero page addressing mode.

The zero-page addressing mode allows one to directly reference a location on page zero. This is accomplished by specifying only the eight least significant bits of the address, rather than an entire sixteen-bit address. The importance of this addressing mode is that only one byte is required to specify the memory address rather than two bytes as in the absolute mode. For example, an instruction to store the accumulator contents in location $10 on page 00 requires

only two bytes. One byte is used to indicate a store-A-zero-page instruction (machine code $85), and the second byte to indicate the page zero location where data is to be stored ($10). If the same data were to be stored in location $10 on page $02, the absolute mode version of the STA instruction would be required. This absolute mode instruction would use three memory locations: One location for the machine code ($8D) and two more to designate the address ($10 and $02). This one byte difference between the two modes may not appear to be a significant savings, however it does add up as the length of the program increases.

Often, programs are written which require a large amount of data storage. Pointers, counters and data buffers are typical of such information used by a program. Data storage location can be an important factor in program efficiency. The short one- or two-byte data values should be stored on page zero. Also, the data which is most frequently referenced should have priority over the temporary data which may be called out only a few times. Long strings of data are effectively referenced by storing a base pointer on page zero and using one of the indirect addressing modes to access it. One must be careful in the use of page zero to minimize memory requirements.

Storing data in the stack can help ease the burden on page zero. Occasionally a single byte of data is generated at one point in a routine, but not needed until several intermediate steps are executed. A location on page zero could be set up to hold this data. Or, one could push this data onto the stack by using the PHA instruction. Thus, the stack would provide a temporary holding register for the data. When the routine is ready, the stored data may be pulled from the stack by the PLA instruction. In this way, the necessity of using page zero for temporary data storage is alleviated.

A word of caution: Don't try to push data onto the stack before calling a subroutine which is to use it. When the subroutine attempts to pull the data, it will end up with the low portion of its return address, rather than the data it is expecting. Remember, the subroutine call is going to push the return address onto the stack, displacing the stack pointer. Data that is used by a subroutine should be referenced by an address or pointer, not by pushing it onto the stack.

Using the Indirect Pointers

Another attribute of page zero is storing the pointers for the indirect indexed addressing modes. The indirect indexed addressing mode is of significant importance because it provides the program-

mer with a convenient means of sequentially selecting memory locations. Using the indirect indexed mode, sequential locations from several different areas of the memory are easily indicated. A program for transferring data from one table to another, or for comparing two storage areas would be ideal candidates for using this addressing mode. The proper manipulation of these indirect pointers is essential if full advantage of this addressing mode is desired.

Outputting data from a section of the memory to an output device is one common use for the indirect indexed addressing mode. Editor and assembler programs use it to transmit text and source data from one buffer to another. The following sample routine illustrates the basic structure of these routines.

In this routine, the pointer is stored on page zero. FMPNT is the pointer to the area from which data is to be output. CRTDSP is the address of the video display device which will receive the transmitted data. The Y index register is initialized to zero. After each byte is transmitted, the Y index register is incremented. If, as a result of incrementing, the Y index register becomes zero, the upper half of FMPNT is incremented. Incrementing in this fashion allows more than 256 bytes to be transmitted at one time. This routine also tests the data transmitted to indicate the end of the buffer. A zero byte in the buffer is used to signify the end. When it is encountered, the routine returns to the calling program.

```
         *=$0000
FMPNT    *=*+2            From pointer storage
         *=$0200
TBLOUT   LDA (FMPNT),Y    Read character from table
         BEQ ZCHAR        Zero byte indicates end of buffer
         STA CRTDSP       Output to CRT display
         INY              Advance index pointer
         BNE TBLOUT       Not zero, continue output
         INC FMPNT+1      Advance base pointer by 256
         BNE TBLOUT       Continue output
ZCHAR    RTS              Last character output, return
```

A Conditional Branch Can Save Memory

One very useful programming trick is indicated in this routine. Following the INC FMPNT+1 instruction, the program jumps back to the beginning by using the two-byte BNE TBLOUT instruction. FMPNT+1 is the page portion of the pointer. In 95 percent of the

6502-based computers, the last RAM location never exceeds $DFFF. It is safe to assume that incrementing FMPNT+1 will not cause it to go to zero. Thus, the two-byte BNE saves one memory location over the three-byte JMP instruction. One should also watch for places where a conditional branch is followed by a jump instruction. In these situations, the branch instruction for the opposite condition of the first branch may be used in place of the jump instruction.

Counting Characters and Events

The TBLOUT routine was terminated when a specific code, namely $00, was encountered in the data being transmitted. Another method of detecting when the program has completed its operation is to set up a register or memory location as a countdown register. This register would initially contain a binary count of the number of times the program loop is to be executed. Then, for each pass through the loop, the countdown register would be decremented and checked for a value of zero. If the register did not go to zero, the loop would be continued. When the register reaches zero, the program would jump out of the loop. The following program listing performs an operation similar to TBLOUT. However, rather than check for a terminating zero byte, the program uses the X index register as a countdown register. Before calling this routine, the X index register must be set to the number of bytes to be transmitted. FMPNT and Y are initialized as before.

```
        *=$0000
FMPNT   *=*+2           Indirect pointer
        *=$0200
MESSAG  LDA (FMPNT),Y   Fetch character to output
        STA CRTDSP      Display on CRT
        INY             Advance buffer pointer
        DEX             Decrement character count
        BNE MESSAG      Count = zero? No, continue
        RTS             Yes, output complete
```

It is often desired to have a computer count the number of times a specific operation is performed. The counter may perform this function. It may also indicate the number of characters received from an input device. There are several ways a counter may be set up. One is to designate one of the index registers as the counter register, and each time a count is to be made, the INX or INY instruc-

tion may be executed. Since the index registers are eight-bits long, they may be used to indicate a count from 0 to 255.

To set up an index register as a counter, one would load the designated register with zeros and insert the proper increment instruction at the location within the program where the count is to be made. When the process is complete, the designated index register will contain a binary count of the number of times the operation occurred.

The accumulator and index register are generally required to perform other operations while a counter is to be maintained. One or more memory locations may be designated as the counter storage area. If the count is expected to exceed 255, several memory locations would have to be used for the counter. For a single memory location, an increment instruction would be inserted in the instruction sequence where the count is to be made. When two or more memory locations are to be used for the counter, a series of instructions would be needed to increment the counter. The instruction sequence is to increment the memory location containing the least significant portion of the count and check its contents for a value of zero. If the least significant portion went to zero as a result of the increment, the next memory location would also be incremented. If a third byte of memory is used as part of the counter, its contents would be incremented whenever incrementing the second byte resulted in the second byte going to zero. A sample program listing for incrementing a triple precision counter is presented here. The memory locations used for the counter are designated CNTR1, CNTR2 and CNTR3.

```
COUNTR    INC CNTR1    Increment the LSByte of the counter
          BNE OVER     = zero? No, skip other incr instruction
          INC CNTR2    Yes, increment second byte of counter
          BNE OVER     = zero? No, skip next instruction
          INC CNTR3    Yes, incr the MSByte of the counter
OVER      ...          Continue processing
```

Chapter 3

General Purpose Routines

Whenever one writes a program, there are usually several basic operations that occur over and over. These operations may be executed for completely unrelated tasks. However, the instruction sequence may be the same. For example, rotating a group of memory locations to the left can be used to multiply a binary number by two or to properly position a BCD number. Such frequently encountered instruction sequences are often set up as subroutines.

Subroutines usually fall into two classifications. General purpose subroutines are written to perform a specific function for a variety of applications. This is accomplished by defining how certain registers and memory locations are to be initialized before calling the subroutine. In the rotate subroutine, the X index register might be pointing to the first location to be rotated and the Y index register could be used as a counter for the number of locations affected. Thus, any number of memory locations can be rotated by proper selection of X and Y, before calling the rotate subroutine. This class of subroutines is placed in a library of subroutines so a programmer does not have to continually "reinvent the wheel."

The second class of subroutine is that which performs an instruction sequence unique to a given program. As one writes a program, an algorithm may occur two or more times throughout the program. This algorithm may be essential to the program it is written for, but may have no meaning to any other one. Such an algorithm might execute an unusual calculation or generate an output to an uncommon peripheral device. It is important to recognize these algorithms when they occur and form subroutines from them. This will aid in conserving memory usage. Added time savings occur by

shortening the assembly and debugging time, since the subroutine is assembled and debugged only once.

A number of subroutines come under the general purpose classification. Although these routines are presented as subroutines, they may be revised to be used in line with a given instruction sequence. This can help save memory space if the routine is called upon only once. The necessary revision is accomplished by either deleting the return instruction or replacing it with a jump instruction.

The subroutines presented in this chapter make use of several addressing modes. Each one has a function which may make it more efficient to use than another. Some apply better when short strings of data are being manipulated. Others lend more readily to operations involving long banks of data, possibly extending over numerous pages in the memory. The addressing mode used in each subroutine is the most efficient in memory usage and execution time for the description presented. There are other alternatives, however, depending on the applications.

Clearing a Section of Memory

When setting up a program for entering data or storing the results of a calculation, it is often desirable to clear the memory locations to be used for storage. This operation is achieved by filling the memory locations with zeros. One way to do this is to store zero in the accumulator and perform a series of STA ADDR instructions in which the ADDR designates each memory location to be cleared. This method is fine if the area to be cleared consists of only two or three memory locations, and the clearing operation is required in only one or two differerent portions of the program. However, if a lengthy table area must be cleared, such as an input buffer (which may store 72 characters or more for a single line of input), this would be highly impractical. It would use more memory locations than necessary, even for short tables to be cleared by different routines throughout a program.

An alternative subroutine which, when called, will clear as many locations in a table area as defined by the calling program. The routine listed below will fill up to 256 memory locations with zeros. The calling routine must store the lowest address of the table in TOPNT on page zero. Also, the X index register must contain the binary count of the number of locations to be cleared. CLRMEM is the start of the subroutine. The accumulator and the X index register will be equal to zero upon returning.

TOPNT is a successive pair of memory locations set up on page

zero. It will be used as storage for a temporary pointer. The low portion of the address stored in TOPNT is stored in the lower addressed byte and the page portion is stored in the next higher byte.

```
CLRMEM   LDA #$00         Set up zero value
         TAY              Initialize index pointer
CLRM1    STA (TOPNT),Y    Clear memory location
         INY              Advance index pointer
         DEX              Decrement counter
         BNE CLRM1        Not zero, continue clearing
         RTS              Return
```

Transferring a Section of Memory

Programs of varying applications often have a similar requirement: the transfer of information from one section of memory to another. For example, an editor program may transfer data from the input buffer to the main text buffer. A calculator may transfer a multiple precision value from a storage area to a working area in the memory. The programming to perform this function is basically the same in either case. The start address for the section of memory to be transferred and the section of memory to receive the data, are set up. Either the count for the number of memory locations to be transferred, or the address of the last location to be transferred must be indicated.

The first transfer routine is limited to 256 bytes of memory. This is useful for moving data values from a storage area to a working buffer and vice versa. The initial pointers are set up in FMPNT and TOPNT. FMPNT is a two-byte pointer on page zero, similar to TOPNT, which must point to the table from which data is to be transferred. TOPNT must be set to the destination storage area. Index register X is set to the number of bytes to be moved. The Y index register is initialized by the routine to zero and is used as the index pointer to both memory areas.

```
MOVIND   LDY #$00         Initialize the index pointer
MOVIN1   LDA (FMPNT),Y    Fetch byte to transfer
         STA (TOPNT),Y    Store byte in new location
         INY              Advance index pointer
         DEX              Decrement byte counter
         BNE MOVIN1       Not zero, continue
         RTS              Return
```

The next transfer routine compares the contents of the pair of memory locations labeled ADDCHK with the FROM pointer to determine when the last location has been moved. This method is applicable when the last location is always known, or constant, and the byte count is variable. When this subroutine is called, ADDCHK must be set to the last location of the section to be transferred. FMPNT and TOPNT should indicate the starting addresses of their respective areas of memory. The X and Y index registers are set to zero. After each byte is transferred, FMPNT is compared to ADDCHK. When they are equal, the transfer is complete.

```
MOVEAD   LDA (FMPNT,X)    Fetch data to transfer
         STA (TOPNT),Y    Store data in new location
         LDA FMPNT        Fetch LS half to FMPNT
         CMP ADDCHK       Is it equal to LS half of last address?
         BNE PNTADV       No, advance pointer
         LDA FMPNT+1      Fetch MS half of FMPNT
         CMP ADDCHK       Is it equal to MS half of last address?
         BNE PNTADV       No, advance pointer
         RTS              Yes, return to calling program
PNTADV   INY              Advance index pointer
         BNE FMADV        N or zero, advance FROM pointer
         INC TOPNT+1      Advance TO base pointer
FMADV    INC FMPNT        Advance LS half of FROM pointer
         BNE MOVEAD       Not zero, continue transfer
         INC FMPNT+1      Advance MS half of FROM pointer
         JMP MOVEAD       Continue transfer
```

Multiple Precision Routines

When dealing with numerical data, it is often necessary to use more than one eight-bit byte to represent a binary number. Since a single byte can only represent a value from 0 to 255, one would be quite limited in the type of calculations that could be performed. This problem is solved by manipulating the data in several bytes as through they were one long register or memory location: N X 8 bits long (N = number of bytes used to represent the data value). For example, by using two bytes as though they were a single sixteen-bit register, the decimal values from 0 to 65,535 may be represented in binary format. This form of representation is referred to as multiple precision.

In order to perform operations that consider several bytes as one, there must be some link to carry the effects of an operation on

one byte over to the next. This link is the carry flag. The carry flag indicates whether an operation on one byte of a multiple precision operation should carry over to the next byte. When the addition of a number to a low order byte of a multiple precision value creates an overflow, the carry flag will be set to a "1." This will be included in the addition of the next higher byte of a multiple precision value. Similarly, when a subtraction requires a borrow for the MSB of a multiple precision byte, the carry will be reset and will create a borrow from the LSB of the next higher byte of the multiple precision number.

The subroutines described next perform a variety of multiple precision operations on values stored in the memory. These operations include incrementing, decrementing, rotating left, rotating right, and complementing a single precision value. Also, they may be used for adding, subtracting, and comparing a pair of multiple precision values with each other. For these routines, the multiple precision value(s) is assumed to be stored in consecutive memory locations with the least significant byte in the lowest address.

Incrementing a Multiple Precision Value

There are a number of different reasons why a multiple precision value may have to be incremented. It may be to (1) advance a pointer that is stored in the memory, (2) increment an event counter, or (3) simply add one to a binary value. For whatever reason, the basic process consists of incrementing the least significant byte and, if it goes to zero as a result, the next byte will be incremented. This process ends when a byte does not go to zero, or when the most significant byte has been incremented. The first instruction sequence may be used to increment a double precision value. It increments the least significant byte and, if zero, increments the second byte.

```
        INC MEMADR      Increment the LS Byte
        BNE NEXT        Not zero, skip next instruction
        INC MEMADR+1    Increment the MS Byte
NEXT       —            Continue processing
```

The next routine increments a multiple precision value stored in the memory. The label VALUE should be set to the first location of the page in which the data is stored. For example, if the data is stored on page 00, VALUE should be set equal to zero. This restricts the subroutine to numbers stored on the designated page. However,

one should keep all data storage confined to the same section of the memory. Then, the X index register is set to the location on page zero of the least significant byte of the multiple precision number. The Y index register is used as a byte counter. It should be initialized to the number of bytes in the multiple precision number.

There are two important facts upon returning from this subroutine. First, the contents of the index register will point to one of two locations. Either the last byte which, when incremented, did not go to zero, or to the last location plus one of the multiple precision value. Also, the Z flag will be set to "1" upon returning when the entire value has gone to zero. If any of the bytes do not go to zero, the Z flag will be "0" when the return is executed.

```
INCMEM   INC VALUE,X    Increment memory contents
         BNE INRET      If result not zero, return
         INX            Advance index pointer
         DEY            Decrement byte counter
         BNE INCMEM     Not zero, continue incrementing
INRET    RTS            Complete, return
```

Decrementing a Multiple Precision Value

The procedure for decrementing a multiple precision value is similar to incrementing. However, different criteria are used to determine when the succeeding byte should be decremented. The next byte is decremented only when the byte being decremented goes from zero to $FF. In this case, a borrow is required from the next byte. The DEC instruction does not condition the flags to indicate the change from zero to $FF, so a different instruction sequence must be used. This sequence uses the SBC #$01 instruction to decrement a byte because it will cause the C flag to be reset to "0" when the zero-to-$FF transition occurs.

Since the SBC instruction is used, this routine may decrement a decimal multiple-precision value as well as a binary value. This is accomplished by setting the decimal mode flag before calling this subroutine. For binary multiple precision values, the decimal mode flag must be cleared.

The following subroutine decrements the multiple precision value indicated by TOPNT, which is set to the least significant byte. The Y index register must be initialized to zero and the X index register to the number of bytes in the value to be decremented. The contents of the Y index register cannot be assumed to point to any one particular byte upon returning.

```
DCRMEM   LDA (TOPNT),Y   Fetch byte to be decremented
         SEC             Set carry for subtraction
         SBC #$01        Decrement value by one
         STA (TOPNT),Y   Restore byte in memory
         INY             Advance pointer
         DEX             Decrement byte counter
         BNE DCRET       Last byte decremented, return
         BCC DCRMEM      Next byte should be decremented
DCRET    RTS             Return to calling program
```

Rotating a Multiple Precision Value

A binary number can be multiplied times two by shifting each bit one position to the left, and loading the LSB with a "0." Conversely, by shifting each bit of a binary number to the right one bit position, and setting the MSB to "0," the binary value is divided by two. When rotating a multiple precision number, it is necessary to carry the bit shifted out of a byte over to the next byte. This is accomplished by the rotate instructions, which include the carry flag as part of the byte when rotating either left or right. For a rotate left operation, the MSB shifted out of the lower order byte will be shifted into the LSB of the next byte.

The first routine listed here is labeled the ROTATL subroutine. It uses the constant VALUE set to the first location of the page on which the data to be rotated resides. The X index register must be initialized to the location of the least significant byte of the data. The number of bytes to be rotated must be stored in the Y index register. The initial operation is to clear the carry flag. This creates the "0" bit, which must be loaded into the LSB of the multiple precision value. If it is desired to check for a "1" rotated out of the MSB of the value, the carry flag will be properly conditioned upon returning. Also, the X index register will point to the most significant byte.

```
ROTATL   CLC             Clear the carry
ROTL     ROL VALUE,X     Rotate the byte left
         DEY             Decrement the byte counter
         BNE MORRTL      Not zero, continue rotate
         RTS             Done, return
MORRTL   INX             Advance memory pointer
         JMP ROTL        Continue to rotate left
```

The ROTATR subroutine rotates the designated multiple pre-

cision value to the right. The X index register must indicate the location of the most significant byte of the value when calling this routine, since it works from the most significant byte down to the least significant byte. Here again, VALUE is set to the zero location of the page on which the data resides. The Y index register must be set to the number of bytes in the multiple precision value. The carry flag is cleared initially to provide the "0" to be shifted into the MSB of the value. If it is desired to rotate a "1" into the MSB of the most significant byte, the carry flag may be set and this routine may be entered at the second entry point, labeled ROTR.

```
ROTATR   CLC            Clear the carry
ROTR     ROR VALUE,X    Rotate the byte right
         DEY            Decrement the byte counter
         BNE MORRTR     Not zero, continue rotate
         RTS            Done, return
MORRTR   DEX            Decrement memory pointer
         JMP ROTR       Continue rotate right
```

Complementing a Multiple Precision Number

The complement of a binary value is performed by changing each bit to the opposite condition of its current state. If a bit is a "1," it is changed to a "0"; if a bit is a "0," it is changed to a "1." This type of complement is often referred to as the one's complement of a binary number. The one's complement is used to complement data received from an input device if it is in the opposite state of that required by the program. The complement of the inputted data may be derived by a simple EOR #$FF instruction just after reading in the data.

Another form of binary complement is the two's complement which may be formed by subtracting the binary number from zero. The two's complement is generally used when a negative value of a binary number is desired. Or, it may be used to form the negative of a subtrahend value that may be added to the minuend. This subtracts the subtrahend from the minuend. The two's complement of a single byte may be achieved by complementing and then incrementing the byte.

The following routine forms the two's complement of a multiple precision binary number stored in the memory. When this routine is called, the X index register must indicate the least significant byte of the multiple precision value to be complemented. The Y index register must contain the number of bytes defined for the value.

VALUE is again assumed to point to the first memory location of the page on which the number to be complemented resides. When the routine returns, the X index register will point to the most significant byte +1.

Both two's-complement and one's-complement operations are executed in this routine. First, the least significant byte is two's complemented. This is accomplished by exclusive ORing $FF with the byte and incrementing the result. If the result leaves the byte equal to zero, the next byte also will be two's complemented. When a byte is left with a nonzero result, the remainder of the number is one's complemented.

```
COMPLM   SEC            Set carry for two's complement
COMPL    LDA #$FF       Load $FF for complement operation
         EOR VALUE,X    Complement byte
         ADC #$00       If carry = one, two's complement
         STA VALUE,X    Store byte in memory
         INX            Advance memory pointer
         DEY            Decrement byte counter
         BNE COMPL      Not zero, continue
         RTS            Return to calling program
```

Multiple Precision Addition and Subtraction

Addition and subtraction are common functions often required when dealing with multiple precision values that represent numeric data. Both operations work from the least significant byte up to the most significant byte, using the carry flag as the link between bytes. When the addition of two bytes results in an overflow from the MSB, the carry flag is set and is included in the addition of the LSB's of the next pair of bytes. Conversely, if the subtraction of a pair of bytes results in a borrow required from the next byte of the minuend, the carry flag is reset. This causes a borrow from the LSB of the next byte of the subtraction. These routines can operate on binary numbers or decimal numbers, depending on the setting of the decimal mode flag.

The addition routine is labeled ADDER. This routine adds the multiple precision value indicated by the pointer in TOPNT to the value indicated by FMPNT. The result of the addition is stored in place of the value indicated by TOPNT. FMPNT and TOPNT must be set to the least significant byte of the multiple precision numbers. The X index register must be set to the binary count of the number of bytes in the multiple precision values. The carry flag will indicate

whether an overflow from the MSB of the most significant byte has occurred upon returning from this routine. The calling routine may have to check this flag since an overflow would usually indicate an error condition. At the completion of this routine, the index register will be pointing to the most significant byte plus one of the result.

```
ADDER    CLC              Clear carry flag
ADDR1    LDA (TOPNT),Y    Fetch byte from one value
         ADC (FMPNT),Y    Add to byte of other value
         STA (TOPNT),Y    Store result
         INY              Increment index pointer
         DEX              Decrement counter
         BNE ADDR1        Not zero, continue addition
         RTS              Return
```

The subtraction routine, labeled SUBBER, subtracts two multiple precision values stored in the memory. TOPNT must indicate the least significant byte of the minuend. FMPNT must indicate the least significant byte of the subtrahend. The X index register must contain the binary count of the number of bytes in each multiple precision value. The result of the subtraction is stored in place of the minuend. The carry flag will be reset if a borrow was required by the subtraction of the MSB of the most significant byte.

```
SUBBER   SEC              Set carry flag
SUBB1    LDA (TOPNT),Y    Fetch byte from minuend
         SBC (FMPNT),Y    Subtract byte from subtrahend
         STA (TOPNT),Y    Store result over minuend
         INY              Increment index pointer
         DEX              Decrement byte counter
         BNE SUBB1        Not zero, continue subtraction
         RTS              Return
```

Comparing Two Multiple Precision Values

It is often desired to determine whether one number is larger or smaller in magnitude than another. This fact may change the manner in which a program is to deal with two numbers. For example, when subtracting two numbers, it is usually necessary to subtract the larger from the smaller and, if indicated, change the sign of the result. The following routine may be used to compare two multiple precision numbers stored in memory.

The COMPARE routine compares the multiple precision value

indicated by the pointer in FMPNT against the value indicated by the pointer in TOPNT. These pointers initially must be set to the least significant byte minus one of the respective values to be compared. The Y index register must be set to the binary count of the number of bytes to be compared. Upon returning, the carry and zero flags will be set to indicate the outcome of the comparison. The calling program must check these flags and enter the proper routine as a result of the comparison. This is accomplished by using the BCC, BCS or BEQ conditional branch instructions. The index register will equal zero if the two values are equal, or the location of the byte at which the comparison failed.

```
CMPLOP   DEY              Decrement pointer
         BEQ CMPRET       If zero, both values equal**
CMPMEM   LDA (FMPNT),Y    Fetch compare data
         CMP (TOPNT),Y    Compare to indicated data
         BEQ CMPLOP       If equal, continue comparing
CMPRET   RTS              Return with C and Z flags conditioned
```

A similar routine may be used to compare alphabetic information such as: (1) one name against another, (2) duplication, (3) if the character set is well ordered (as is the case with the ASCII code), to place the names in alphabetical order.

To compare two character strings, first set FMPNT and TOPNT to the first character of each string. The Y index register should be initialized to zero and the X index register to the number of characters to be compared. At the instruction marked by **, replace it with INY and insert DEX immediately after it. This setup assumes the first character of each string is stored in the lowest address.

The following listing illustrates a possible instruction sequence for calling the CPRMEM routine, and checking the results of the compare operation for one of the three possible conditions.

```
LDA #TABL1-1   Set up pointer to value to be
STA FMPNT      Compared against
LDA #TABL1H    Set up page portion of pointer
STA FMPNT+1
LDA #TABL2-1   Set up pointer to value to be compared
                 to
STA TOPNT
LDA #TABL2H    Set up page portion of pointer
STA TOPNT +1
```

```
LDY #COUNT      Set byte count in Y
JSR CMPMEM      Compare FMPNT
BEQ EQUAL       TABL1 = TABL2
BCC GRTR        TABL1 > TABL2
...             TABL 1 < TABL2, begin processing
                For less than
```

Checking for Value within Limits

Another type of comparison often required is to check whether the value of a byte of data falls within expected limits. One frequent application is to check the code received from an input device. For example, a calculator program may check each character input for a legal digit code when it expects to be receiving only digital information. Or, a control program may check inputs from a sensing device to determine whether a parameter is within allowable limits. When the data being checked falls within sequential limits (limits defined by an upper and lower bound), the following type of routine may be used.

This routine compares a byte of data against the lower limit minus one and the upper limit of the boundaries in which the data byte must fall. The reason for checking the lower limit minus one is to allow the condition of the carry flag, upon returning, to indicate whether the byte falls within the designated limits. When the routine returns to the calling program with the carry set, the byte is not within the limits. When this routine is called, the data byte to be checked must be in the accumulator.

The routine listed below checks the ASCII code for the digits 0 through 9. ($B0 to $B9). To check for the ASCII code for the alphabetic characters A through Z, the immediate portion of the compare instructions would simply be changed to $C0 (ASCII A minus 1) and $DA (ASCII D). This routine begins at the label LMTCHK.

```
LMTCHK   CMP #$AF      Is byte less than ASCII zero?
         BCS LMTRET    Yes, not in limits, return with C = 1
         CMP #$B9      Is data byte greater than ASCII nine?
         BCS CRCLR     If not, reset C to zero before returning
         SEC           If so, return with C = 1
LMTRET   RTS           Return to calling program
CRCLR    CLC           Within limits, return with C = zero
         RTS           Return
```

Programmed Time Delays

The computer is designed to execute a program stored in its memory as rapidly as possible. It does not hesitate between instructions to contemplate the next operation it should perform. However, there are certain types of programs that require a hesitation, or delay, between one operation and the next. One program is a display program that outputs a frame of characters or pattern to a video device, and then must wait a specific amount of time before outputting the next frame or pattern. Or, a delay may be required after outputting a control command, which turns on a motor driven device, to allow the motor to get up to speed before a data transfer may be initiated. A programmed delay also may be required between outputting of each bit of a serial data pattern to allow the program to control the data transmission rate. By inserting program time-delay sequences, one may affect these real time program applications.

Each instruction requires a specific number of cycles and therefore needs a specific amount of time to execute. A delay may be created by knowing the exact time for each instruction and programming a group of instructions whose total execution time is close to the desired delay. (For the 6502 with a clock frequency of one Megahertz, one should be able to program a delay within two microseconds of the required time.) Depending on the type of memory used in one's system, the actual timing of the instructions may vary from those presented in Appendix A. Before getting into the time-delay programming, it is necessary to understand the differences between various types of memories so that one will be able to discern the actual timing for one's own particular system.

This description is presented in general terms. It is not intended to present specific details of memory accessing. Refer to the manual supplied by the particular hardware manufacturer for details on memory accessing.

When a computer must access a memory location in order to read an instruction, obtain data, or write data into it, the address of the memory location is first placed on the memory address bus. Then, the memory must be given time to select the memory location. The contents of the location then may be read from the data bus by the CPU, or the contents of the data bus may be written into the memory location. The length of time required to access a memory location for reading or writing is referred to as the memory speed. The delay required between sending the address and accessing the memory location may vary, depending on this speed. If the nor-

mal delay in the CPU instruction time is sufficient for the memory to react to the address selection, the data may be read or written in the following cycle. However, if the memory cannot react fast enough, one or more wait cycles must be executed before reading from, or writing to the memory location.

A wait cycle is a "do nothing state." The hardware slows down to allow time to access the slower memory. A single wait cycle for the 6502 takes approximately one microsecond. (It varies with different clock frequencies.) The number of wait cycles used by a specific microcomputer is detailed by the hardware manual. Knowing the number of wait cycles allows one to program exact delays to a system that uses ROM or static RAM.

Dynamic RAM memory makes it difficult to calculate the instruction timing accurately. The reason for this is that the dynamic RAM memory requires a refresh cycle at least once every one or two milliseconds. (This time may vary for different types of dynamic memory.) A refresh cycle means that within the allotted time, each memory address must be accessed with a read cycle in order for the memory to maintain its current contents. This refresh process may interrupt the timing of the CPU instructions, since the refresh circuitry may be accessing a memory location at the same time the CPU may require a memory access. In this case, the CPU would have to wait for the refresh read to complete, thereby extending the time required for the instruction to execute. It is possible only to calculate a minimum time delay for a given instruction sequence, and not the maximum, when using dynamic RAM memory.

One feature of the 6502 which affects its timing is its pipelining capability. Pipelining means that the CPU can internally execute a portion of an instruction while fetching the next byte from the memory. This overlapping has the effect of shorting the execution time of an instruction. The number of memory accesses governs the actual number of cycles an instruction requires.

The use of ROM or static RAM memory coupled with an understanding of pipelining allows one to determine the exact timing for each instruction. However, this is detailed in Appendix A. Each instruction is given as well as the number of actual cycles required for execution. By multiplying the number of cycle times, the timing for all instructions can be calculated. If one or more wait states are added, this additional time must be added to each memory access executed per instruction.

With a knowledge of the timing necessary for the instructions, one may begin to program time delays of specific duration. In program-

ming a delay, strive to use as few instructions as possible. Also, assure that the instructions used in the delay do not interfere with the operation of the main program. Unless stated otherwise, the times given below are those listed in Appendix A, and assume no wait cycles have been added to the memory access time.

For very short delays (2 to 20 microseconds), several instructions that fall in the direct sequence of the program may be used. Suppose a delay of 6 microseconds is required at a certain point. A no-operation instruction requires 2 microseconds to execute, so the desired 6-microsecond delay may be derived by using three NOP instructions at the point in the program requiring the delay.

Another method used to create short delays is to insert a jump-to-subroutine instruction that jumps to a location that contains a return instruction. This sequence would delay 6 microseconds for the jump-to-subroutine instruction plus 6 microseconds for the return, for a total of 12 microseconds. To conserve memory, the return instruction may be part of an existing routine. It need not be set up as a return specifically for this delay.

For longer delays, the method of inserting the delay instructions in sequence with the main program would begin to waste a great deal of memory. An alternative is to use a subroutine that will form a timing loop to delay the desired amount of time. The following routine allows control of the delay time be selection of an initial value for the Y index register. The delay is created by forming a program loop that decrements the Y index register until it reaches zero, and then returns. The larger the initial value of the Y index register, the longer the delay. (The exception is that the initial value of zero will create the longest delay.)

```
DELAY      DEY          Decrement delay cntr (2 μsec)
           BNE DELAY    If cntr ≠ zero, loop back (3 μsec)
           RTS          Counter = zero, return (6 μsec)
```

The amount of time used by this routine is calculated by adding the time required for each instruction every time it is executed. The execution time for each instruction is given in parenthesis after each comment. The following formula may be used to calculate the delay time for a given value of Y. If Y is initially zero, the value of 256 must be substituted in this equation.

```
             LDY JSR DEY BNE        RTS
DELAY TIME = 2 + 6 + (2 + 3) * (Y) + 6
```

The time required for the instruction LDY and JSR, which sets up the Y index register to the required constant and then calls the DELAY subroutine, must be included in the calculation. The time given for the LDY instruction assumes the immediate addressing mode. The time required for setting up the Y index register may be excluded if it is set up before the time delay begins.

The time delay that can be created by this program loop runs in increments of five from a minimum of 19 microseconds for the Y index register equal to one, to a maximum of 1294 microseconds for the Y index register equal to zero. This incremental factor is controlled by the loop DEY,BNE. Should it be desired to expand the increment, and thereby extend the maximum delay possible, additional instructions may be added. For example, if the incremental factor is desired to be 8 microseconds rather than 6, a NOP instruction can be inserted between the DEY and BNE instruction. This will add 2 microseconds to the loop without altering the basic operation of the routine.

Sometimes the actual delay required by a program does not equal one of the incremental times generated by the delay loop. The delay may be adjusted by setting the Y index register to the closest incremental time without exceeding the time desired. Then, adding one or two instructions to the calling sequence brings the total delay within 1 or 2 microseconds. As an example, suppose a delay of 428 microseconds is needed. Selecting a value of 82 for the Y index register will provide a delay of 424 microseconds. The additional 4 microseconds can be added by two NOP instructions in the calling routine before the JSR DELAY instruction. These additional instructions will add the necessary 4 microseconds to the total delay.

Substantially longer timing loops can be derived by nesting delay loops. Using both index registers one can set up a delay loop within a delay loop. Then, when one loop goes through a complete cycle, the second loop will be decremented once. This multiplies the time required for the inside loop by the initial value (minus one) of the index register in the outside loop. The following routine, which includes the calling sequence, illustrates this method of nesting delay loops. The Y index register is used for the outside loop. The greater the initial value of the registers, the longer the delay, with the exception of zero, which creates the longest delay.

```
        ...
        LDY #$YY        Set initial inside loop (2 µsec)
```

```
         LDX #$XX        Set initial outside loop (2 µsec)
         JSR DLYLOP      Call delay loop routine (6 µsec)
         ...
DLYLOP   DEX             Decrement outside loop (2 µsec)
         BNE DLYLP1      If ≠ zero, branch to inside loop
                          (3 µsec)
         RTS             If = zero, return, delay over (6 µsec)
DLYLP1   DEY             Decrement inside loop (2 µsec)
         BEQ DLYLOP      If = zero, branch to outside loop
                          (3 µsec)
         JMP DLYLP1      If ≠ zero, continue inside loop
                          (3 µsec)
```

Calculation of the time amount required for execution can be made from the formula given. This formula is shown in two forms. One indicates the instruction sequence that is executed, and the second provides a condensed version for use in making the actual calculation.

```
                LDY   LDX  JSR
 DELAY TIME =   2  +   2 +  6

     DEX  BNE                   DEY  BEQ JMP  DEY BEQ
      [2 + 3 + ((INIT Y) -1)* (2   + 3 + 3) + 2 +  3)] +

                     DEX  BNE        DECB BEQ JMP    DECB BEQ
     [((INIT X) -2)* (2  + 3 + (255* (2    + 3 + 3))) + 2 + 3] +

     DEX BNE RTS
      2 + 3 +  6

DELAY TIME = (((INIT X) -2)* 2045) + (((INIT Y) -1)* 8) + 36
```

The first formula has two sections in brackets. The first bracketed section indicates the time for the first pass through the inside loop. The second bracketed section indicates the time for all successive passes. The reason for the separation of these times is that on the first pass through the inside loop, the value of the Y index register will be as initialized by the calling program. After the first pass, the Y index register will always be zero when the inside loop is entered. This formula is only valid for initial values of the X index register from two to 256. (In actual operation of the subrou-

tine, the index registers are initialized to zero when 256 is used in the formula.) If the X index register is initially set to one, the execution time is simply the sum of the times not enclosed in the brackets, which is 21 microseconds. For values of index register X from two to 256, the time delay can be set within the limits of 36 to 521,506 microseconds in intervals of 8 microseconds. If finer selection is required, the technique discussed previously of inserting instructions in the calling sequence may be used.

Random Number Generators

The purpose of a random number generator is to provide a nonrepeating series of random numbers. These random numbers may be applied to several different programs. When a game (such as dice or blackjack) is programmed, the program must provide a random assortment of numbers for the roll of the dice or a draw of a card. This is accomplished by using some form of a random-number generator routine. Another application for these generators it to create random patterns for testing devices such as a computer's memory, which may be sensitive to various patterns.

Two methods of programming random number generators will be presented. The first is very simple and is used when the numbers are required in response to an input from the program operator. The second method uses a routine that will produce a new random number each time it is called.

With many game programs, a random number is required in response to an input received from the operator. The random number may be derived by constantly incrementing a memory location until the input is received. This may be accomplished by forming a program loop that increments the register and then checks the status of the input device for an input from the operator. If the status indicates there is no input, the routine will loop back to increment the memory location again. This program loop should be short, probably in the range of 30 to 50 microseconds. It would be impossible for a human to select the precise time to input a character to stop the loop when a specific value is present. For programs that require random numbers following an operator input, the following routine may be used to generate random numbers. The CHKINP in this program is assumed to check the input device status and return with the sign flag set to "1" if a character has been entered on the keyboard, or set to "0" if a character has not been entered. When the character has been received, the value in RNDM may be used as the random number for the program.

```
RNDMLP    INC RNDM      Increment random number
          JSR CHKINP    Check for character entered
          BPL RNDMLP    Not entered
             ...        Use RNDM for random number
```

When a program requires random numbers at various times throughout its operation (not necessarily after an input), the following routine may be used. It generates a pseudo-random data pattern of eight-bit bytes. This random-number generator is not a true generator because, depending on its initial values, it will create a repetitive pattern every 1000 to 4000 numbers. However, the program that uses it can make the data "more random" by using a trick that will be described shortly.

This random-number generator uses two consecutive memory locations to save the random number and an incrementing addend. Each time the routine is called, the random number created last is used in generating the next random number. It is operated by the series of instructions in this routine, and then the addend is added to it to create the new random number. At the same time, the addend will be incremented either once or twice, depending on the result of the addition of the random number to the addend. The new random number will be saved in the memory and returned to the calling program in the accumulator.

The trick referred to previously is to have the calling program alter the contents of the addend at a point in the program that is occasionally executed. This will increase the overall random pattern generated. For example, there may be a subroutine called once for every ten or fifteen times the random-number generator is called. An instruction sequence should be added to this subroutine to alter the addend. It may be altered by incrementing once, or adding five, or resetting the addend to zero. No matter what method is used to change the value of the addend, the result will be that of altering the data pattern generated. The instruction sequence that follows the routine below may be used by the calling program to alter the addend by adding five to it.

```
RANDOM
          LDA RNDM      Fetch random number
          ROL RNDM      Perform a series of
          EOR RNDM      Operations on it to
          ROR RNDM      Create a new random number
```

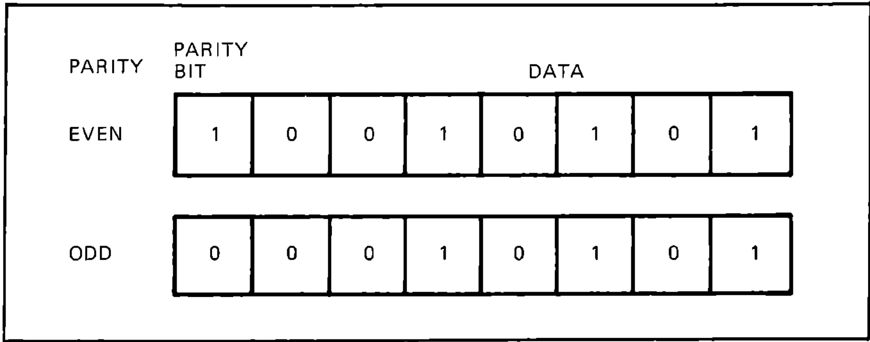
```
            INC RNDM+1   Increment the addend
            ADC RNDM+1   Add the addend
            BVC SKIP     If V = 0, increment addend once
            INC RNDM+1   Otherwise, increment it twice
SKIP        LDA RNDM     Store new random number
            RTS          Return to calling program
            ...          Sequence to add five to addend
            LDA RNDM+1   Fetch random number addend
            ADC #$5      Add five to addend
            STA RNDM+1   Save new addend
            ...          Continue processing
```

Checking Parity: An Error Detection Technique

An error may occur when data is transmitted from one device to another. Perhaps it's because of an intermittent problem with the transmitting or receiving equipment. Or, the error may be introduced by the communication channel (extraneous noise on a telephone line). Regardless of the cause, a technique to test for such errors is often desirable.

Checking for parity is a widely used method of error detection. It is used frequently when data is formed into small groups, eight bits, for example. Seven of the bits contain the data to be communicated. The eighth bit is used as the parity bit. The number of "one" bits among the seven determines the condition of the eighth. Odd parity simply means that of the eight bits, there must be an odd number of ones. If the seven data bits contain an even number, say two ones, the parity bit would be set to one. Likewise, if there were three ones among the seven, the parity bit would be set to zero. The objective in even parity is to manipulate the parity bit so that the total number of ones comes out even. Thus, the parity bit is always set to one or zero in order to make the total of ones equal to an even number in the case of even parity, and to an odd number for odd parity. The following example illustrates this point.

The following routine checks the parity of an eight-bit group. It may be used for determining either the condition of the parity bit for outputting data, or testing the parity of data received. When calling this routine, the data to be checked must be in the accumulator. The parity is checked by rotating each bit into the carry and incrementing a parity count for each bit found to be one. The parity count is stored in a separate memory location referred to by the label PTYCNT. After each bit is tested, the least significant bit of



the parity count is loaded into the C flag. When the routine returns, the accumulator will maintain its initial contents. The C flag will be set to one if the data has odd parity, or reset to zero if the data has even parity. If this routine is used to determine the setting of the parity bit for outputting the data, the parity bit should be zero when calling this routine, and then conditioned to create the desired parity by checking the C flag upon returning. The instruction sequence that follows this listing illustrates a method of setting up data to be transmitted with even parity. The MSB is assumed to be the parity bit. For checking the parity of data received, the data should be loaded into the accumulator before calling this routine. Upon returning, the C flag may be tested for odd or even parity.

```
PARITY   LDY #$08      Set bit counter
         CLR PTYCNT    Clear parity counter
LOOP     ROL A         Rotate bit into carry
         BCC ZEROBT    Bit = zero, don't increment parity count
         INC PTYCNT    Bit = one, add one to parity counter
ZEROBT   DEY           Decrement bit counter
         BNE LOOP      ≠ zero, continue parity check
         ROL A         Rotate once more to restore data
         ROR PTYCNT    Rotate LSB of parity cntr into carry
         RTS           Ret, C=1 odd parity, C=0 even parity
         ...
         JSR PARITY    Check parity of data to be transmitted
         BCC EVEN      Even parity, output data as is
         ORA #$80      Odd parity, set parity bit to make it even
EVEN     ...           Proceed to output data
```

Chapter 4

Conversion Routines

The real power provided by a computer is exemplified by its capability to operate with unlimited variations of character codes by simply changing a program. It can accept information in one form, convert it to another for processing, and then output it in the same format as initially received. Or, the output may be converted once again to an entirely different format. One may be aware of various other devices that perform such conversions. However, the input and output codes are most likely fixed, allowing no variation. A computer may be programmed to utilize a variety of codes for input, output and processing.

The need for code conversion and the type of conversion required is governed by two factors. The code used by the peripheral devices for transmitting and receiving data is one factor. If the input device transmits one code, and the output device must receive a different one, conversion from one code to the other is necessary. The format required by the program to process the data is the other factor.

Standard and Special Character Sets

The codes used by different I/O devices to transmit and receive data can vary greatly. Some codes are recognized as standard character sets which many peripherals utilize. Other codes may be the result of a hardware design which is most economical. This would create a special purpose code for which software conversion would be necessary. Several of the standard codes used to represent l-phanumeric information are ASCII, BAUDOT, EBCDIC and HOLLERITH. ASCII and BAUDOT are commonly used on keyboard and printer or display devices such as CRT terminals and teletypewriter machines. EBCDIC generally is used for mass storage

devices such as magnetic tape units, and HOLLERITH is normally associated with card reader/punch devices. The special purpose codes may be derived when interfacing a calculator-style keyboard in the configuration of a matrix.

The code used by a program to process the information may be the same code as received. Or, it may require conversion to a format more convenient for the computer. When a program deals with the manipulation of text, such as an editor program, the character code received by the program is often used for storing the text information. As each character is input, the code is stored as the representation to be used by the program for that character. For programs that deal with numeric data, for arithmetic operations, or designating digital information, conversion from the character code received to the binary or decimal equivalent may be required. A calculator program might receive the data as ASCII encoded decimal digits. This must be converted to the binary equivalent for processing and then back to ASCII digits to output the answer. A monitor program may require the conversion of the coded octal or hexadecimal input to the binary equivalent for defining memory addresses and their contents.

How Different Are ASCII and BAUDOT?

There are a number of standard codes used to transfer data from a peripheral device to a computer, and vice versa. For the following discussion on conversion from one code to another, the ASCII and BAUDOT codes will be used. Their contrasting formats aid in describing various methods of code conversion. Therefore, to preface the conversion routines, a brief discussion of each code is presented.

ASCII is a seven-bit code that represents the entire alphanumeric character set plus punctuation marks and a number of nonprinting control characters. An eighth bit is often added to this code. This bit can be used to provide parity for error checking or it can be set to a constant "1" or "0" condition for all characters. The ASCII codes for the printing characters and several of the control characters are presented in both octal and hexadecimal notation by Appendix D. The code used throughout this book wherever ASCII is discussed assumes the eighth bit is always set to "1."

As the reader may notice by examining Appendix D, the ASCII code is well-ordered. The letters of the alphabet are represented in sequential order from $C1 for A to $DA for Z. The numbers are similarly ordered from $B0 to $B9 for numbers zero through nine.

This coding for the numbers allows easy conversion from ASCII to binary-coded decimal by simply dropping the four most significant bits of the ASCII code. The ASCII code convenience sharply contrasts the BAUDOT code discussed next.

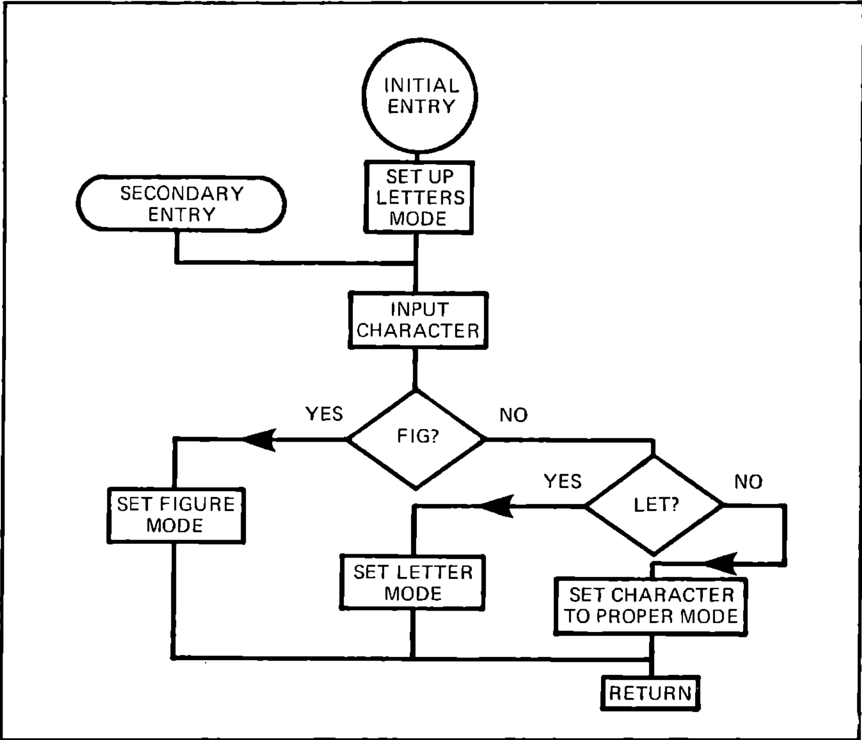
BAUDOT is a five-bit code used to represent the alphanumeric character set plus several punctuation marks and control characters. Appendix E contains the hexadecimal representation. It assumes the three most significant bits are all "0." The reader may question how five bits can be used to represent more than 32 characters. The answer is quite simple. Each of the letters of the alphabet shares its code with a numeral or punctuation mark. Two separate control characters are used to determine which of the two possible characters is being transmitted. One indicates the letters and the other indicates figures (numerals or punctuation marks). The proper mode (letters or figures) must be set by outputting the corresponding control character before the output of one or more of the characters of that mode. For example, if a sentence consisting entirely of letters were to be typed on a BAUDOT keyboard, the letters control character would be entered first. The letters that make up the words of the sentence would follow. Then, to end the sentence, the figures control character would be entered followed by the period, which shares its code with the letter M. The codes for space, carriage return, line feed, and null characters are common to both modes.

Examination of the BAUDOT code in Appendix E reveals the obvious scrambled pattern of character codes. There is no set pattern that would lend itself to ease of recognition of the BAUDOT letters as there is with the ASCII code. And, conversion of the BAUDOT code for numerals to the equivalent BCD values is not as trivial as conversion of the ASCII digits described previously.

Making BAUDOT More Workable

Programs that operate with the BAUDOT code must have some means of differentiating between the two characters a BAUDOT code may represent. This can be accomplished by defining one of the three most significant bits as a mode designator. One of these three bits would be set to "0" for the letter mode, and to "1" for the figure mode. For this discussion, bit five will be so designated. The following pair of routines may be used to encode and decode the BAUDOT characters according to this method for separating the letters from the figures. The first routine is used to encode the BAUDOT characters as they are input. There are two entry points for this routine. The first, labeled BAUDIN, is used when the input

of characters are to be initialized. The initialization is done by outputting a letters control character to the printer device before calling the input routine to receive a character from the keyboard. A memory location is set up to indicate the current mode of the printer device. This memory location, labeled CNTRL, is conditioned by the receipt of the letters or figures control characters. It is used to encode the characters as they are received. The other entry point, at label INBAUD, is used after the initialization has been completed. This entry point assumes that CNTRL is properly conditioned. The routine returns to the calling program with the character contained in the accumulator. The listing and flow chart for this routine are now presented.




```
BAUDIN   LDA #$1F      Load letters code into accumulator
         JSR OUTPUT    Call routine to send BAUDOT charac-
                         ter
         JSR LETCOD    Initialize CNTRL to letters code
```

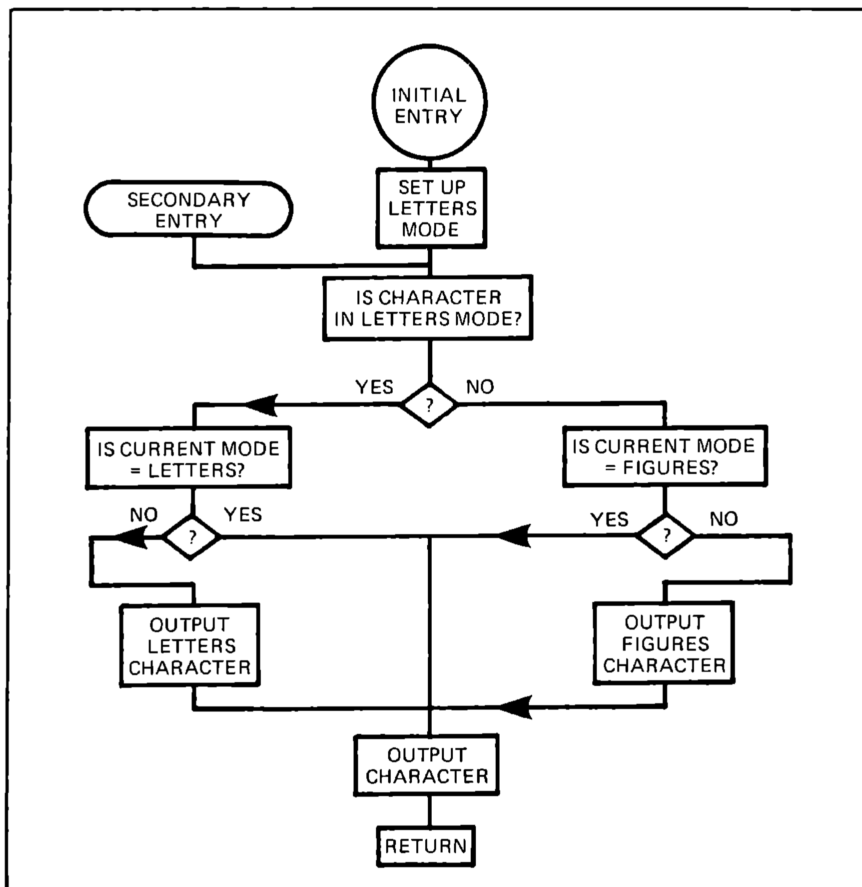
```
INBAUD   JSR INPUT     Now accept BAUDOT char from keyboard
         CMP #$1B      See if figures code
         BEQ FIGCOD    Yes, set CNTRL to $20
         CMP #$1F      See if letters code
         BEQ LETCOD    Yes, set CNTRL to $00
         CLC           Clear carry for addition
         ADC CNTRL     Add condition of sixth bit
         RTS           Return to process data
FIGCOD   LDY #$20      Set sixth bit of CNTRL
         STY CNTRL     By loading it with $20
         RTS
LETCOD   LDY #$00      Clear sixth bit of CNTRL
         STY CNTRL     By loading it with $00
         RTS
```

Two subroutines are called out in this listing to perform the input and output operations with the BAUDOT devices. The INPUT routine inputs a character from the BAUDOT keyboard, and returns to this routine with that character in the accumulator. The OUTPUT routine must transmit the character contained in the accumulator to the BAUDOT output device. The reader may refer to Chapter Seven for methods of implementing these INPUT and OUTPUT routines.

Another routine may be used to decode BAUDOT characters before outputting. It also has two entry points. The BAUDOT entry point is called when the initial character of the string of characters is to be output. This entry point sets up the output device and the CNTRL memory location to the letters mode before outputting the character. After the first character, the subsequent characters are output by using entry point OTBAUD. OTBAUD first checks the character to be output for a change from the current mode. If different, the proper mode control character will be output before the character. The character to be output must be stored in the Y index register before calling either of these entry points. The OUTPUT routine must function in the same manner as previously described.

```
BAUDOT   LDA #$1F      Load letters code into accumulator
         JSR OUTPUT    Call routine to send BAUDOT code
         LDA #$00
         STA CNTRL     Reset CNTRL to letters code
OTBAUD   TYA           Fetch character to output
```

```
         AND #$20       Is figures character indicated?
         BEQ LTCHAR     No, character is a letter
         LDA CNTRL      Figure, was last output a figure
         BEQ LASLET     No, must output figure code
OUTCOD   TYA            Put character in accumulator
         JSR OUTPUT     Output the BAUDOT character
         RTS            Return to calling program
LASLET   LDA #$20       Last was letter, set CNTRL
         STA CNTRL      To indicate figure code
         LDA #$1B       Output figure code
LASFIG   JSR OUTPUT     Send control character
         JMP OUTCOD     Now send character
```



```
LTCHAR    LDA CNTRL     See if last was letter
          BEQ OUTCOD    Yes, output character
          LDA #$00
          STA CNTRL     Reset CNTRL to letter mode
          JMP LASFIG    By using routine above
```

From ASCII to BAUDOT and Back

Using the ASCII and BAUDOT codes as the sample codes, two methods of code conversion will now be presented. The first method uses a look-up table. The table consists of both the ASCII and BAUDOT codes for each character of the character sets. The entries in the table are arranged in pairs. The first entry of a pair contains the ASCII code for the character, and the second entry contains the BAUDOT code for the same character. In cases where there is no equivalent BAUDOT code for a character, an appropriate substitute may be inserted (for example, the BAUDOT code for the left and right parenthesis, (and), may be substituted as the equivalent code for the ASCII left and right brackets, [and]). The BAUDOT null character is used when no suitable substitute is available.

The conversion program that uses this table begins at one end of the table and compares the character code to be converted against the entries in the table of the same character set. For conversion from ASCII to BAUDOT, the ASCII code to be converted is compared to the ASCII entries in the table. When a match is found, the BAUDOT entry of the pair is returned as the BAUDOT equivalent. A similar process is used to convert BAUDOT to ASCII. A flow chart indicates the logic used for conversion in either direction.

Address		Hexa Code	
0700	ASBDTB	C1	ASCII A
0701	BDASTB	03	BAUDOT A
0702		C2	ASCII B
0703		19	BAUDOT B
.		.	
073C		FF	ASCII RUBOUT
073D		00	BAUDOT NULL
073E		A0	ASCII SPACE
073F		04	BAUDOT SPACE
0740		A1	ASCII !
0741		2D	BAUDOT !
0742		A2	ASCII "

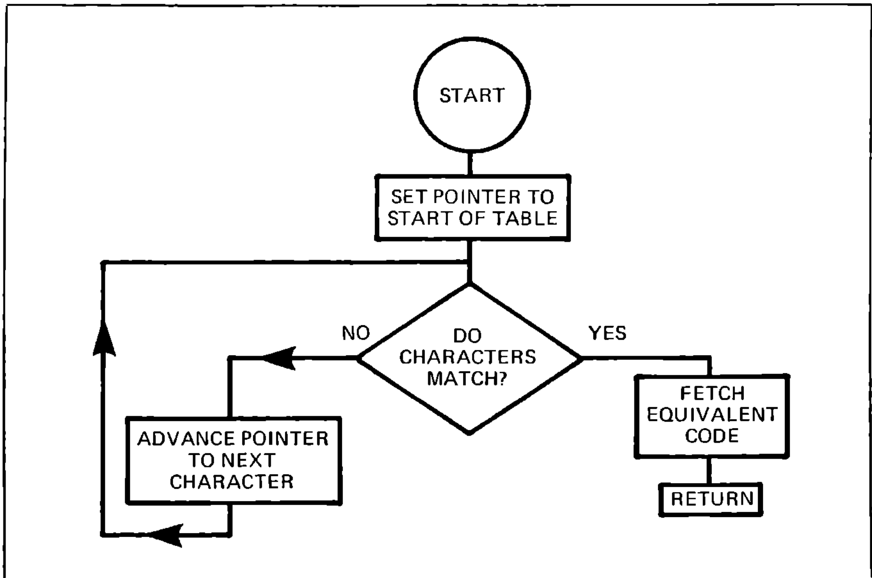
0743	31	BAUDOT "
0744	A3	ASCII #
0745	34	BAUDOT #
.	.	
.	.	
.	.	
075E	AF	ASCII /
075F	3D	BAUDOT /
0760	A8	ASCII (
0761	2F	BAUDOT (
0762	A9	ASCII)
0763	32	BAUDOT)
0764	DB	ASCII [
0765	2F	BAUDOT ((substitute)
0766	DD	ASCII]
0767	37	BAUDOT) (substitute)
.	.	
.	.	
077C	BF	ASCII ?
077D	39	BAUDOT ?
077E	C0	ASCII @
077F	00	BAUDOT NULL (substitute)

One rule must be followed. When substitute characters are used, the true code for a conversion must be located such that it will be found before the substitute codes are encountered. For example, the ASCII and BAUDOT pairs for left and right parenthesis must be placed so that conversion from BAUDOT to ASCII will find the ASCII code for left and right parenthesis, not left and right brackets. This positioning is illustrated in the table.

Listings for conversion routines from ASCII to BAUDOT, and vice versa, using the look-up table are now presented. The code to be converted must be in the accumulator when the routine is called. Also, the converted data is returned to the calling program in the accumulator, resulting in a loss of the initial character.

```
ASBAUD   LDX #$00        Initialize table pointer
FASCII   CMP ASBDTB,X    Is character equal to table entry?
         BEQ FNDBDO      If match, do conversion
         INX             No match, advance pointer
         INX             To next ASCII code
```

```
         JMP FASCII        ††Continue looking
FNDBDO   LDA ASBDTB+1,X    Fetch BAUDOT equivalent
         RTS               Return with code in accumulator
BAUDA$   LDX #$00          Initialize table pointer
FBAUDO   CMP BDASTB,X      Is character equal to table entry?
         BEQ FNDASC        If match, do conversion
         INX               No match, increment pointer
         INX               To next BAUDOT code
         JMP FBAUDO        ††Continue looking
FNDASC   LDA BDASTB−1,X    Fetch ASCII equivalent
         RTS               Return with code in accumulator
```



Watch for the Table's End

Both of these routines assume that the code to be converted is valid (one which is included in the table). If, for some reason, the accumulator does not contain a valid code, the table will be overshot. It is for this reason that a test for the end of the table should

be added. The following instruction sequence may be inserted in place of the JMP instructions marked by the ††.

The immediate portion of the CPX instruction must be set equal to the number of entries in the table. The value is $80. If the end of the table is reached without a match, some method is needed to inform the calling program of the error. One method, indicated in this instruction sequence, might be to set the accumulator equal to an invalid code.

```
CPX #$80        Compare X to the table count
BNE FZZZZZ      Not end, continue search at
                  FASCII or FBAUDO
LDA #$40        End of table, return with A = $40
RTS
```

The Input Points the Way

Another code conversion is to form a pointer out of the character code to be converted. This pointer is used to point to the corresponding code in a conversion table. The conversion table contains a list of the conversion codes. Each entry is located at the address in the table to which the code to be converted will point when the pointer is formed.

In the following example, the conversion from ASCII to BAUDOT is made by resetting the two most significant bits of the ASCII code to zero forming a pointer to the corresponding BAUDOT code in the conversion table. This method of setting up the pointer means the table must begin at location 00 of the page on which it resides. If it does not, a displacement constant must be added to the pointer to properly adjust it. For this routine, it is assumed that the table begins at location 00.

The conversion table uses 64 memory locations. Each one contains the BAUDOT codes for the characters in the order corresponding to the pointer formed by the equivalent ASCII code. As in the previous look-up table, the use of substitute characters is required at the locations in the table for which no BAUDOT equivalents exist. Therefore, the first table entry is the null character, since an @ does not exist in the BAUDOT code. The next entry is the BAUDOT code for an A, then B, and so on.

Yield to Nonsequential Characters

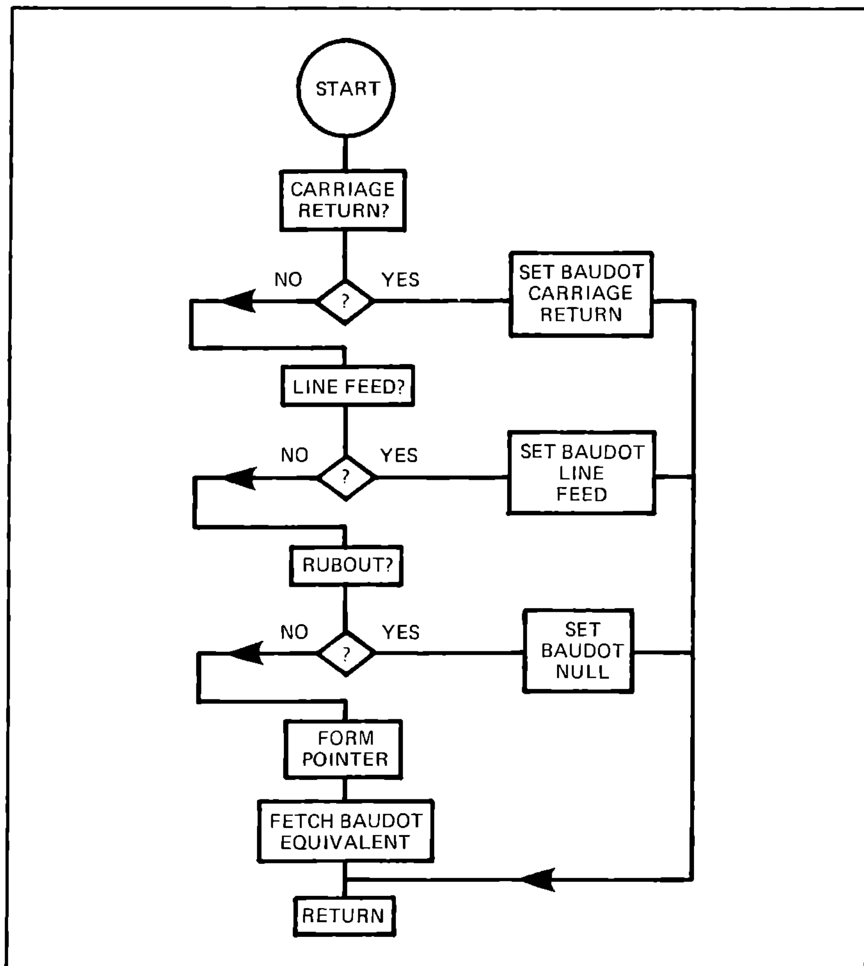
A special condition arises when the characters such as carriage return, line feed, and rubout are converted. In forming the

pointer for these characters, the carriage return forms a pointer to the same location as the letter M and the line feed forms a pointer to the same location as the "?." Because only three characters of this type need to be converted to BAUDOT, the conversion routine can check for their individual codes before forming the pointer. This eliminates the possibility of erroneous conversion. However, if the codes being converted have ten or more codes that overlap in this fashion, it would be more efficient (in memory usage) to expand the conversion table from 64 entries to 128, and to zero only the MSB of the ASCII code to form the pointer. This means that there will be more substitute characters contained in the table. But, the actual conversion routine will not have to check each code for special characters.

The conversion routine shown below uses the pointer technique to convert from ASCII to BAUDOT, with special consideration given to the carriage return, line feed, and rubout characters. The X index register is set up as the pointer. This routine assumes that the ASCII code of the character to be converted is contained in the accumulator when the routine is called. The converted code is returned in the accumulator.

```
ASBDPT   CMP #$8D      Carriage return?
         BEQ CARRET    Yes, fetch BAUDOT carriage return
         CMP #$8A      Line feed?
         BEQ LINFED    Yes, fetch BAUDOT line feed
         CMP #$FF      Rubout?
         BEQ RUBOUT    Yes, fetch BAUDOT null
         AND #$3F      Mask off 2 MSB of ASCII
         TAX           Form pointer to conversion table
         LDA BDOTBL,X  Fetch BAUDOT code from table
         RTS           Return with code in accumulator
CARRET   LDA #$08      Set BAUDOT carriage return
         RTS           Return
LINFED   LDA #$02      Set BAUDOT carriage return
         RTS           Return
RUBOUT   LDA #$00      Set BAUDOT null
         RTS           Return
0400     BDOTBL        00          NULL FOR @
0401                   03          A
0402                   19          B
                                   INSERT BAUDOT
                                    CODES C TO Y FROM
```

041A	11	Z
041B	2F	(FOR [
041C	00	NULL FOR
041D	32	) FOR]
041E	00	NULL FOR ↑
041F	00	NULL FOR ←
0420	04	SPACE
0421	2D	!
0422	31	"



0423	34	#
0424	29	$
0425	00	NULL FOR %
0426	3A	&
0427	2B	'
0428	2F	(
0429	32	)
042A	00	NULL FOR *
042B	00	NULL FOR +
042C	2C	,
042D	23	-
042E	3C	.
042F	3D	/
0430	36	0
0431	37	1
0432	33	2
0433	21	3
0434	2A	4
0435	30	5
0436	35	6
0437	27	7
0438	26	8
0439	38	9
043A	2E	:
043B	3E	;
043C	00	NULL FOR <
043D	00	NULL FOR =
043E	00	NULL FOR >
043F	39	?

Things to Consider

There are several considerations when choosing which method to use for code conversion. The first one is whether the conversion will be made in both directions (from code A to code B for input, and then code B back to code A for output), or only one direction. If conversion is in one direction only, the pointer method would shorten the table space required because only one code is included in the table area. For conversion in both directions, either method results in approximately the same memory requirement unless the table for the pointer method has gaps of unused locations caused by the code forming the pointer having a nonsequential bit pattern.

For programs requiring speed of conversion, the pointer method is the choice. It provides the correct code with just a single pass through the instruction sequence. The look-up table method will remain in a loop until the correct code is found. This means that it could take up to 60 or more times longer than the pointer method to make a single conversion.

Numeric Conversion Is Quite Common

Another common type of conversion is the conversion of numeric data from one number base to another. The typical conversion is from decimal to binary, and binary to decimal. The reason this conversion is common is because the decimal number system is used in real world mathematical and numeric applications, while the computer is generally designed to operate with binary numbers. Thus, to allow the real world and the computer to operate in their most desirable number systems, the conversion of decimal to binary, and vice versa, is required.

The first routine converts a number designated by decimal digits in binary-coded decimal format to the equivalent triple precision binary value. The decimal digits are contained in a table labeled DECMAL, with one BCD digit stored per byte. The table BINVAL consists of three consecutive memory locations used to store the binary number. This triple precision representation allows conversion of decimal values from 0 to 16, 777, 215. The routine starts with the most significant decimal digit and works down to the least significant one.

The major part of the conversion is done by a subroutine that multiplies the current contents of BINVAL by ten, and then adds one decimal digit to this new value. This subroutine, labeled TIMS10, performs the multiplication by a series of rotate and addition operations, as explained in the commented portion of the listing. Several of the subroutines presented in Chapter Three are used by this subroutine to aid in performing its function.

The data table that precedes this listing defines the locations used by both the binary-to-decimal and decimal-to-binary conversion routines for storing temporary data. This table indicates the number of memory bytes to be assigned to each label. The *=*+ in the mnemonic column is an assembler directive. It informs the assembler program of the number of bytes to be reserved for the indicated label.

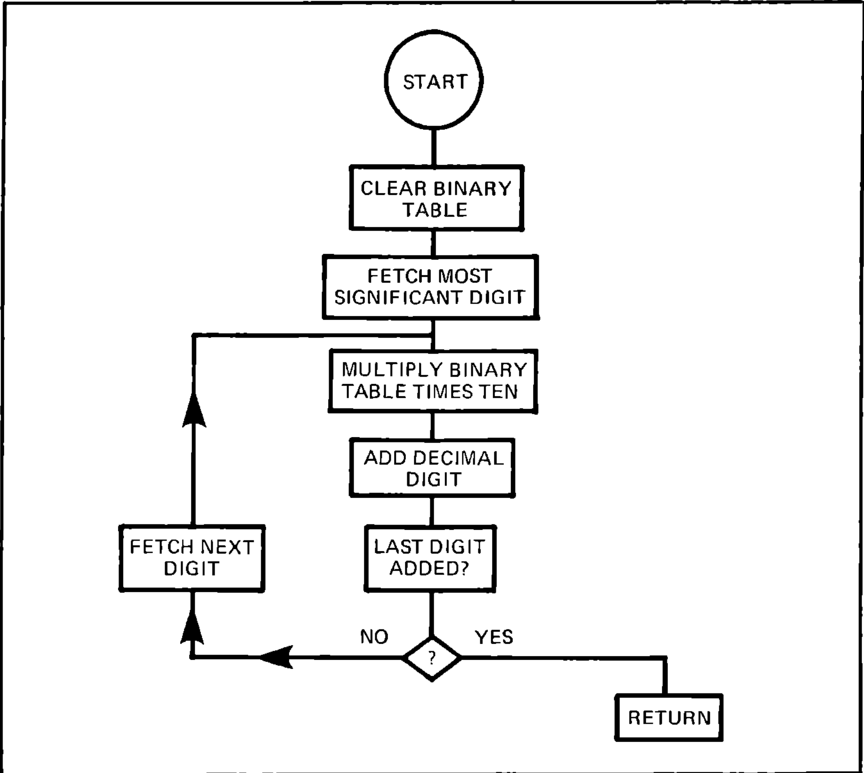
```
FMPNT     *=*+2          Temporary pointer storage
```

```
TOPNT    *=*+2          Temporary pointer storage
DECTBL   *=*+1          Pointer to DECMAL table
DGTCNT   *=*+1          Counter storage for BNTODC
BINVAL   *=*+3          Binary equivalent storage
WRKARA   *=*+3          Temporary working area
DECMAL   *=*+7          Decimal equivalent storage
DECML8   *=*+1          M.S. digit of decimal equivalent
TIMS10   PHA            Save digit to be added
         LDX #BINVAL    Set up pointer to BINVAL
         STX FMPNT      Store in FMPNT
         LDA #WRKARA    Set up pointer to WRKARA
         STA TOPNT      Store in TOPNT
         LDA #$00       Set up page portion of pointers
         STA FMPNT+1    Store in FMPNT
         STA TOPNT+1    And TOPNT
         LDY #$03       Set precision counter
         JSR ROTATL     Multiply BINVAL X 2
         LDX #$03       Set precision counter
         JSR MOVIND     Move BINVAL X 2 to WRKARA
         LDX #BINVAL    Set pointer to rotate BINVAL left
         LDY #$03       Set precision counter
         JSR ROTATL     Multiply (BINVAL X 2) X 2 (total =X4)
         LDX #BINVAL    Set pointer to rotate BINVAL left
         LDY #$03       Set precision counter
         JSR ROTATL     Multiply BINVAL X 4 X 2 (total =X8)
         LDX #$03       Set precision counter
         JSR ADDER      Add (BINVAL X 2) + (BINVAL X 8)
         PLA            Fetch decimal digit from stack
         LDX #WRKARA    Set pointer to WRKARA
         STX FMPNT      Store pointer in FMPNT
         STA BINVAL     Load BINVAL with decimal digit
         LDA #$00       Load remainder of BINVAL
         STA BINVAL+1   With zero
         STA BINVAL+2
         LDX #$03
         JSR ADDER      Add BINVAL X 10 to new digit
         RTS            Return with sum in BINVAL
```

Decimal to Binary Conversion

The DCTOBN routine fetches the BCD digits from the DECMAL table for conversion to binary by the TIMS10 subroutine. First, using the CLRMEM subroutine, the three words used for the

binary number storage are cleared. The routine then fetches one decimal digit at a time, beginning with the most significant digit, and calls the TIMS10 subroutine to add it to the binary value. Then the conversion is complete and the routine returns to the calling program. Once again, this routine assumes the decimal digits are stored in the DECMAL table in BCD format, one digit per byte, before being called. This routine begins at the label DCTOBN.




```
DCTOBN   CLD              Clear decimal mode flag
         LDX #BINVAL      Set pointer to BINVAL
         STX TOPNT        Store in TOPNT
         LDX #$0          Set page portion of pointer to zero
         STX TOPNT+1      Store in TOPNT
         LDX #$03         Set precision counter
         JSR CLRMEM       Clear binary storage area
         LDX #DECML8      Set pointer to MS deçimal digit
```

```
          STX DECTBL      Store in temporary storage
DBCNVT    LDA VALUE,X     Fetch decimal digit
          JSR TIMS10      Multiply BINVAL X 10 and add digit
          DEC DECTBL      Decrement decimal pointer
          LDX #DECMAL-1   Is last decimal digit added?
          CPX DECTBL      To BINVAL?
          BNE DBCNVT      No, continue process
          RTS             Yes, return with sum in BINVAL
```

The TIMS10 subroutine may be inserted in place of the JSR TIMS10 instruction, rather than being set up as a subroutine. This has not been done to call attention to the portion of the routine that performs the actual conversion. Also, this subroutine is used in Chapter Five to convert the decimal numbers directly to their binary equivalents as they are entered by the operator.

Binary-To-Decimal Conversion

This routine performs the reverse function of the DCTOBN one. It converts the triple precision binary value in BINVAL to the equivalent eight-digit decimal number and is stored in the DECMAL table. This routine is called BNTODC.

BNTODC uses a subroutine labeled DCEQVL to perform the actual conversion of the binary value to decimal. The conversion is made by subtraction of a binary constant equal to the decimal power of ten. When this subroutine is called, the pointer TOPNT must contain the address of the least significant byte of the power of ten to the subtracted. The indicated power of ten is then subtracted from the binary value being converted. When the result of the subtraction requires a borrow for the MSB (indicated by the carry flag being reset after the subtraction), the current power of ten is added back to the binary value to correct for the last subtraction. The memory location labeled DECCNT contains the decimal value for the power of ten being subtracted when the subroutine returns.

As an example, suppose the binary value of one million can be subtracted five times from the binary number before the borrow occurs. The value of five would be the seventh digit of the decimal equivalent.

This subroutine, like the TIMS10 subroutine in the previous conversion routine, can be placed in line with the BNTODC instruction sequence. By replacing the JSR DCEQVL instruction, one can shorten the memory required as well as the execution time. It is

presented as a subroutine to bring out the significance of its operation to this conversion routine.

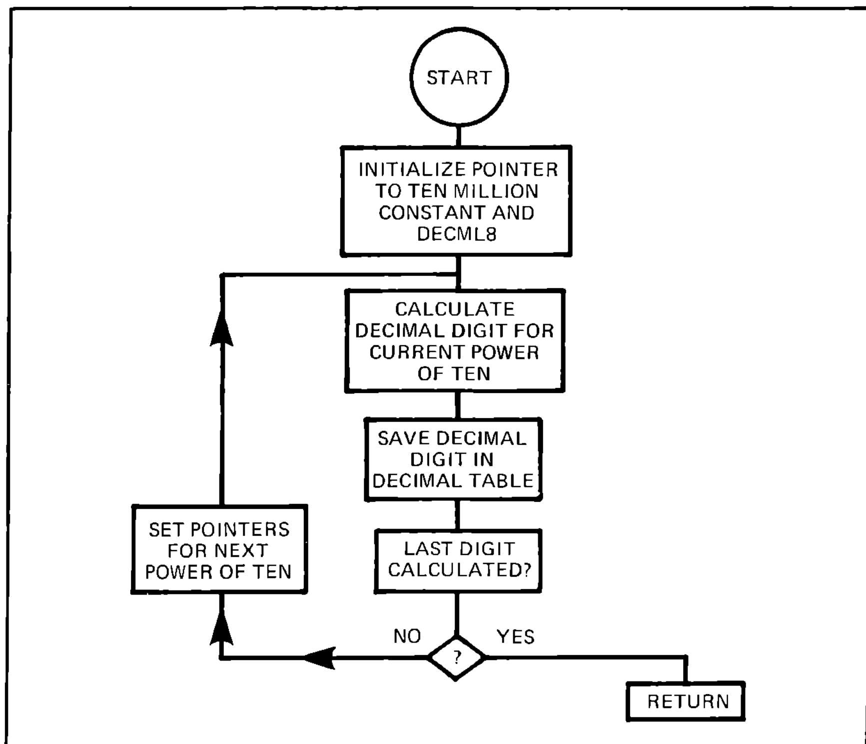
```
DCEQVL   LDA #$00       Set up initial decimal counter
         STA DGTCNT     Store zero in DGTCNT
         LDX #BINVAL    Set pointer to BINVAL
         STX TOPNT      Store in TOPNT
         STA TOPNT+1    Set page portion of TOPNT
         LDX #$03       Set precision counter
DCLOOP   JSR SUBBER     Subtract binary constant
         LDX #$03       Set precision counter
         BCS INCRVL     No borrow, increment decimal count
         JSR ADDER      Add constant back to BINVAL
         RTS            Return with digit in DECCNT
INCRVL   INC DGTCNT     Increment decimal counter
         JMP DCLOOP     Continue subtraction
```

The BNTODC routine sets up and keeps track of the current power of ten being subtracted from the binary value by DCEQVL. As each power of ten is subtracted and the value of the respective decimal digit value is returned in DGTCNT, BNTODC stores the decimal digit value in DECMAL. It then advances to the next lower power of ten. When the decimal value of one has been subtracted, the subroutine returns with the decimal equivalent stored in the DECMAL table. It is important to note that the value in BINVAL will be zero when the conversion is complete. The calling program must save the original value of BINVAL if it is required after the conversion.

The listing and flow chart are presented here. The assembler directive .BYTE is used in this table. It informs an assembler program to assign one byte for each of the values which follow it.

```
BNTODC   CLD            Clear decimal mode flag
         LDX #DECML8    Set pointer to decimal storage
         STX DECTBL     Save pointer in DECTBL
         LDX #<TENMIL   Set pointer to binary constant
         STX FMPNT      Store pointer in FMPNT
         LDX #>TENMIL   Set page portion of constant pointer
         STX FMPNT+1    Store in FMPNT
BNDC     JSR DCEQVL     Calculate decimal value of digit
         LDX DECTBL     Fetch pointer to decimal storage
         LDA DGTCNT     Fetch digit just calculated
```

```
          STA VALUE,X     Store in decimal table
          DEC DECTBL      Back up table pointer
          LDA FMPNT       Fetch pointer to constant table
          CLC             Clear carry for addition
          ADC #$03        Advance pointer to binary constants
          STA FMPNT       Store in FMPNT
          CMP #<ONE+3     Is pointer at end of table?
          BNE BNDC        No, continue conversion
          RTS             Yes, return
TENMIL    .BYTE           $80,$96,$98  Ten million in binary
ONEMIL    .BYTE           $40,$42,$0F  One million in binary
HUNTHO    .BYTE           $A0,$86,$01  One hundred thousand in
                                         binary
TENTHO    .BYTE           $10,$27,$00  Ten thousand in binary
ONETHO    .BYTE           $E8,$03,$00  One thousand in binary
HUNRED    .BYTE           $64,$00,$00  One hundred in binary
TEN       .BYTE           $0A,$00,$00  Ten in binary
ONE       .BYTE           $01,$00,$00  One in binary
```



Chapter 5

Floating Point Routines

Complex mathematics is one of the key functions for which a computer is best suited. Those unfamiliar with the techniques involved often consider such programs too complicated a task to undertake. However, if one takes the time to break down the basic operations, the overall algorithms are not quite so difficult.

The digital computer is capable of performing mathematical operations with numbers of considerable magnitude. This is possible by representing numbers as multiple precision values in which more than one memory location is used to hold the numeric information. However, by increasing the number of locations assigned to represent a number, one could reach a point where the least significant bits become too insignificant with respect to the total value. A more practical representation would be to condense the size of the required number of significant digits. The overall magnitude of the value may be indicated by a power of the number base. This representation is referred to as floating point format.

Format of Floating Point Numbers

Floating point format allows one to define a number as a product of two values. The first value contains the significant digits of the number. This value is referred to as the "mantissa." It should contain as many significant digits as needed to properly define its relative value. The second value contains the power to which the number base is to be raised. This value, called the "exponent," indicates the magnitude of the significant digits of the mantissa. For example, the decimal value 1,000,000 would require a triple precision binary number to be properly represented. However, this same value can be defined as "1 X 10**6," or, in floating point notation, "1.0 E+6." This form contains the mantissa, 1, which is

the single significant digit of this value, and the exponent 6, which indicates the power of ten, or the number of places the decimal point should be moved to the right. This shorter notation also requires fewer memory locations to represent the indicated value — one location to contain the significant digit (1), and a second to hold the exponent value (6).

One more advantage this notation has over the individual multiple precision value is the capability to represent fractional numbers. By providing a sign bit for the exponent, negative, as well as positive values of the exponent can be expressed. Remember, a negative exponent forms the reciprocal of the power. For base ten, the exponent, —1 would indicate the value of 1/10 times the mantissa. The negative exponent moves the decimal point to the mantissa one place to the left for each integer value of the exponent.

This notation can be used to represent binary numbers as well. The binary mantissa contains the significant bits of the binary value. The binary exponent will indicate the power of two to which the mantissa is raised, thereby indicating the location of the decimal point (or, to properly refer to it, the binary point). The same properties of the decimal exponent apply to the exponent for the binary numbers. If the exponent is positive, the binary point in the mantissa is actually located to the right by the number of places indicated by the exponent. A negative exponent shifts the binary point to the left. Putting it in more relative terms, if the mantissa is shifted to the right, the exponent must be incremented. Shifting the mantissa to the left means the exponent must be decremented. The following illustrates three ways of expressing the same number in binary floating point format.

```
  101.0 E + 0 = 5 X 1 = 5
   .101 E + 3 = 5/8 X 8 = 5
101000.0 E — 3 = 40 X 1/8 = 5
```

This notation may be used to represent a wide range of values with a minimum number of memory locations. One or more memory locations may be set up to store the mantissa and the exponent. The number of locations used will depend on the number of significant bits desired to express each quantity.

The floating point routines to be presented in this chapter operate with binary floating point numbers in the following format. Each number will be stored in four memory locations. The first location will contain the exponent with the most significant bit

indicating the sign of the exponent. The sign bit will indicate a positive number if reset to zero, and a negative exponent if set to one. The next three locations will be used to store the mantissa as a triple precision binary number. The most significant bit of the most significant byte is used to indicate the sign of the mantissa. The binary point will always be implied to be to the right of the sign bit in the mantissa. One should note that there is no implied binary point in the exponent since the exponent is always assumed to be an integer value. This format is illustrated below.

Exponent	MSB	Mantissa	LSB
SEEEEEEE	S.MMMMMMM	MMMMMMMM	MMMMMMMM
MEM LOC N+3	MEM LOC N+2	MEM LOC N+1	MEM LOC N

The order for storing the data in the memory location should be noted. The exponent is stored in the highest address of the four locations used to store the floating point number. Also, since the sign bit takes up one bit for both the mantissa and the exponent, the number of bits used to represent each value is 23 (decimal) and 7, respectively.

Before presenting the floating point routines, it should be noted that various locations on page 00 are used for data storage. This data includes pointers and counters required at different times, several temporary storage tables, and two areas that are frequently used as operating registers. These two areas shall be referred to as the floating point accumulator and the floating point operand. The floating point accumulator is used as the accumulator of the floating point routines in performing calculations and storing the results of the operations performed. The floating point operand is used to store and manipulate the number operated on by the accumulator. These two locations will have the same format as defined previously for the floating point numbers. The floating point accumulator and operand shall be abbreviated as FPACC and FPOP throughout the remainder of this chapter.

Floating Point Normalization

The first routine is used to adjust the floating point numbers to a common format. This format is required for proper operation of the other floating point routines. In order for the floating point arithmetic routines to operate with the highest degree of accuracy possible, the value in the FPACC must be adjusted to a standard representation before the operations are performed. This represen-

tation is referred to as the normalized value. A number is considered to be normalized when the mantissa's most significant bit with a value of "1" is immediately to the right of the implied binary point. If this bit is not a "1," the number is normalized by shifting the mantissa to the left until the most significant "1" is to the right of the implied binary point. For each bit position shifted to the left in the mantissa, the corresponding exponent must be decremented to maintain the actual value of the number. The resultant value of the mantissa will be a number greater than or equal to one half, and less than one. This process is illustrated below:

BEFORE NORMALIZATION	0.00011011100011000011010 E+0
AFTER NORMALIZATION	0.11011100011000011010000 E−3

The process of normalizing a floating point number is required to set up the values in a common format with which the other routines can work effectively. Also, normalizing a number allows more significant digits in the mantissa. By insuring that one is using the highest number of digits possible, the accuracy of the calculations will be increased.

The normalization routine is written to operate with positive mantissa values. If the number to be normalized is negative, this routine will convert it to its two's complement form before normalizing, and then complement it again after the normalization. The following example illustrates the process for normalizing the value −5, as it may appear after an arithmetic operation.

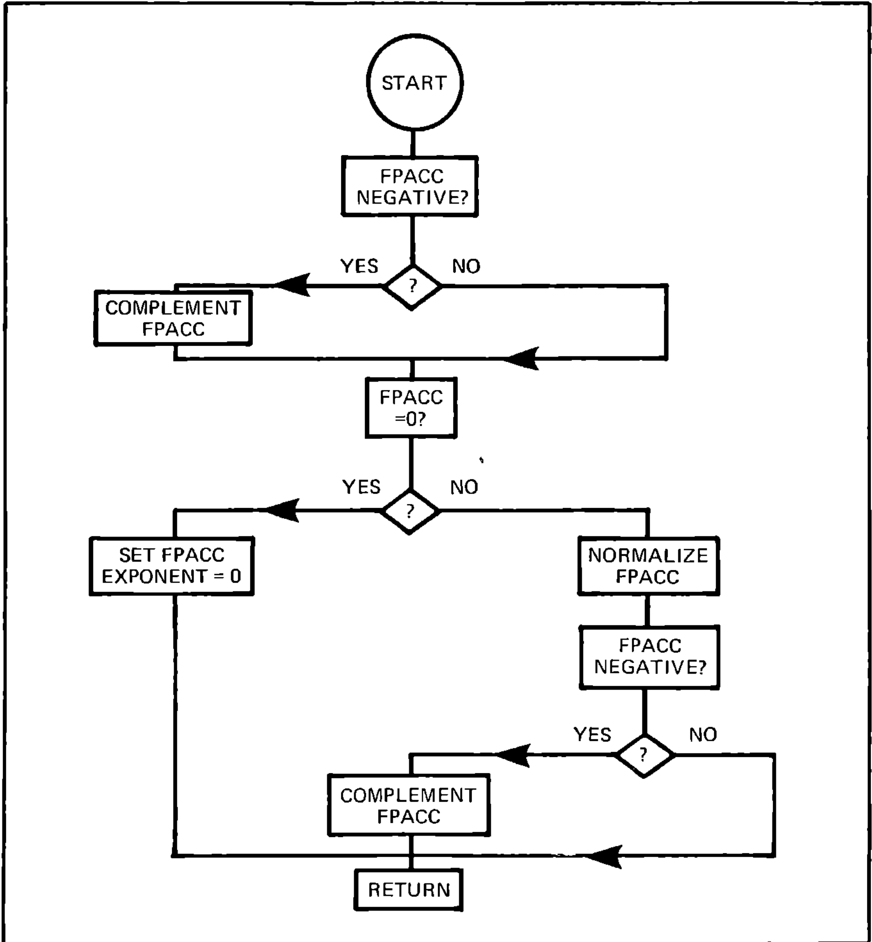
INITIAL VALUE	1.1111111	011000000	00000000	E $0A
COMPLEMENTED	0.0000000	101000000	00000000	E $0A
NORMALIZED	0.101000	000000000	00000000	E $03
COMPLEMENTED	1.011000	000000000	00000000	E $03

One special test must be made by this routine. It must check for an initial mantissa value of zero. If the mantissa is initially all zeros, and the normalization routine is allowed to perform its normal sequence, it would become caught in an endless loop looking for the first "one" bit. Therefore, to eliminate this possibility, the FPACC mantissa is initially checked for a value of zero. If found, the FPACC exponent is zeroed and the routine returns.

The routine uses four memory locations for the mantissa in the initial stages of the process. This is necessary to handle some special cases that occur in the multiplication routine that require the addi-

tional precision. For the routines that do not use the additional byte, the least significant byte minus one of the mantissa must be set to zero before calling the FPNORM routine.

```
FPNORM   LDX #TSIGN     Set pointer to sign register
         LDA FPMSW      Fetch FPACC MS Byte
         BMI ACCMIN     If negative, branch
         LDY #$00       If positive, clear sign register
         STY PAGE0,X    By storing  zero
         JMP ACZERT     Then test if FPACC=0
ACCMIN   STA PAGE0,X    Set sign indicator if minus
```



```
        LDY #$04       Set precision counter
        LDX #FPLSWE    Set pointer to FPACC LS Byte -1
        JSR COMPLM     Two's complement FPACC
ACZERT  LDX #FPMSW     Set pointer to FPACC MS Byte
        LDY #$04       Set precision counter
LOOK0   LDA PAGE0,X    See if FPACC=0
        BNE ACNONZ     Branch if nonzero
        DEX            Decrement index pointer
        DEY            Decrement byte counter
        BNE LOOK0      If counter not zero, continue
        STY FPACCE     FPACC = 0, clear exponent, too
NORMEX  RTS            Exit normalization routine
ACNONZ  LDX #FPLSWE    Set pointer to FPACC LS Byte -1
        LDY #$04       Set precision counter
        JSR ROTATL     Rotate FPACC left
        LDA PAGE0,X    See if one in MS Bit
        BMI ACCSET     If minus, properly justified
        DEC FPACCE     If positive, decrement FPACC exponent
        JMP ACNONZ     Continue rotating
ACCSET  LDX #FPMSW     Set pointer to FPACC MS Byte
        LDY #$03       Set precision counter
        JSR ROTATR     Compensating rotate right FPACC
        LDA TSIGN      Is original sign positive
        BEQ NORMEX     Yes, simply return
        LDY #$03       With pointer at LS Byte, set precision
                         counter
        JMP COMPLM     Restore FPACC to negative and return
```

Several of the Chapter Three subroutines are used here. These are the ROTATL, ROTATR and COMPLM subroutines. Throughout the remainder of the floating point routines, these and other subroutines, such as MOVIND, CLRMEM and ADDER will be called upon to perform their various functions.

Floating Point Addition

The basic function of this routine is carried out by the ADDER subroutine. However, there are a number of conditions that must be considered before the actual addition is performed.

First, the FPACC and the FPOP are tested for a value of zero. If both values are zero, or only the FPOP is zero, the routine can be exited immediately, since the answer is already in the FPACC. (Remember, the results of all floating point operations are returned

in the FPACC!) If the FPACC is zero, the contents of the FPOP are transferred to FPACC before returning.

Should both numbers contain values other than zero (as is most likely the case when FPADD is called) the relative magnitude of one number to the other must be compared. With both numbers expressed in floating point notation, the range of values can vary quite a bit. For the addition routine, there is a limit in which the relative magnitude of the two numbers must fall. If one value is so much larger than the other, that the significant digits of the smaller are outside the range of the significant digits of the larger, the addition would result in no change to the larger number. The answer would simply be equal to the larger number. This range is equal to the number of bits used to represent the value of the mantissa. For the floating point format used by these routines, the allowable limit on the difference between the two exponents is 23. If the difference is greater than 23, the number of greater magnitude is returned in the FPACC as the answer.

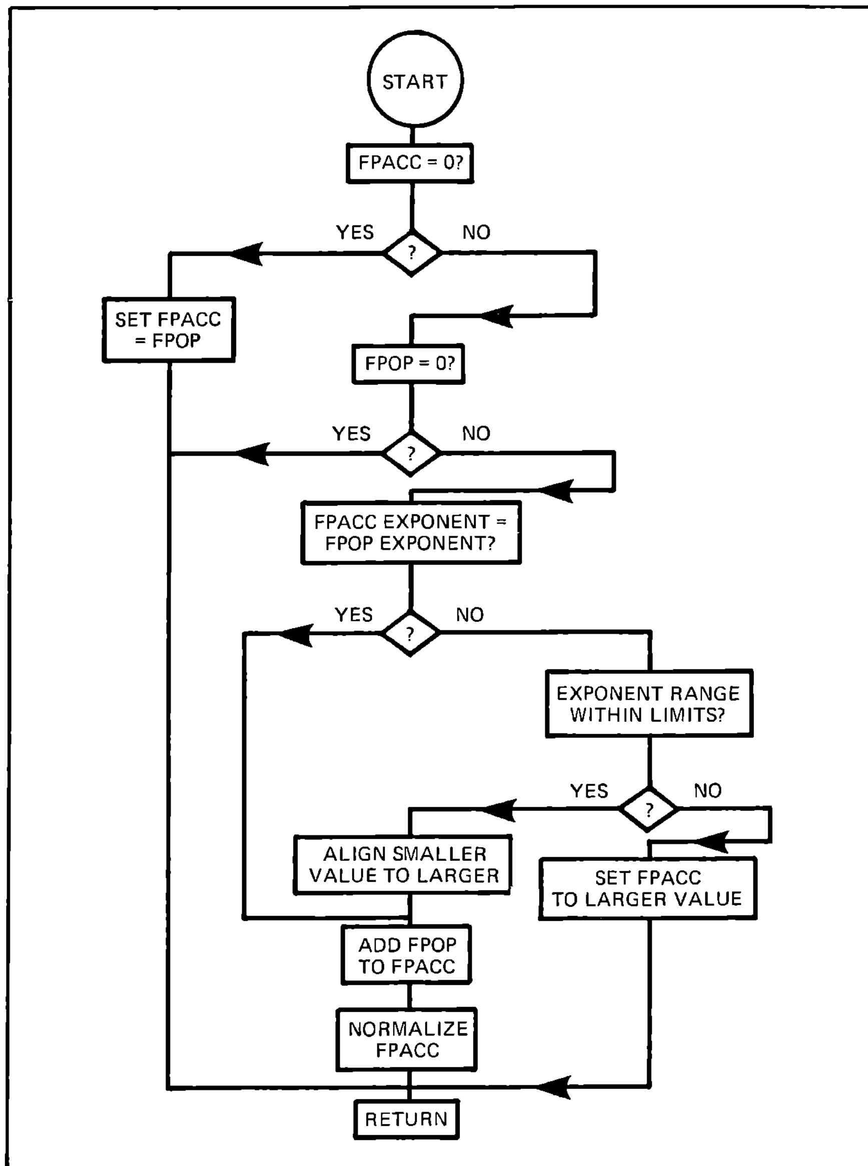
Assuming that the two numbers fall within the allowable range, the mantissas must be properly aligned before the addition can be executed. The two numbers are aligned when the exponents of each are equal. This alignment is made by shifting the mantissa of the smaller value to the right, while incrementing its exponent until it is equal to the exponent of the larger. Of course, if the exponents are equal at the start, this is not necessary. The only special consideration in this procedure is when the mantissa being shifted is negative. In this case, a "1" must be shifted into the MSB of the mantissa to maintain the negative condition. This is accomplished by setting the carry flag and calling the second entry point, ROTR, of the ROTATR subroutine. This will not clear the carry at the start of the rotate operation.

The final operation before the addition is performed is to shift the FPACC and FPOP one bit to the right. This leaves the MSB open to accept a possible overflow as a result of the addition. This eliminates the need to test the carry flag for an overflow when the addition is complete. Also, quad-precision is utilized in both the shifting and addition. This maintains the integrity of the LSB when the result of the addition is normalized.

```
FPADD    LDA FPMSW       See if FPACC MS Byte =0
         BNE NONZAC      Branch if not zero
MOVOP    LDX #FOPLSW     Set pointer to FPOP LS Byte
         STX FMPNT       Save in FMPNT
```

```
        LDX #FPLSW      Set pointer to FPACC LS Byte
        STX TOPNT       Save in TOPNT
        LDA #$00        Set page zero value
        STA FMPNT+1     Store in page portion of FMPNT
        STA TOPNT+1     And page portion of TOPNT
        LDX #$04        Set precision counter
        JMP MOVIND      Move FPOP to FPACC and return
NONZAC  LDA FOPMSW      See if FPOP MS Byte = 0
        BNE CKEQEX      No, check exponents
        RTS             Yes, return, result = FPACC
CKEQEX  LDX #FPACCE     Set pointer to FPACC exponent
        LDA PAGE0,X     Fetch FPACC exponent
        CMP FOPEXP      Is it equal to FPOP exponent?
        BEQ SHACOP      Branch ahead if equal
        SEC             If not equal, determine which is
                          larger
        LDA #$00        Form the two's complement of
        SBC PAGE0,X     The FPACC exponent
        ADC FOPEXP      Add in FPOP exponent
        BPL SKPNEG      If +, FPOP > FPACC
        SEC             If −, form two's complement
        STA TEMP1       Of the result
        LDA #$00        This will be used to test the
        SBC TEMP1       Magnitude of the difference in
                          exponents
SKPNEG  CMP #$18        Is difference < 18 hexadecimal?
        BMI LINEUP      If so, align the mantissas
        SEC             If not, is the FPOP > FPACC?
        LDA FOPEXP      This is tested by comparing
        SBC PAGE0,X     The exponents of each
        BPL MOVOP       FPOP larger, move FPOP to
                          FPACC
        RTS             FPACC larger, return
LINEUP  LDA FOPEXP      Fetch FPOP exponent
        SEC             Set carry for subtraction
        SBC PAGE0,X     Subtract FPOP—FPACC exponents
        TAY             Save difference in Y
        BMI SHIFTO      If neg., FPACC >, shift FPOP
MORACC  LDX #FPACCE     Set pointer to FPACC exponent
        JSR SHLOOP      Shift FPACC to right, one bit
        DEY             Decrement difference counter
        BNE MORACC      If not zero, continue
```

```
         JMP SHACOP      When zero, set up for addition
SHIFTO   LDX #FOPEXP     Set pointer to FPOP exponent
         JSR SHLOOP      Shift FPOP to right, one bit
         INY             Increment difference counter
```



```
          BNE SHIFTO      Not zero, continue
SHACOP    LDA #$00        Prepare for addition
          STA FPLSWE      Clear FPACC LS Byte -1
          STA FOLSWE      Clear FPOP LS Byte -1
          LDX #FPACCE     Set pointer to FPACC exponent
          JSR SHLOOP      Rotate FPACC right to allow for over-
                            flow
          LDX #FOPEXP     Set pointer to FPOP exponent
          JSR SHLOOP      Rotate FPOP right to keep alignment
          LDX #FOLSWE     Set pointer to FPOP LS Byte -1
          STX FMPNT       Store in FMPNT
          LDX #FPLSWE     Set pointer to FPACC LS Byte -1
          STX TOPNT       Store in TOPNT
          LDX #$04        Set precision counter
          JSR ADDER       Add FPOP to FPACC
          JMP FPNORM      Normalize result and return
SHLOOP    INC PAGE0,X     Increment exponent value
          DEX             Decrement pointer
          TYA             Save difference counter
          LDY #$04        Set precision counter
FSHIFT    PHA             Store difference counter on stack
          LDA PAGE0,X     Fetch MS Byte of value
          BMI BRING1      If negative, must rotate one in MSB
          JSR ROTATR      Positive, rotate value right one bit
          JMP RESCNT      Return to calling program
BRING1    SEC             Set carry to maintain minus
          JSR ROTR        Rotate value right one bit
RESCNT    PLA             Fetch difference counter
          TAY             Restore in Y
          RTS             Return
```

Floating Point Subtraction

Floating point subtraction may be derived by simply forming the two's complement of the value contained in the FPACC and then jumping to the FPADD routine, as the following FPSUB routine illustrates.

```
FPSUB     LDX #FPLSW      Set pointer to FPACC LS Byte
          LDY #$03        Set precision counter
          JSR COMPLM      Complement FPACC
          JMP FPADD       Subtract by adding negative
```

Floating Point Multiplication

Floating point multiplication is essentially carried out by a series of shifting and addition operations. As presented previously, a binary number is multiplied by two by simply shifting it one bit position to the left. With the proper addition function, one can create a multiplication algorithm for multiple precision binary numbers. This algorithm would operate in the following manner.

The two numbers to be multiplied shall be referred to as the multiplier and the multiplicand. A third register, called the partial-product, shall be used to store the product as it is being calculated. First, examine the LSB of the multiplier. If it is a "1," add the multiplicand to the partial-product register. After the addition, or if the LSB was zero, shift the multiplicand to the left, one-bit position (multiplying it by two). Examine the bit to the left of the LSB of the multiplier and, if it is a "1," add the current value of the multiplicand to the partial-product. Then, shift the multiplicand to the left again. The process continues for each bit of the multiplier, working up to the MSB. Each time the multiplier bit is equal to "1," the current multiplicand is added to the partial-product. The multiplicand is always shifted left following the examination of each bit of the multiplier (and addition to the partial-product if the bit is "1"). The result of the multiplication is contained in the partial-product register when the operation is complete.

The algorithm just described performs multiplication of standard binary numbers. Using this basic procedure, a multiplication algorithm for the mantissa in floating point format can be written. The following flow chart illustrates the process to be used to multiply the floating point values. The only major difference between the algorithm above and the process used by this floating point multiplication routine is that the partial-product is shifted right for each bit examined, rather than shifting the multiplicand to the left.

The exponent portion of the binary floating point numbers is manipulated in the same manner as the exponent of decimal floating point numbers for multiplication. They are simply added together.

The mantissa signs of both the multiplier and the multiplicand must be examined before the multiplication is executed. Since the multiplication algorithm only works for positive numbers, if either value is negative it must be two's complemented before multiplying. Also, following the laws of multiplication, if the two values are the same sign, the result will be positive; if the signs are

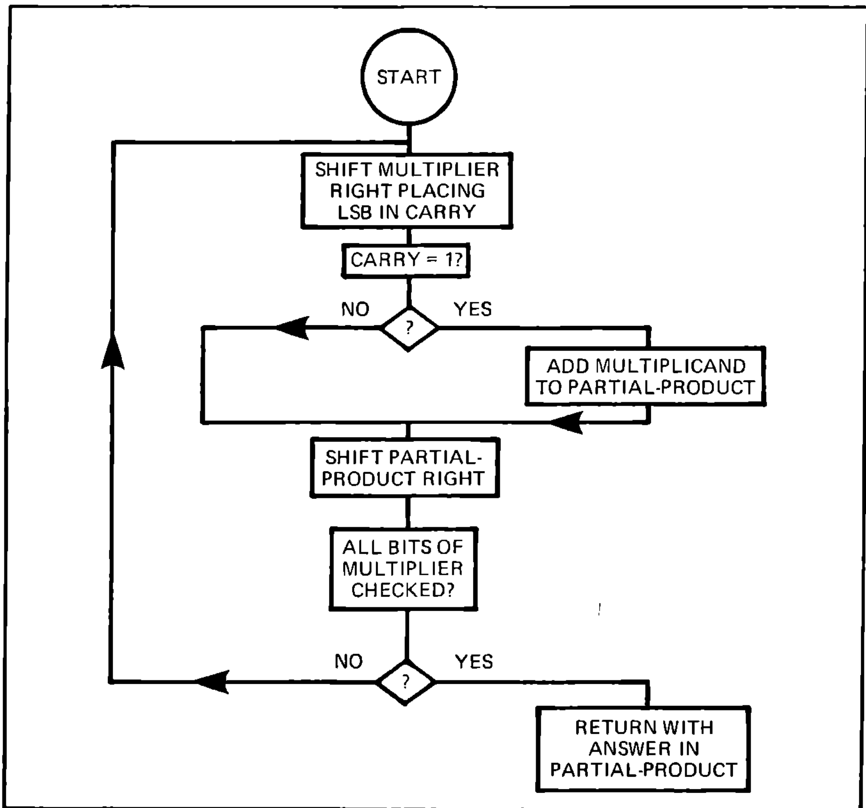
opposite, the result will be negative. This condition must be tested at the beginning, and, if the result is to be negative, the final value must be two's complemented before returning.

If the partial-product is rotated right for each bit of the multiplier, it is necessary for the partial-product register to contain twice as many bits as the multiplier. Although the partial-product register contains more precision than the program is designed to handle, it is essential to maintain the required significant bit for the answer. At the completion of the multiplication algorithm, the 24th bit of the partial-product is used to round off the final result. The result is then normalized to the proper 23 bit floating point format. This manner of handling the partial-product allows maximum precision for the multiplication routine.

```
FPMULT   JSR CKSIGN     Set up and check sign of mantissas
         LDA FOPEXP     Get FPOP exponent
         CLC            Add FPACC exponent
         ADC FPACCE     To FPOP exponent
         STA FPACCE     Save in FPACC exponent
         INC FPACCE     Add one for algorithm compensation
SETMCT   LDA #$17       Set bit counter
         STA CNTR       Store bit counter
MULTIP   LDX #FPMSW     Set pointer to FPACC MS Byte
         LDY #$3        Set precision counter
         JSR ROTATR     Rotate FPACC right
         BCC NADOPP     Carry = zero, don't add partial-product
ADOPP    LDX #MCAND1    Pointer to LS Byte of multiplicand
         STX FMPNT      Store pointer
         LDX #WORK1     Pointer to LS Byte of partial-product
         STX TOPNT      Store pointer
         LDX #$6        Set precision counter
         JSR ADDER      Add multiplicand to partial-product
NADOPP   LDX #WORK6     Set pointer to MS Byte of partial-
                          product
         LDY #$6        Set precision counter
         JSR ROTATR     Rotate partial-product right
         DEC CNTR       Decrement bit counter
         BNE MULTIP     Not zero, continue multiplying
         LDX #WORK6     Else, set pointer to partial-product
         LDY #$6        Set precision counter
         JSR ROTATR     Make room for possible rounding
         LDX WORK3      Set pointer to 24th bit of partial-
```

```
                                product
         LDA PAGE0,X      Fetch LS Byte -1 of result
         ROL A            Rotate 24th bit to sign
         BPL PREXFR       If 24th bit = zero, branch ahead
         CLC              Clear carry for addition
         LDY #$3          Set precision counter
         LDA #$40         Add one to 23rd bit of partial-product
         ADC PAGE0,X      To round off result
         STA WORK3        Store sum in memory
CROUND   LDA #$0          Clear A without changing carry
         ADC PAGE0,X      Add with carry to propagate
         STA PAGE0,X      Store in partial-product
         INX              Increment index pointer
         DEY              Decrement counter
         BNE CROUND       Not zero, add next byte
PREXFR   LDX #FPLSWE      Set pointer to FPACC LSW -1
         STX TOPNT        Store in TOPNT
         LDX #WORK3       Set pointer to partial-product
                            LSW -1
         STX FMPNT        Store in FMPNT
         LDX #$4          Set precision counter
EXMLDV   JSR MOVIND       Move partial-product to FPACC
         JSR FPNORM       Normalize result
         LDA SIGNS        Get sign storage
         BNE MULTEX       If not zero, sign is positive
         LDX #FPLSW       Else, set pointer to FPACC LS Byte
         LDY #$3          Set precision counter
         JSR COMPLM       Complement result
MULTEX   RTS              Exit FPMULT
CKSIGN   LDA #$0          Set page portion of pointers
         STA TOPNT+1      Store in TOPNT
         STA FMPNT+1      Store in FMPNT
         LDA #WORK0       Set pointer to work area
         STA TOPNT        Store in TOPNT
         LDX #$8          Set precision counter
         JSR CLRMEM       Clear work area
         LDA #MCAND0      Set pointer to multiplicand storage
         STA TOPNT        Store in TOPNT
         LDX #$4          Set precision counter
         JSR CLRMEM       Clear multiplicand storage
         LDA #$1          Initialize sign indicator
         STA SIGNS        By storing one in SIGNS
```

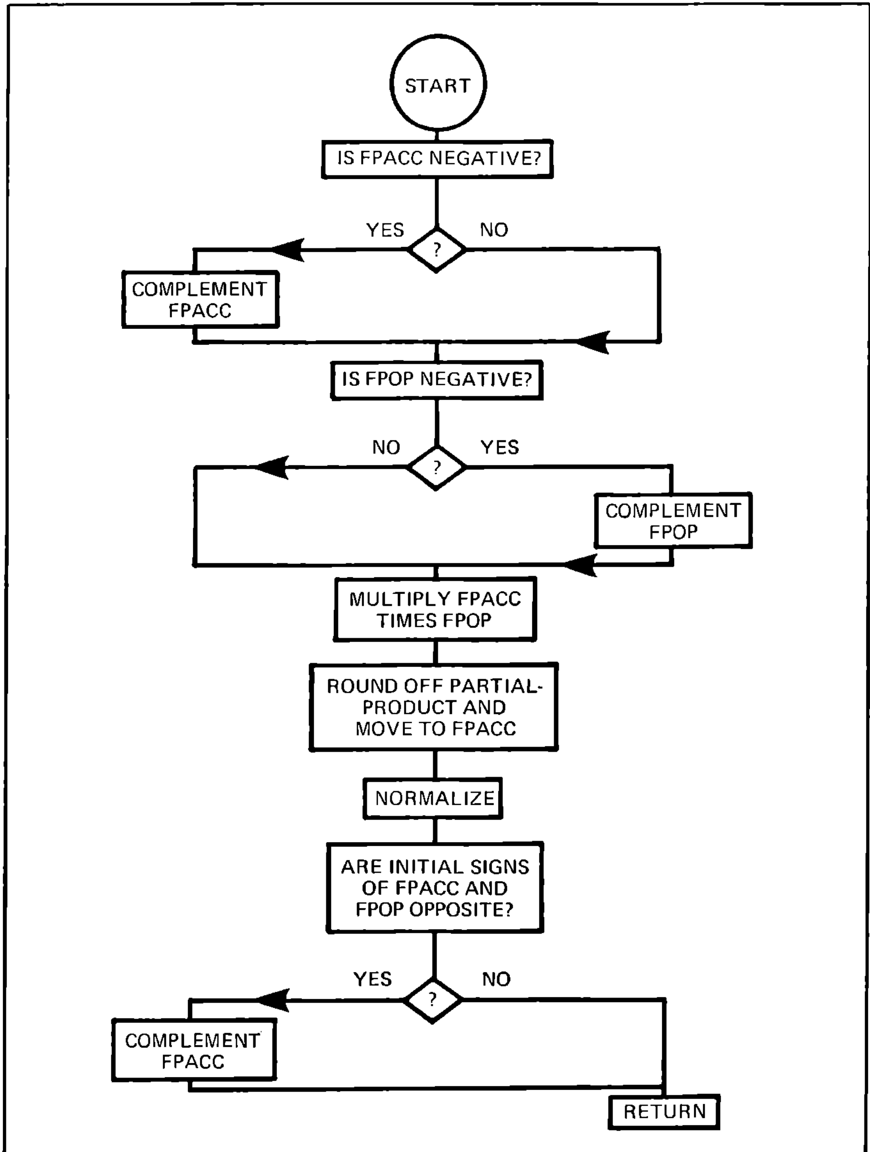
```
          LDA FPMSW     Fetch FPACC MS Byte
          BPL OPSGNT    Positive, check FPOP
NEGFPA    DEC SIGNS     If negative, decrement SIGNS
          LDX #FPLSW    Set pointer to FPACC LS Byte
          LDY #$3       Set precision counter
          JSR COMPLM    Make positive for multiplication
OPSGNT    LDA FOPMSW    Is FPOP negative?
          BMI NEGOP     Yes, complement value
          RTS           Else, return
NEGOP     DEC SIGNS     Decrement SIGNS indicator
          LDX #FOPLSW   Set pointer to FPOP LS Byte
          LDY #$3       Set precision counter
          JMP COMPLM    Complement FPOP and return
```



Floating Point Division

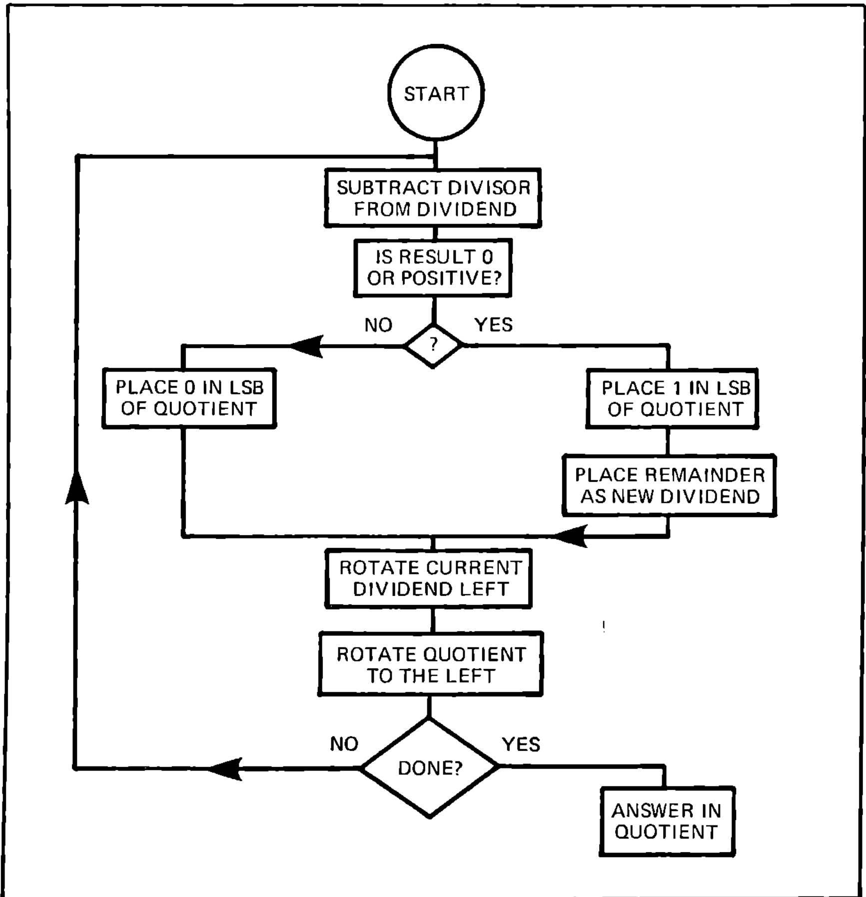
The procedure for division almost can be considered the reverse

of that for multiplication. The division algorithm consists of a series of subtraction and shifting operations. The algorithm is illustrated in the following flow chart and is written for division of numbers in floating point format rather than straight binary. For operating with



numbers in standard binary format, the most significant bits of the divisor and dividend would have to be properly aligned, and the location of the binary point in the quotient would have to be accounted for in cases where the result is not a pure integer.

A sample division of two floating point numbers using this algorithm in a step-by-step fashion is given below. This illustration will divide the binary equivalent of the value 15 (decimal) by 5. The numbers are presented as four-bit values to keep the illustration short. However, in the FPDIV routine, the operation is carried out 23 times for each significant bit of the mantissa of the dividend. Once again, this algorithm assumes the numbers are in normalized floating point format.



0.1111 Original DIVIDEND at start of routine.
0.1010 DIVISOR (Note floating point format.)

———

0.0101 This is the REMAINDER from the subtraction operation. Since the result was POSITIVE, a "1" is placed in the LSB of the QUOTIENT register.

0.0001 QUOTIENT after first loop.

NOW BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

0.1010 New DIVIDEND (which is the previous REMAINDER rotated once to the LEFT).
0.1010 DIVISOR (Does not change during routine).

———

0.0000 RESULT of this subtraction is zero and thus qualifies to become a NEW DIVIDEND. Also, QUOTIENT LSB gets a "1" for this case!

0.0011 QUOTIENT after second loop.

AGAIN BOTH QUOTIENT AND DIVIDEND (NEW REMAINDER) ARE ROTATED LEFT

0.0000 New DIVIDEND (which is the last remainder rotated once to the left).
0.1010 DIVISOR (still same old number).

1.0110 RESULT of this subtraction is a minus number (note that the SIGN bit changed). Thus, old DIVIDEND stays in place and QUOTIENT gets a "0" in LSB!

0.0110 QUOTIENT after third loop.

NOW BOTH QUOTIENT, AND IN THIS CASE, THE OLD DIVIDEND, ARE ROTATED LEFT

0.0000 Old DIVIDEND rotated once to the left.

0.1010	Same old DIVISOR
1.0110	RESULT of this subtraction is again a minus. Old DIVIDEND stays in place. QUOTIENT gets another "0" in LSB.

0.1100 QUOTIENT after fourth loop.

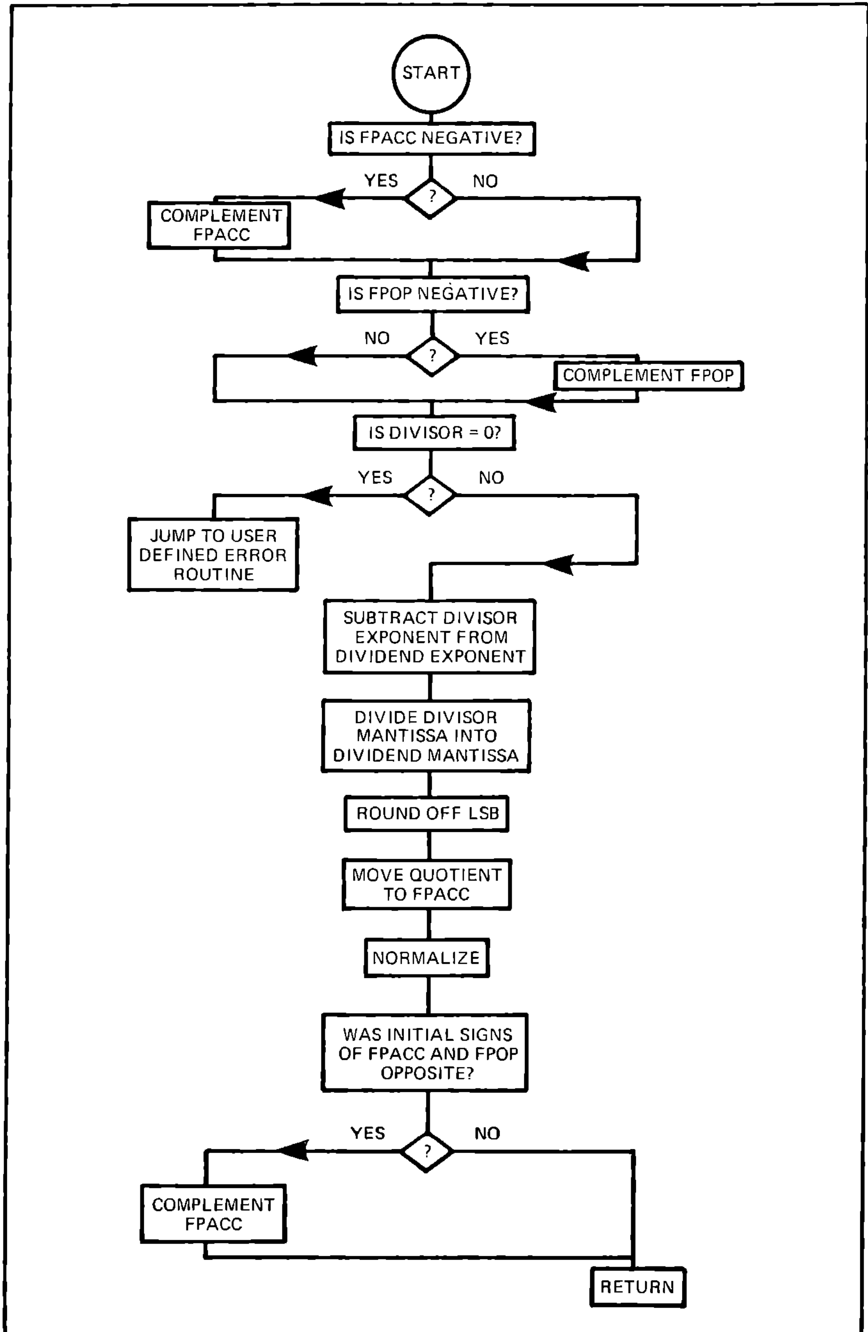
With only four significant bits in the dividend, the calculation illustrated ends after the fourth loop. The answer is contained in the quotient. The exponents are the next quantity that must be dealt with, since the values are represented in floating point notation. Just as in division of decimal floating point numbers, the exponents of the binary counterparts are subtracted; dividend exponent minus the divisor exponent. In the example given, the dividend would have an exponent of four for the normalized binary value of 15 (decimal), and the divisor would have a binary exponent of three. The algorithm as presented requires a compensation factor of +1 after subtracting the exponents in order to have the correct floating point result. Thus, the exponent of the quotient in the previous example would be $(4 - 3) + 1 = 2$. This can be verified by moving the implied binary point in the quotient two places to the right — the binary value of three would indeed be observed.

In the division algorithm, just as in the multiplication, the sign of the dividend and divisor must be positive for the algorithm to operate properly. If either is negative, it must be two's complemented before the division is performed. Also, if the signs are the same, the sign of the quotient must be positive. If the signs are opposite, the quotient must be two's complemented before exiting the routine to make the answer negative.

While examining the FPDIV listing, note that two other conditions are considered by the routine. If the quotient has a remainder after the final loop through the divide algorithm, which would result in a "1" in the 24th bit position, it is rounded off by adding a "1" to the 23rd bit. Also, if a divide by zero is attempted (which is an illegal operation), the FPDIV routine jumps to a routine labeled DERROR. The user may use this to perform whatever is deemed necessary when this error occurs.

```
FPDIV   JSR CKSIGN      Clear work area and set SIGNS
        LDA FPMSW       Check for divide by zero
        BEQ DERROR      Divisor = zero, divide by zero error
```

```
SUBEXP   LDA FOPEXP    Get DIVIDEND exponent
         SEC           Set carry for subtraction
         SBC FPACCE    Subtract DIVISOR exponent
         STA FPACCE    Store result in FPACC exponent
         INC FPACCE    Compensate for divide algorithm
SETDCT   LDA #17       Set bit counter storage
         STA CNTR      To 17 hexadecimal
DIVIDE   JSR SETSUB    Subtract DIVISOR from DIVIDEND
         BMI NOGO      If result is minus, rotate zero in
                         QUOTIENT
         LDX #FOPLSW   Set pointer to DIVIDEND
         STX TOPNT     Store in TOPNT
         LDX #WORK0    Set pointer to QUOTIENT
         STX FMPNT     Store in FMPNT
         LDX #$3       Set precision counter
         JSR MOVIND    Move QUOTIENT to DIVIDEND
                         storage
         SEC           Set carry for positive results
         JMP QUOROT    Rotate into QUOTIENT
DERROR   LDA #$BF      Set ASCII for "?"
         JMP ERROUT    Print "?" and return
NOGO     CLC           Negative result, clear carry
QUOROT   LDX #WORK4    Set pointer to QUOTIENT LS Byte
         LDY #$3       Set precision counter
         JSR ROTL      Rotate carry into LSB of QUOTIENT
         LDX #FOPLSW   Set pointer to DIVIDEND LS Byte
         LDY #$3       Set precision counter
         JSR ROTATL    Rotate DIVIDEND left
         DEC CNTR      Decrement bit counter
         BNE DIVIDE    If not zero, continue
         JSR SETSUB    Do one more for rounding
         BMI DVEXIT    If minus, no rounding
         LDA #$1       If 0 or +, add one to 23rd bit
         CLC           Clear carry for addition
         ADC WORK4     Round off LS Byte of QUOTIENT
         STA WORK4     Restore byte in work area
         LDA #$0       Clear A, not the carry
         ADC WORK5     Add carry to second byte of
                         QUOTIENT
         STA WORK5     Store result
         LDA #$0       Clear A, not the carry
         ADC WORK6     Add carry to MS Byte of
```


START
IS FPACC NEGATIVE?
YES
NO
?
COMPLEMENT FPACC
IS FPOP NEGATIVE?
NO
YES
?
COMPLEMENT FPOP
IS DIVISOR = 0?
YES
NO
?
JUMP TO USER DEFINED ERROR ROUTINE
SUBTRACT DIVISOR EXPONENT FROM DIVIDEND EXPONENT
DIVIDE DIVISOR MANTISSA INTO DIVIDEND MANTISSA
ROUND OFF LSB
MOVE QUOTIENT TO FPACC
NORMALIZE
WAS INITIAL SIGNS OF FPACC AND FPOP OPPOSITE?
YES
NO
?
COMPLEMENT FPACC
RETURN

```
                          QUOTIENT
          STA WORK6       Store result
          BPL DVEXIT      If MSB = 0, exit
          LDX #WORK6      Else prepare to rotate right
          LDY #$3         Set precision counter
          JSR ROTATR      Clear sign bit counter
          INC FPACCE      Compensate exponent for rotate
DVEXIT    LDX #FPLSWE     Set pointer to FPACC
          STX TOPNT       Store in TOPNT
          LDX #WORK3      Set pointer to QUOTIENT
          STX FMPNT       Store in FMPNT
          LDX #$4         Set precision counter
          JMP EXMLDV      Move QUOTIENT to FPACC
SETSUB    LDX #WORK0      Set pointer to work area
          STX TOPNT       Store in TOPNT
          LDX #FPLSW      Set pointer to FPACC
          STX FMPNT       Store in FMPNT
          LDX #$3         Set precision counter
          JSR MOVIND      Move FPACC to work area
          LDX #WORK0      Prepare for subtraction
          STX TOPNT       Store pointer to DIVISOR
          LDX #FOPLSW     Set pointer to FPOP LS Byte −1
          STX FMPNT       Store pointer to DIVIDEND
          LDY #$0         Initialize index pointer
          LDX #3          Set precision counter
          SEC             Set carry for subtraction
SUBR1     LDA (FMPNT),Y   Fetch FPOP byte (DIVIDEND)
          SBC (TOPNT),Y   Subtract FPACC byte (DIVISOR)
          STA (TOPNT),Y   Store in place of DIVISOR
          INY             Advance index pointer
          DEX             Decrement precision counter
          BNE SUBR1       Not zero, continue subtraction
          LDA WORK2       Set sign bit result in N flag
          RTS             Return with flag conditioned
```

The floating point routines presented to this point, when assembled into the object code, will reside in approximately two and one half pages of memory. Additional memory is required for the data areas on page 00 which are used to store various counters and data values. The locations used on page 00 by these floating point routines are listed in the following table. The addresses listed here are the same as those used by the floating point package presented in

Address	Program Label	Definition
0000	FMPNT	FROM pointer
0002	TOPNT	TO pointer
0004	CNTR	Counter Storage
0005	TSIGN	Sign Indicator
0006	SIGNS	Signs Indicator (Multiply and Divide)
0007	FPLSWE	FPACC Extension
0008	FPLSW	FPACC Least Significant Byte
0009	FPNSW	FPACC Next Significant Byte
000A	FPMSW	FPACC Most Significant Byte
000B	FPACCE	FPACC Exponent
000C	MCAND0	Multiplication Work Area
000D	MCAND1	Multiplication Work Area
000E	MCAND2	Multiplication Work Area
000F	FOLSWE	FPOP Extension
0010	FOPLSW	FPOP Least Significant Byte
0011	FOPNSW	FPOP Next Significant Byte
0012	FOPMSW	FPOP Most Significant Byte
0013	FOPEXP	FPOP Exponent
0014	WORK0	Work Area
0015	WORK1	Work Area
0016	WORK2	Work Area
0017	WORK3	Work Area
0018	WORK4	Work Area
0019	WORK5	Work Area
001A	WORK6	Work Area
001B	WORK7	Work Area

The floating point routines are extremely powerful routines that can be of considerable value to someone who requires such mathematical calculations on a 6502-based microcomputer. These routines provide the capability to handle binary numbers equivalent to six or seven significant decimal digits raised to plus or minus the 38th power of ten. Using these routines as a base, a wide variety of mathematic operations can be performed by loading FPACC and FPOP with the numbers in normalized floating point format and calling the proper routine.

One of the most common requirements of a program that

deals with binary numbers is the conversion to and from decimal because it is often necessary to communicate with a human operator. Therefore, to illustrate a method for converting from floating point decimal to floating point binary, and back, the following three routines are included.

Floating Point Input Routine

The first of these routines performs the conversion of decimal floating point numbers to floating point binary. The overall requirement of this routine is to receive the decimal number in floating point format, normalize the mantissa portion to an all-integer value, and convert to the equivalent floating point binary value.

Floating point decimal values may be expressed in various forms, as indicated below.

123.45
or
1.2345 E+2

As either of these formats are received, the mantissa portion is converted to binary. The exponent is also formulated during the input to provide the proper normalized decimal value. Unlike the binary normalization, which shifts the binary point to the left of the MSB, decimal normalization maintains the decimal point to the right of the least significant digit. This provides a purely integer mantissa. Thus, the example above would be normalized to

12345 E–2

The conversion of the decimal mantissa to binary is accomplished by the routine labeled DECBIN, which is a version of the TIMS10 subroutines presented in Chapter Four. This subroutine converts each digit entered. First, it multiplies the binary equivalent of the digits already received by ten. The BCD value of the latest digit input is added to create the new binary number.

Once the mantissa is converted, the decimal exponent is input and converted to binary. At this point, it is necessary to normalize the mantissa of the binary equivalent be calling the FPNORM routine. The FPACC exponent is set to a value of 23 before calling the FPNORM routine. Then, using the FPMULT routine, the normalized binary equivalent is multiplied by ten (for each unit of a positive decimal exponent received), or by 0.1 (for each unit of a negative

exponent received).

The input and output portions of this routine require that the user provide driver routines for the specific input and output devices associated with one's system. The requirement for the INPUT routine is to return to the calling routine with the ASCII code for the character entered in the accumulator. The routine to output characters to a display device, such as a mechanical printer or video display, must accept the character to be output as an ASCII character stored in the accumulator. This output routine labeled ECHO, is called to echo the characters received from the input device back to the display device. Refer to Chapter Seven for methods of creating these routines. The ECHO routine should return with the ASCII code for the character output in the accumulator.

Presented next is the decimal to binary input routine listing. Both formats illustrated previously are allowed as legal entries. The routine accounts for positive and negative mantissas and exponents. The operator has the option to cancel the current input by entering a control zero character. Several locations on page 00 are used to store the input characters and save counters and indicators. These locations will be summarized later in this chapter.

```
FPINP    LDA #$00      Clear page portions of TOPNT
         STA TOPNT     And FMPNT to set up pointers
         STA FMPNT     To page zero, where data is stored
         CLD           Clear decimal mode flag
         LDX #INMTAS   Set pointer to storage area
         STX TOPNT     Store in TOPNT
         LDX #$0C      Set precision counter
         JSR CLRMEM    Clear storage area
         JSR INPUT     Get character from kybd
         CMP #$AB      Test if + sign
         BEQ SECHO     Yes, echo and continue
         CMP #$AD      Test if – sign
         BNE NOTPLM    No, test if valid character
         STA INMTAS    Make input sign nonzero
SECHO    JSR ECHO      Echo character to output
NINPUT   JSR INPUT     Get character from kybd
NOTPLM   CMP #$8F      Test for control zero
         BNE SERASE    No, skip erase
ERASE    LDA #$BC      Yes, print < as a rubout
         JSR ECHO      Output <
         JSR SPACES    Print several spaces
```

```
         JMP FPINP       Restart input string
SERASE   CMP #$AE        Test for decimal point
         BNE SPRIOD      No, skip period
PERIOD   BIT INPRDI      Decimal point already received?
         BPL PER1        No dec. pt. yet, continue
         BMI ISLAND      Yes, end input
PER1     STA INPRDI      Set dec. pt. indicator
         LDY #$0
         STY CNTR        Reset digit counter
         JSR ECHO        Echo dec. pt. to output
         JMP NINPUT      Get next character
SPRIOD   CMP #$C5        Test for E for exponent
         BNE SFNDXP      No, skip exponent
FNDEXP   JSR ECHO        Yes, echo E — to output
         JSR INPUT       Input next character of exponent
         CMP #$AB        Test for + sign
         BEQ EXECHO      Yes, echo it
         CMP #$AD        Test for — sign
         BNE NOEXPS      No, test for digit
         STA INEXPS      Yes, store minus indicator
EXECHO   JSR ECHO        Echo to output
EXPINP   JSR INPUT       Get next character for exponent
NOEXPS   CMP #$8F        Test for control zero
         BEQ ERASE       Yes, start again
         CMP #$B0        Number, test low limit
ISLAND   BMI ENDINP      No, end input string
         CMP #$BA        Test upper limit
         BPL ENDINP      No, end input string
         AND #$0F        Mask and strip ASCII
         STA TEMP1       Store BCD in temporary storage
         LDX #IOEXPD     Set pointer to exponent storage
         LDA #$03        Test for upper limit of exponent
         CMP $0,X        Is ten's digit > 3?
         BMI ENDINP      Yes, end input
         LDA $0,X        Store temporarily in A
         CLC             Clear carry
         ROL $0,X        Exponent X 2
         ROL $0,X        Exponent X 4
         ADC $0,X        Add original (X 5)
         ROL A           Exponent X 10
         ADC TEMP1       Add new input
         STA $0,X        Store in exponent storage
```

```
         LDA #$B0       Restore ASCII code
         ORA TEMP1      By setting $B0
         BNE EXECHO     Echo number
SFNDXP   CMP #$B0       Test for valid number
         BMI ENDINP     Too low, end input
         CMP #$BA       Test for upper limit
         BPL ENDINP     If not valid, end input
         TAY            Save temporarily
         LDA #$F8       Input too large?
         BIT IOSTR2     Test for too large
         BNE NINPUT     Yes, ignore present input
         TYA            No, fetch digit again
         JSR ECHO       Echo to output
         INC CNTR       Increment digit counter
         AND #$0F       Mask off ASCII
         PHA            Save BCD digit temporarily
         JSR DECBIN     Multiply previous value X 10
         LDX #IOSTR     Set pointer to storage
         PLA            Fetch digit just entered
         CLC            Clear carry for addition
         ADC $0,X       Add digit to storage
         STA $0,X       Save new total
         LDA #$0        Clear A for next addition
         ADC $1,X       Add carry to next byte
         STA $1,X       Save new total
         LDA #$0        Clear A again for addition
         ADC $2,X       Add carry to final byte
         STA $2,X       Save final byte of total
         JMP NINPUT     Look for next character input
ENDINP   LDA INMTAS     Test is positive or negative
         BEQ FINPUT     Indicator zero, number positive
         LDX #IOSTR     Index to LSB of input mantissa
         LDY #$03       Set precision counter
         JSR COMPLM     Two's complement for negative
FINPUT   LDA #$0
         STA IOSTR-$1   Clear input storage LSB-1
         LDA #FPLSWE    Set TOPNT to FPACC
         STA TOPNT
         LDA #IOSTR-$1  Set FMPNT to input storage
         STA FMPNT      Set byte counter
         LDX #$04       Move input to FPACC
         JSR MOVIND     Set exponent for FPNORM
```

```
         LDY #$17       Store exponent for normalization
         STY FPACCE     Normalize the input
         JSR FPNORM     Test exponent sign indicator
         LDA INEXPS     Positive? Same exponent
         BEQ POSEXP     Minus, form two's complement
         LDA #$FF       Of exponent value
         EOR IOEXPD     By complementing and incrementing
         STA IOEXPD
         INC IOEXPD     Test period indicator
POSEXP   LDA INPRDI
         BEQ EXPOK      If zero, no decimal point
         LDA #$0        Clear A
         SEC            Set carry for subtraction
         SBC CNTR       Form negative of count
EXPOK    CLC            Clear carry for addition
         ADC IOEXPD     Add to compensate for dec. pt.
         STA IOEXPD     Store results
         BMI MINEXP     Negative exponent, adjust to zero
         BNE EXPFIX     Not zero, adjust to zero
         RTS            Return with value in FPACC
EXPFIX   JSR FPIX10     Multiply by ten
         BNE EXPFIX     Exponent not zero, multiply again
         RTS            Return
FPX10    LDA #$04       Multiply FPACC X 10
         STA FOPEXP     Load FPOP with a value of ten
         LDA #$50       By setting the exponent to four
         STA FOPMSW     And the mantissa to $50,$00,$00
         LDA #$00
         STA FOPNSW
         STA FOPLSW
         JSR FPMULT     Multiply FPACC X FPOP
         DEC IOEXPD     Decrement decimal exponent
         RTS            Return to test for completion
MINEXP   JSR FPD10      Compensated decimal exponent minus
         BNE MINEXP     FPACC X 0.1 till decimal exponent = zero
         RTS            Return
FPD10    LDA #$FD       Place 0.1 in FPOP by
         STA FOPEXP     Setting FPOP exponent to −3
         LDA #$66       And loading mantissa with $66,$66,
                         $67
         STA FOPMSW
         STA FOPNSW
```

```
        LDA #$67
        STA FOPLSW
        JSR FPMULT     Multiply FPACC X FPOP
        INC IOEXPD     Increment decimal exponent
        RTS            Return
DECBIN  LDA #$00
        STA IOSTR3     Clear MS Byte + 1 of result
        LDX #IOLSW     Set pointer to I/O work area
        STX TOPNT      Store in TOPNT
        LDX #IOSTR     Set pointer to I/O storage
        STX FMPNT      Store in FMPNT
        LDX #$04       Set precision counter
        JSR MOVIND     Move I/O storage to work area
        LDX #IOSTR     Set pointer to original value
        LDY #$04       Set precision counter
        JSR ROTATL     Start X 10 routine (total =X2)
        LDX #IOSTR     Reset pointer
        LDY #$04       Set precision counter
        JSR ROTATL     Multiply by two again (total =X4)
        LDX #IOLSW     Set pointer to I/O work area
        STX FMPNT      Store in FMPNT
        LDX #IOSTR     Set pointer to I/O storage
        STX TOPNT      Store in TOPNT
        LDX #$04       Set precision counter
        JSR ADDER      Add original to rotated (total =X5)
        LDX #IOSTR     Reset pointer
        LDY #$04       Set precision counter
        JMP ROTATL     X2 again (total =X10) and return
```

Floating Point Output Routine

The next routine converts the floating point binary number in the FPACC to its floating point decimal equivalent, and output it to the display device as ASCII characters in the following format:

0.1234567 E+07

First, the normalized value is converted to a binary value in which the binary exponent is within the range of —4 to —1. As this is done, the decimal exponent is generated. Once the binary exponent is properly adjusted, the decimal mantissa is output by multiplying the adjusted binary mantissa by ten for each decimal digit. Each multiplication causes the next decimal digit to be pushed out

into the most significant byte +1 of the binary mantissa. As each digit is pushed out, its ASCII code is formed, and the ECHO routine is called to output the digit. When the mantissa has been output, the decimal exponent is converted. This conversion makes use of the method described in Chapter Four for binary to decimal conversion. The exponent is then output.

```
FPOUT    LDA #$0
         STA IOEXPD     Clear decimal exponent storage
         LDA FPMSW      Is value to be output negative?
         BMI OUTNEG     Yes, make positive and output "–"
         LDA #$AB       Else, set ASCII code for "+"
         BNE AHEAD1     Go display + sign
OUTNEG   LDX #FPLSW     Set pointer to LS Byte of FPACC
         LDY #$3        Set precision counter
         JSR COMPLM     Make FPACC positive
         LDA #$AD       Set ASCII code for "–"
AHEAD1   JSR ECHO       Output sign of result
         LDA #$B0       Set up ASCII zero
         JSR ECHO       Output zero to display
         LDA #$AE       Set up ASCII decimal point
         JSR ECHO       Output decimal point
         DEC FPACCE     Decrement FPACC exponent
DECEXT   BPL DECEXD     If compensated, exponent >= 0
         LDA #$4        Exponent negative, add four to FPACCE
         CLC            Clear carry for addition
         ADC FPACCE     Add four to FPACC exponent
         BPL DECOUT     If exponent >= 0, output mantissa
         JSR FPX10      Else, multiply mantissa by ten
DECREP   LDA FPACCE     Get exponent
         JMP DECEXT     Repeat test for >= 0
DECEXD   JSR FPD10      Multiply FPACC by 0.1
         JMP DECREP     Check status of FPACC exponent
DECOUT   LDX #IOSTR     Set up for move operation
         STX TOPNT      Set TOPNT to working register
         LDX #FPLSW     Set pointer to FPACC LS Byte
         STX FMPNT      Store in FMPNT
         LDX #$3        Set precision counter
         JSR MOVIND     Move FPACC to output registers
         LDA #$0
         STA IOSTR3     Clear output register MS Byte +1
         LDX #IOSTR     Set pointer to output LS Byte
```

```
          LDY #$3        Set precision counter
          JSR ROTATL     Rotate to compensate for sign bit
          JSR DECBIN     Output register X 10, overflow in MS
                           Byte +1
COMPEN    INC FPACCE     Increment FPACC exponent
          BEQ OUTDIG     Output digit when compensation done
          LDX #IOSTR3    Else, rotate right to compensate
          LDY #$4        For any remainder in binary exponent
          JSR ROTATR     Perform rotate right operation
          JMP COMPEN     Repeat loop until exponent = zero
OUTDIG    LDA #$7        Set digit counter to seven
          STA CNTR       For output operation
          LDA IOSTR3     Fetch BCD, see if first digit = zero
          BEQ ZERODG     Yes, check remainder of digits
OUTDGS    LDA IOSTR3     Get BCD from output register
          ORA #$B0       Form ASCII code for numbers
          JSR ECHO       And output digit
DECRDG    DEC CNTR       Decrement digit counter
          BEQ EXPOUT     = zero, done output exponent
          JSR DECBIN     Else, get next digit
          JMP OUTDGS     Form ASCII and output
ZERODG    DEC IOEXPD     Decrement exponent for skipping
                           display
          LDA IOSTR2     Check if mantissa = zero
          BNE DECRDG     If not zero, continue output
          LDA IOSTR1
          BNE DECRDG
          LDA IOSTR
          BNE DECRDG
          LDA #$0        Mantissa zero, clear exponent
          STA IOEXPD
          BEQ DECRDG     Before finishing display
EXPOUT    LDA #$C5       Set up ASCII code for E
          JSR ECHO       Display E for exponent
          LDA IOEXPD     Test if negative
          BMI EXOUTN     Yes, display "–" and negate
          LDA #$AB       No, set ASCII code for "+"
          JMP AHEAD2     Display exponent value
EXOUTN    EOR #$FF       Two's complement exponent
          STA IOEXPD     To make negative value positive
          INC IOEXPD     For output of exponent value
          LDA #$AD       Set ASCII code for "–"
```

```
AHEAD2   JSR ECHO      Output sign of exponent
         LDY #$0       Clear ten's counter
         LDA IOEXPD    Fetch exponent
SUB12    SEC           Set carry for subtraction
         SBC #$0A      Subtract ten's from exponent
         BMI TOMUCH    If minus, ready for output
         STA IOEXPD    Restore positive result
         INY           Advance ten's counter
         JMP SUB12     Continue subtraction
TOMUCH   TYA           Put MS digit into A
         ORA #$B0      Form ASCII code
         JSR ECHO      Output ten's digit to display
         LDA IOEXPD    Fetch unit's digit
         ORA #$B0      Form ASCII code
         JMP ECHO      Output digit and return
```

Putting the Pieces Together

This final routine ties the FPINP and FPOUT routines together, along with the floating point mathematical routines FPNORM, FPADD, FPSUB, FPMULT and FPDIV to create a floating point calculator program. All that is required by the reader is to supply the I/O driver routines. The program allows one to enter and receive data in the following format:

27.6E–2 X–5 = –0.1380000E+01

```
FPCONT   LDA #$8D      ASCII carriage return
         JSR ECHO      Output carriage return
         LDA #$8A      ASCII line feed
         JSR ECHO      Output line feed
         JSR FPINP     Get first FP decimal number
         JSR SPACES    Output two spaces
         LDX #TPLSW    Set pointer to temporary storage
         STX TOPNT     Store in TOPNT
         LDX #FPLSW    Set pointer to FPACC LS Byte
         STX FMPNT     Store in FMPNT
         LDX #$04      Set precision counter
         JSR MOVIND    Move FPACC to temporary storage
NVALID   JSR INPUT     Fetch operator from input
         CMP #$AB      Test for "+" sign
         BNE NOTADD    No, try "–"
```

```
           JSR OPERAT      Input FPACC value
           JSR FPADD       Add FPOP to FPACC
           JMP FINAL       Output result of addition
NOTADD     CMP #$AD        Test for "–" sign
           BNE NOTSUB      No, try "X"
           JSR OPERAT      Yes, input FPACC value
           JSR FPSUB       Subtract FPACC from FPOP
           JMP FINAL       Output result of subtraction
NOTSUB     CMP #$D8        Test for "X" sign
           BNE NOTMUL      No, try "/"
           JSR OPERAT      Yes, input FPACC value
           JSR FPMULT      Multiply FPOP times FPACC
           JMP FINAL       Output result of multiplication
NOTMUL     CMP #$AF        Test for "/" sign
           BNE NOTDIV      No, try delete
           JSR OPERAT      Yes, input FPACC value
           JSR FPDIV       Divide FPOP by FPACC
FINAL      JSR FPOUT       Output answer
           JMP FPCONT      Set up for new input
NOTDIV     CMP #$8F        Not operator, try control zero
           BNE NVALID      No, ignore, try again
           BEQ FPCONT      Yes, restart input string
OPERAT     JSR ECHO        Display control operator
           JSR SPACES      Display a few spaces
           JSR FPINP       Fetch second FP decimal number
           JSR SPACES      Display two spaces
           LDA #$BD        Set ASCII code for "="
           JSR ECHO        Display "=" sign
           JSR SPACES      Display two spaces
           LDX #FOPLSW     Set pointer to FPOP LS Byte
           STX TOPNT       Store in TOPNT
           LDX #TPLSW      Set pointer to temporary storage
           STX FMPNT       Store in FMPNT
           LDX #$04        Set precision counter
           JMP MOVIND      Move first input to FPOP and return
```

The three routines, FPINP, FPOUT and FPCONT, as presented, require less than three pages of memory. This requirement may be shortened to some extent by forming subroutines for various common instruction sequences. This has not been done here to maintain clarity of operation. However, the ambitious reader should have little difficulty in shortening the program. The following list defines

the data areas on page zero used by these routines. The addresses listed here are used by the floating point program presented in Appendix F.

Address	Program Label	Definition
001C	INMTAS	I/O Mantissa Sign
001D	INEXPS	I/O Exponent Sign
001E	INPRDI	I/O Period Indicator
001F	IOLSW	I/O Work Area Least Significant Byte
0020	IONSW	I/O Work Area Next Significant Byte
0021	IOMSW	I/O Work Area Most Significant Byte
0022	IOEXP	I/O Work Area Exponent
0023	IOSTR	I/O Storage
0024	IOSTR1	I/O Storage
0025	IOSTR2	I/O Storage
0026	IOSTR3	I/O Storage
0027	IOEXPD	I/O Exponent Storage
0028	TPLSW	Temporary Input Storage Least Significant Byte
0029	TPNSW	Temporary Input Storage Next Significant Byte
002A	TPMSW	Temporary Input Storage Most Significant Byte
002B	TPEXP	Temporary Input Storage Exponent
002C	TEMP1	Temporary Storage to Reside on Pages

This floating point program has been assembled to reside on pages 02 to 07 and is presented in Appendix F as a memory dump. The locations on page zero used to store the temporary data are the same as those called out in the test. The order in which the routines have been presented for explanation is the same order in which they are assembled in Appendix F. A complete symbol table is provided following the memory dump.

Chapter 6

Decimal Arithmetic Routines

When using a computer to process mathematical data, such as data entered by an operator, and after processing, output for the operator to read, the decimal numbering system is most often the base used. This representation allows the operator to enter and read the data in a form most widely accepted and easily understood, since it is usually drummed into everyone from the time they are born. The computer, on the other hand, is generally designed to operate most efficiently with numbers in binary format. Therefore, there must be some means made available to allow the operator and the computer to communicate in a common number system.

Conversion routines from one number base to another are often used. Routines, such as those presented, make it possible to input and output numbers in decimal notation while performing the actual calculations in binary notation. However, inaccuracies can creep into the most elementary calculation as a result of the conversion! For example, the subtraction of 2.1 from 5.0 may be output at 2.8999 rather than 2.9 because of conversion errors.

For applications where the operation required can be performed as decimal addition and subtraction, it would be far more accurate to perform these simple mathematical calculations in the same format as that used for input and output. The 6502 provides for this operation with the decimal mode flag. The decimal mode flag selects between binary and decimal arithmetic. When set, the addition and subtraction instructions assume BCD digits are contained in the two subject bytes. With the decimal mode flag reset, these instructions assume the affected bytes will contain binary data.

Presented here are routines that perform addition, subtraction, multiplication, and division of decimal numbers. The format used

to represent each number will be the same for all routines. Four bits are required to define each BCD digit. Therefore, two digits will be stored in a single eight-bit byte, with the least significant digit of the pair in the least significant half of the byte. These operations work with multiple precision values, allowing up to 256 bytes to be assigned for each number. For the routines presented, the bytes used to represent each number must be stored in a table of sequential memory locations, with the byte containing the least significant digit pair in the lowest address of the table.

The Basic Subroutines

First is the decimal addition routine. If it looks like it is almost a carbon copy of the ADDER routine in Chapter Three, that's because it is! The only difference is that the decimal mode flag is set when the subroutine is executed. The SED instruction has been added to this routine to guarantee setting the flag. However, this may be deleted if the calling program has already set it. FMPNT and TOPNT must be initialized to the least significant byte of their respective values. Index register X must be set to the binary count of the number of bytes per value. The result is stored in TOPNT.

```
DECADD   LDY #00          Initialize pointer
         SED              Set decimal mode flag
         CLC              Clear carry flag
DCADD1   LDA (TOPNT),Y    Fetch byte from one value
         ADC (FMPNT),Y    Add byte of second value
         STA (TOPNT),Y    Store sum
         INY              Increment index pointer
         DEX              Decrement byte counter
         BNE DCADD1       Not zero, continue addition
         RTS              Return
```

The decimal subtraction routine is also the same as the subroutine SUBBER in Chapter Three. However, the decimal mode flag is set at the start of this routine. Just as in the previous routine, TOPNT and FMPNT initially must be set to the least significant byte of the minuend and subtrahend, respectively. The X index register must be set to the binary number of bytes in each value. This routine stores the result in place of the value indicated by TOPNT. For a valid answer, the minuend must be greater than or equal to the subtrahend.

```
DECSUB   LDY #00          Initialize index pointer
         SED              Set the decimal mode flag
         SEC              Set the carry flag
DCSUB1   LDA (TOPNT),Y    Fetch a byte from the minuend
         SBC (FMPNT),Y    Subtract a byte of the subtrahend
         STA (TOPNT),Y    Store the difference in the minuend
         INY              Increment index pointer
         DEX              Decrement byte count
         BNE DCSUB1       Not zero, continue
         RTS              Return
```

Calculating with Signed BCD

The next pair of routines uses the decimal addition and subtraction routines to perform the actual computation. These routines add the capability to perform addition and subtraction of signed decimal numbers. The sign and magnitude of the numbers to be added or subtracted must be checked to determine whether the operation actually calls for an addition or subtraction, and to set up the proper sign for the result of the operation.

The two numbers to be operated on by these routines must be stored in two tables, referred to by the labels DCAC and DCOP. DCAC is the decimal accumulator, which is used to store one addend for the signed addition routine, and the minuend for the signed subtraction routine. DCOP is the decimal operand table, and must contain the other addend for the signed addition routine, and the minuend for the signed subtraction routine. For both routines, the results of the respective operations are stored in DCAC upon returning to the calling program. Also, the initial contents of DCOP are not necessarily maintained.

The number of bytes in each table can be varied to allow for the number of digits desired per value. For these routines, the tables must be of equal length. The tables used by these routines are three bytes long, allowing six BCD digits per number. If the length of the tables is changed, the constant 03 in the instructions whose comments are marked by a double asterisk must be changed to indicate the new byte count.

Unlike binary numbers in which the MSB of the binary value may be considered as the sign bit, BCD representation does not allow for this convenient method of sign designation. One may sacrifice a BCD digit by assigning the MSB of the MS Byte of a value as the sign bit. However, this method does not simplify the procedure for checking the sign of the value. It also complicates the process of

checking for an overflow or underflow, since the C flag will not automatically indicate these errors. A separate memory location will be used to indicate the sign of the decimal values.

The sign of each number is set up in separate memory locations and uses the most significant bit of each byte to indicate a negative number if set, or a positive number if reset. The remaining bits in each sign byte must be all zeros, since there are several locations in the routine in which the sign bytes are checked as being equal. This check involves the contents of the entire byte, not just the MSB. Making the remaining bits equal to zero is consistent with the format used by these routines to set and reset the sign bit of the result. The sign bytes that refer to the sign of the DCAC and DCOP are labeled SIGNAC and SIGNOP, respectively.

Signed Addition

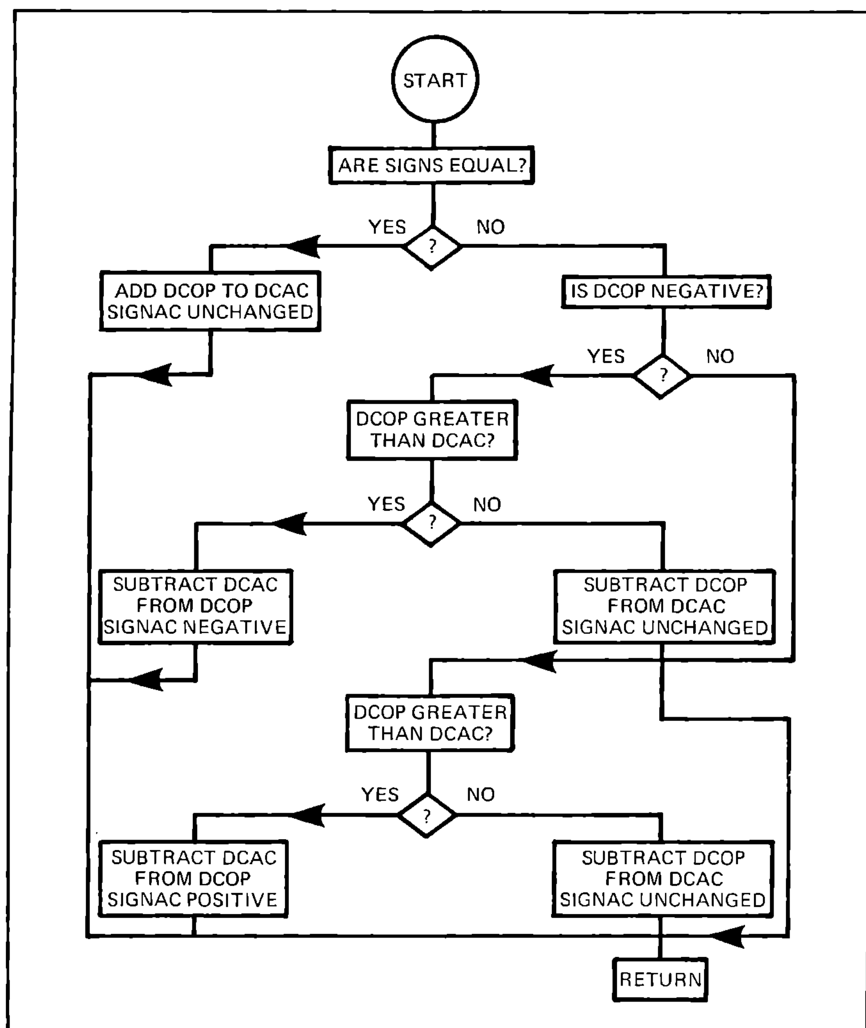
Depending on the sign and magnitude of the values operated on, it may be necessary to exchange the contents of DCAC and DCOP. This is required when the indicated operation is that of subtracting the accumulator from the operand. This exchange is accomplished by a subroutine labeled SHIFT. SHIFT exchanges the contents of the accumulator and operand one byte at a time.

In the process of determining which operation is actually called for (addition or subtraction), the relative magnitudes of the two numbers must be known. This is determined by the CMPR subroutine. Its operation is basically the same as that of the CPRMEM subroutine in Chapter Three. The only difference is that this routine is written specifically for comparing two triple precision values.

The signed addition routine, beginning at the label SGNADD, adds the contents of DCOP to DCAC, and returns with the answer in DCAC. The calling routine simply loads DCAC, SIGNAC, DCOP, and SIGNOP with the desired values before calling this routine. When the sign of each is the same, the addition is performed as indicated. If the signs are different, the value of smaller magnitude is subtracted from the larger value, and the sign of the larger is set as the sign of the answer. The actual computation is done by one of the previous addition or subtraction subroutines. The condition of the carry flag upon returning to the calling program will indicate whether an overflow or underflow has occurred as a result of the operation, signalling a possible error condition. The operation of the signed addition routine is illustrated in the flow chart following the source listing.

```
SIGNOP   *=*+1          Sign byte of DCOP
SIGNAC   *=*+1          Sign byte of DCAC
DCOP     *=*+2          Decimal operand storage
DCOPM    *=*+1          Decimal operand MS Byte
DCAC     *=*+2          Decimal accumulator storage
DCACM    *=*+1          Decimal accumulator MS Byte
SGNADD   LDA SIGNOP     Fetch sign of DCOP
         CMP SIGNAC     Compare to sign of DCAC
         BEQ SAR2       Signs equal, add numbers
         BCC SAR3       SIGNOP negative, SIGNAC positive
SAR1     JSR CMPR       Is DCOP greater than DCAC?
         BCS SB12       No, subtract DCOP from DCAC
         LDA #00        Yes, set up zero byte
         STA SIGNAC     Clear sign of DCAC
SB21     JSR SHIFT      Exchange DCAC and DCOP
SB12     LDA #<DCAC     Fetch low portion of DCAC address
         STA TOPNT      Store in TOPNT
         LDA #<DCOP     Fetch low portion of DCOP address
         STA FMPNT      Store in FMPNT
         LDX #$03       **Set precision counter
         JMP DECSUB     Subtract and return
SAR2     LDA #<DCOP     Set pointer for addition
         STA FMPNT      Of DCOP to DCAC
         LDA #<DCAC
         STA TOPNT
         LDX #$03       **Set precision counter
         JMP DECADD     Add and return
SAR3     JSR CMPR       Is DCOP greater than DCAC?
         BCS SB12       No, subtract DCOP from DCAC
         BEQ SB21       Equal, SIGNAC remains positive
         LDA #$80       Yes, change SIGNAC
         STA SIGNAC     To negative value
         BNE SB21       Subtract DCAC from DCOP
SHIFT    LDX #$00       Initialize index pointer
SHIFTA   LDA DCOP,X     Fetch byte from DCOP
         LDY DCAC,X     Fetch byte from DCAC
         STA DCAC,X     Store DCOP byte in DCAC
         STY DCOP,X     Store DCAC byte in DCOP
         INX            Advance index register
         CPX #03        Last pair of bytes swapped?
         BNE SHIFTA     No, swap next pair
         RTS            Yes, return
```

```
CMPR     LDX #$03        Initialize index pointer
CMPRA    LDA DCOP−1,X    Fetch byte from DCOP
         CMP DCAC−1,X    Compare to byte of DCAC
         BNE CMPRET      Not equal, return
         DEX             Equal, decrement index pointer
         BNE CMPRA       Not done, continue
         RTS             Return, with C and Z conditioned
```

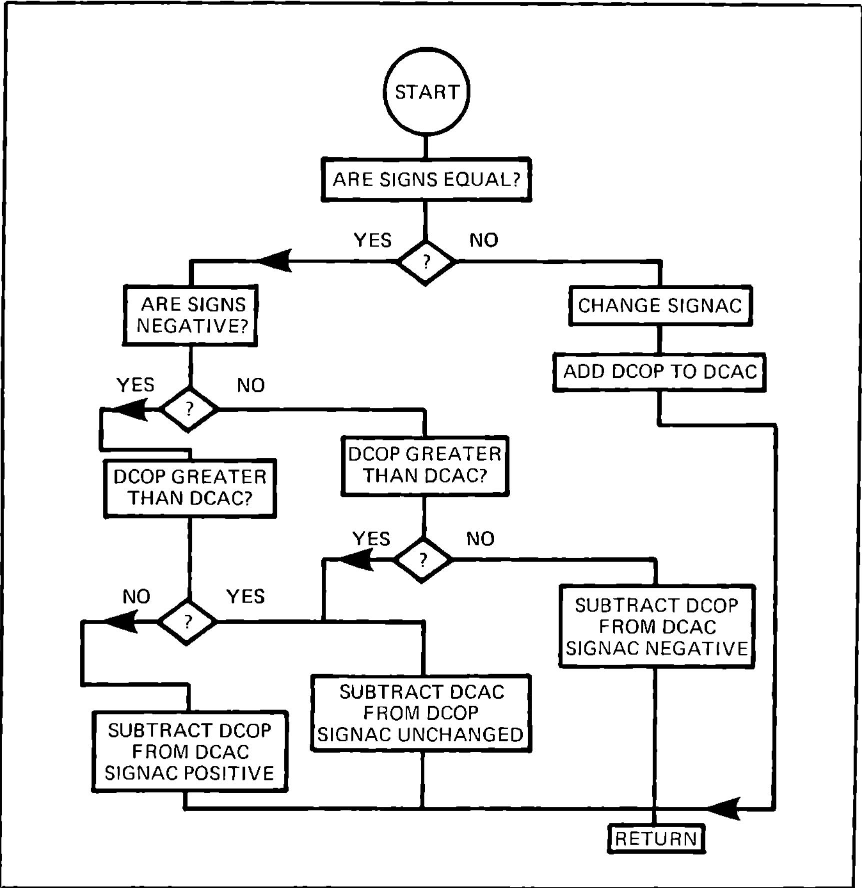


Signed Subtraction

The signed subtraction routine, starting at the label SGNSUB, subtracts the contents of DCOP from the contents of DCAC. The calling program must set the contents of DCAC, SIGNAC, DCOP and SIGNOP with the desired values before calling this routine. The sign and magnitude of each of the numbers is examined to determine the actual operation to be performed. Several of the routines in the signed addition routine are used here. Since the decimal addition or subtraction routine is the last operation to be executed, the condition of the C flag will indicate whether an error has occurred.

```
SGNSUB    LDA SIGNOP      Fetch sign of DCOP
          CMP SIGNAC      Compare to sign of DCAC
          BNE DIFSGN      Not equal, change sign and add
          AND #$80        Are both negative?
          BMI NAGATV      Yes, compare magnitudes
          JSR CMPR        Positive, is DCOP > DCAC?
          BCC SB21        Yes, subtract DCAC from DCOP
          LDA #$80        No, set SIGNAC negative
          STA SIGNAC
          BNE SB12        Subtract DCOP from DCAC
DIFSGN    LDA SIGNAC      Fetch SIGNAC
          ADD #$80        Change SIGNAC to opposite
          STA SIGNAC      Store back in SIGNAC
          JMP SAR2        Add DCOP to DCAC
NEGATV    JSR CMPR        Compare DCAC to DCOP
          BEQ NEG1        Equal, make sign positive
          BCC SB21        Subtract DCAC from DCOP
NEG1      LDA #$00        DCOP < DCAC, SIGNAC positive
          STA SIGNAC
          BEQ SB12        Subtract DCOP from DCAC
```

Using these routines as a base, expanded decimal arithmetic programs can be written. One possible addition might be to include a decimal point by specifying either a fixed number of digits in the DCAC and DCOP to be to the right or left of the decimal point, or setting up a memory location to define the exponent. The exponent may reside in one or more bytes of memory and also have a sign byte associated with it. By following the procedures outlined in Chapter Five, one may develop a floating point program using decimal values for the mantissa and exponent. The following routines may be used to perform the multiplication and division operations



of this type of floating point program.

The multiplication and division routines both operate with a four-byte accumulator and operand. The first three bytes contain the six BCD digits of the respective values. The fourth byte is an extension of each value to allow for an overflow during the calculations. The fourth byte must be cleared before entering either of these routines. Also, a memory location is set aside for both routines to store a digit counter value. This location, labeled DIGCNT, is initially set by these routines to the number of significant digits of the accumulator. As the operations proceed, this value is decremented; when it reaches zero, the operation is complete.

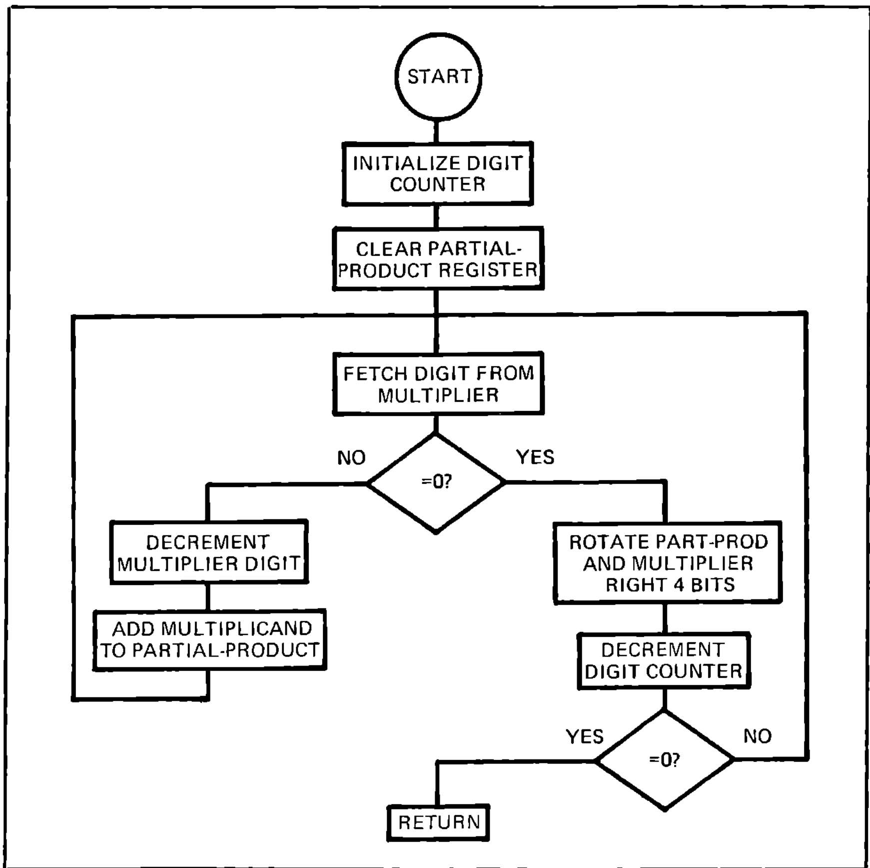
A table area labeled DCPP is used to store the partial-product and quotient for the respective operations. This table consists of

seven bytes, which allows enough room for the multiplication routine to maintain any overflow that may occur.

Multiplication Routine

The multiplication routine multiplies the contents of the decimal operand by the contents of the decimal accumulator. Beginning with the least significant digit of DCAC and working up, each digit is used as a counter for the number of times the operand is to be added to the partial-product. When the counter goes to zero, the contents of the partial-product are rotated right, which achieves the same result as multiplying the operand by ten. The next digit of the accumulator is then selected as a counter for the number of times the operand is added to the partial-product. This multiplication loop is executed once for each significant digit of the decimal accumulator. At the completion, the contents of DCPP0 to DCPP5 contain the 12 significant digits of the result. If this routine is used in part of a floating point program, the results should be normalized. This is accomplished by shifting the partial-product register to the left until a nonzero BCD digit is in the most significant half of the most significant byte. This normalization process follows the same general outline as that defined in Chapter Five.

```
DIGCNT   *=*+1          Digit counter
TMPCNT   *=*+1          Temporary counter storage
DCPP0    *=*+1          Partial-product LS Byte
DCPP1    *=*+1
DCPP2    *=*+1
DCPP3    *=*+1
DCPP4    *=*+1
DCPP5    *=*+1          Partial product of MS Byte
DCPP6    *=*+1          Partial product extension
DCOP     *=*+2          DCOP storage
DCOPM    *=*+2          DCOP MS Byte and extension
DCAC     *=*+2          DCAC storage
DCACM    *=*+2          DCAC MS Byte and extension
DECMUL   LDA #$06       Set digit counter
         STA DIGCNT     Store in memory
         LDX #$07       Set precision counter
         LDY #$00       Initialize index pointer
         STY TOPNT+1    Initialize page of TOPNT
         STY FMPNT+1    Initialize page of FMPNT
         LDA #< DCPP0   Fetch low portion of DCPP0 address
```




```
        STA TOPNT       Store pointer in TOPNT
        JSR CLRMEM      Clear partial-product area
        SED             Set decimal mode flag
NXTDGT  LDA DCAC        Fetch LS Byte of DCAC
        AND #$0F        Mask off upper half
        BEQ DIGDON      If zero, no need to multiply this digit
        STA TMPCNT      Store digit in temporary counter
        LDX #<DCPP3     Set pointer to partial-product storage
        STX TOPNT       Store in TOPNT
        LDX #< DCOP     Set pointer to operand
        STX FMPNT       Store in FMPNT
        LDX #$04        Set precision counter
        JSR DECADD      Add DCOP to partial-product
        DEC TMPCNT      Decrement digit multiplier
```

```
            BNE MULTPL       Not zero, continue multiply loop
DIGDON      LDA #$04         Set rotate counter
PPSHIFT     LDY #$07         Set precision counter
            LDX #<DCPP6      Set pointer to partial-product
            JSR ROTATR       Rotate partial-product right
            LDY #$03         Set precision counter
            LDX #<DCACM      Set pointer to DCAC
            JSR ROTATR       Rotate partial-product to right
            SEC              Set carry for decrement
            SBC #$01         Decrement rotate counter
            BNE PPSHIFT      Not done, continue rotating
            DEC DGTCNT       Decrement digit counter
            BNE NXTDGT       Not zero, continue multiplication
            RTS              Return
```

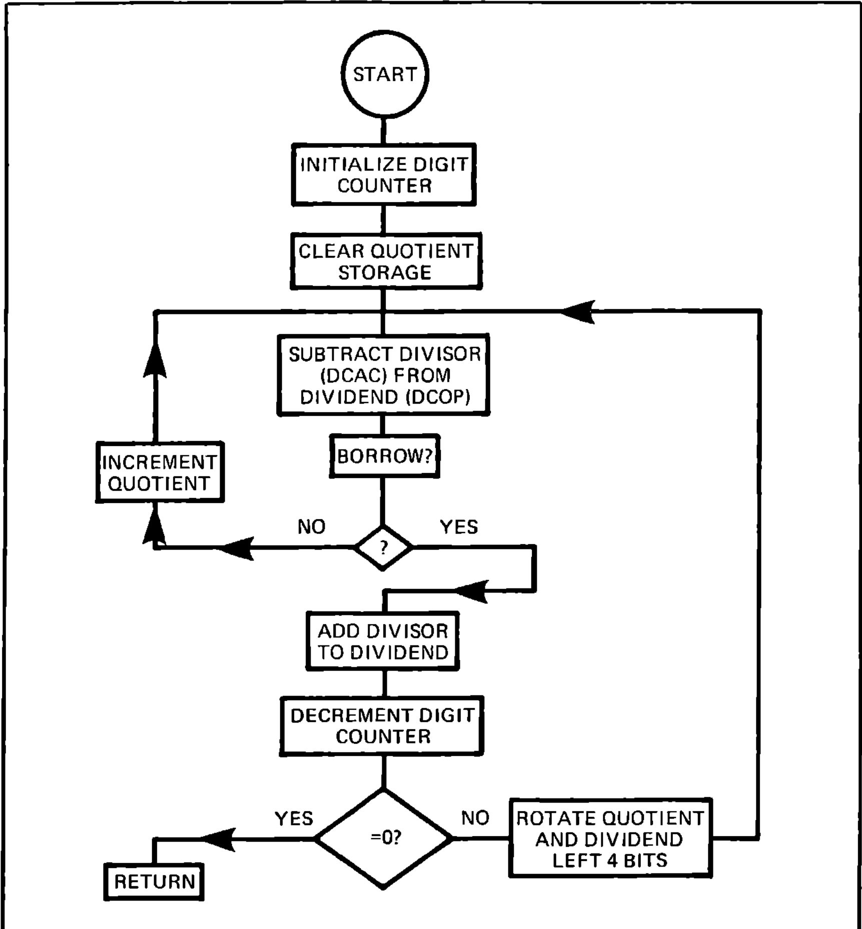
Division Routine

The decimal division routine operates in a manner similar to the binary to decimal conversion routine of Chapter Four. That is, it subtracts the divisor from the dividend until a borrow is required. A count of the number of times the subtraction is successfully performed is maintained. This becomes part of the quotient. When the borrow is detected, the routine rotates the dividend four bits to the left, and the subtraction cycle begins again. As each digit of the quotient is generated, it is shifted into the least significant digit of the quotient.

Before calling this routine, the divisor and dividend must be loaded into the DCAC and DCOP as normalized decimal numbers. Once again, to normalize these decimal values, the most significant nonzero BCD digit must be in the most significant digit location of the respective values. At the completion of this routine, the quotient is contained in DCPP1 through DCPP3. As compensation for the operation of the routine, a value of one must be added to the exponent of the quotient.

```
DECDIV      LDX #$06         Set up digit counter
            STX DIGCNT       Store digit counter in memory
            LDA #<DCPP0      Set up low portion of DCPP0 address
            STA TOPNT        Store in TOPNT for clear routine
            LDY #$00         Set up index pointer
            STY TOPNT+1      Initialize page portion of TOPNT
            STY FMPNT+1      Initialize page portion of FMPNT
            JSR CLRMEM       Clear quotient storage
```

```
        SED              Set decimal mode flag
DVNEXT  LDX #< DCOP      Set pointer to DIVIDEND
        STX TOPNT        Store in TOPNT
        LDX #< DCAC      Set pointer to DIVISOR
        STX FMPNT        Store in FMPNT
        LDX #$04         Set precision counter
        JSR DECSUB       Subtract DIVISOR from DIVIDEND
        BCC SUBDON       If borrow, exit subtraction
        INC DCPP1        Increment decimal counter
        BNE DVNEXT       Continue divide loop
```



```
SUBDON   LDX #$04      Set precision counter
         JSR DECADD    Add DIVISOR back to DIVIDEND
         DEC DIGCNT    Decrement digit counter
         BEQ DVEXIT    Equal zero, return
         LDA #$04      Set rotate left counter
RESULT   LDX #<DCPP1   Set pointer to QUOTIENT
         LDY #$03      Set precision counter
         JSR ROTATL    Rotate QUOTIENT left
         LDX #<DCOP    Set pointer to DIVIDEND
         LDY #$04      Set precision counter
         JSR ROTATL    Rotate DIVIDEND left
         SEC           Set carry for decrement
         SBC #$01      Decrement rotate counter
         BNE RESULT    Not zero, continue rotating
         BEQ DVNEXT    Continue division loop
DVEXIT   RTS           Return
```

Chapter 7

Input/Output Processing

Writing a program to communicate with a peripheral device is as important as almost any other programming task one may have to perform. Nearly every program requires some form of input or output.

The input data may be received from a group of sensors that make up a burglar alarm system. Or, it may be entered through a keyboard device for a variety of control or data entry purposes. Input also could come from a bulk storage device, such as magnetic tape, for loading programs or reading large blocks of data. The output data may be used to turn relays or lights on and off, send characters to a display (such as a mechanical printer or video display), or to store programs or data on a bulk storage device. No matter what the task, it is important to be able to write effective I/O driver programs.

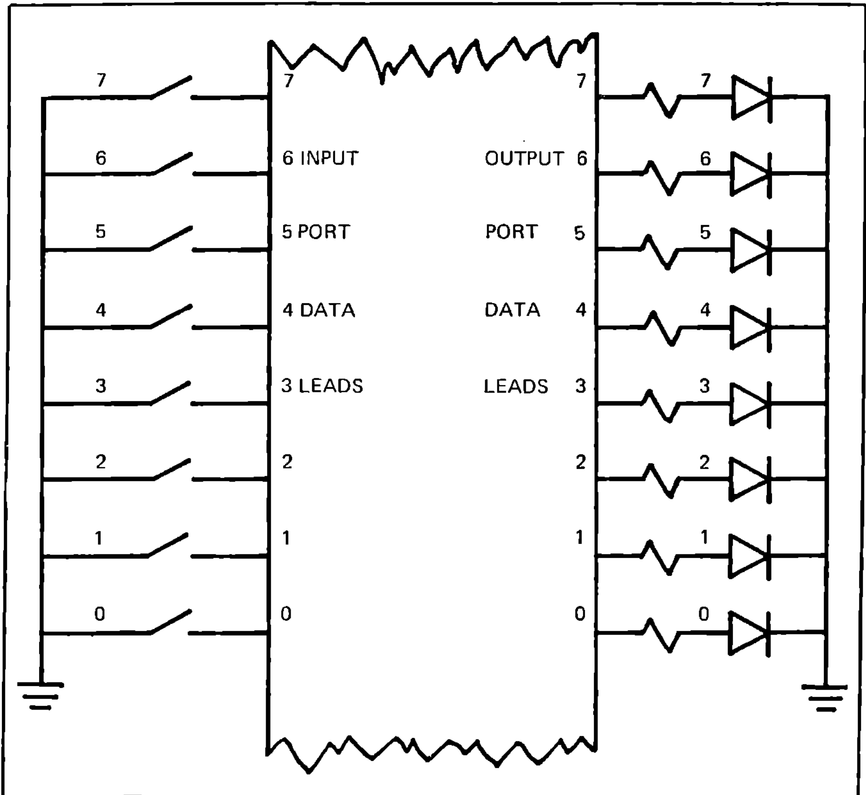
Before the various forms of I/O routines are presented, it is important to understand the input/output setup of the 6502. The 6502 CPU handles input and output in the same manner as reading and writing to the memory. This means that any addressable memory location may be used as an eight-bit parallel I/O port. Therefore, it is possible to have 64K of eight-bit parallel I/O devices on one system. This would be impractical since some memory would be required to store the program to operate the I/O devices. The method of accessing an I/O port as though it is a location in memory allows the use of any of the memory access instructions to transfer data to and from the I/O devices. With this capability, the programmer is afforded considerable flexibility in testing and transferring data with an I/O device.

Some Ground Rules for Discussion

The following convention will be assumed for the I/O ports.

An input port consists of eight parallel data lines that provide true logic to the 6502 CPU. True logic means that a logic "1" transmitted by the input device is seen by the 6502 as a logic "1." An output port is assumed to consist of eight parallel data lines that receive data written to it by the 6502 and maintains the eight-bit data at the output port lines until another data pattern is written to the output port.

The first type of I/O processing to be discussed is one that would be used in conjunction with the simplest form of input and output devices. The input device might be a group of switches, or sensors, that provide a "1" or "0" to each of the input data lines to indicate an open or closed position. The output device might consist of a group of lamps that may be turned on by outputting a "1," or off by outputting a "0." The schematic diagram below illustrates this configuration.



Several Methods of Data Input

As indicated by this diagram, the input port has eight switches, numbered zero through seven, connected to its corresponding eight data leads. These switches might be sensor switches in a burglar alarm system that monitor the opening and closing of doors throughout a building. Assuming that the switches are closed when the doors are properly secured, the following input routine may be used to test for an open door. The label SWINP refers to the memory address of the switch input port.

```
SWTEST   LDA SWINP        Read switch input port
         BEQ SWTEST       If zero, all doors closed, continue
                            testing
         . . .            One or more doors open, alarm
                            condition
```

This routine illustrates the simplicity of inputting information from an input port. The data is read into the accumulator by the LDA instruction. Each bit of the accumulator now indicates the open (1) or closed (0) condition of the switches connected to the input port, and the status flags are conditioned to indicate whether one of the switches is open. For this example, the Z flag will be set to "1" if all the switches are closed. Should any of the switches become open, the data lead corresponding to that switch will go to a "1" condition, and the Z flag will be reset, since the accumulator will not be "0."

It is not necessary to use all eight data leads of an input port. Suppose there are only five switches, zero through four, connected to the input port. The other three leads are not used. In this case a different test procedure would be required. The program listing below loads the accumulator with a value of $1F, and the BIT instruction is used to test for a one in any of the five least significant bits of the input port. The Z flag would indicate the possible open condition of one or more of the five switches.

```
SWTEST   LDA #$1F         Set the bit test byte
         BIT SWINP        Test five least significant bits
         BEQ SWTEST+$2    If zero, all doors closed, continue
                            testing
         . . .            One or more doors open, alarm condi-
                            tion
```

Suppose one data lead was required. By connecting it to bit seven of the input port, the N flag could be used in testing for a "1" or "0" condition. In this case, the conditional branch instruction in the first listing would be changed to a BPL instruction.

Output to Light the LEDS

At the output port, a set of eight lights shown as light emitting diodes, is connected to the eight output port data leads, numbered zero through seven. Each light is turned on by outputting a "1" to the corresponding data lead. The light is turned off by outputting a "0." For example, to turn on every other light, one could load the accumulator with a bit pattern of "10101010" and store it in the output port, as listed below. The label LIGHTS refers to the memory location assigned to the output port.

```
        ...
        LDA #$AA       Load the desired bit pattern
        STA LIGHTS     Output pattern to LIGHTS
        ...
```

These LIGHTS might be connected to the control panel of the burglar alarm system. They could be used to indicate which of the doors have been opened by including an instruction to output the data to the LIGHTS as it is read from the switches. The following sequence may be used.

```
SWTEST  LDA SWINP      Read switch input port
        STA LIGHTS     Output switch conditions to display
        BEQ SWTEST     If zero, all doors closed, loop back
        ...            One or more doors open, alarm condi-
                         tion
```

After inputting the data from the switches, the routine immediately outputs the same data to the LIGHTS. In so doing, any light that turns on will indicate that the corresponding door is open. The program then tests for a door open, just as before, and either continues testing, if the doors are all closed, or performs whatever logic may be necessary when a door is found to be open (i.e., sounding an alarm, calling the police, etc.).

Applications for This Simple Interface

Naturally, the switches and lights used in this example may be

replaced by a wide variety of devices for an even greater number of applications. The input may come from heat, light, or pressure transducers. Or, it can come from analog-to-digital converters, which transform an analog signal to a proportional digital binary, or BCD value. An output port may drive relays, seven segment displays, alarms, or digital-to-analog converters.

A novel application for a simple output device is to connect a speaker to one bit of an output port and have the computer synthesize different frequencies to create music. The different tones are generated by outputting alternate ones and zeros with an appropriate delay in between each output. The shorter the delay, the higher the frequency, and vice versa. By outputting a given set of tones in the proper sequence with each tone lasting the proper duration, a musical tune can be played by the computer. Many 6502-based microcomputers have been known to play such intriguing songs as "Mary Had a Little Lamb," and "A Bicycle Built for Two."

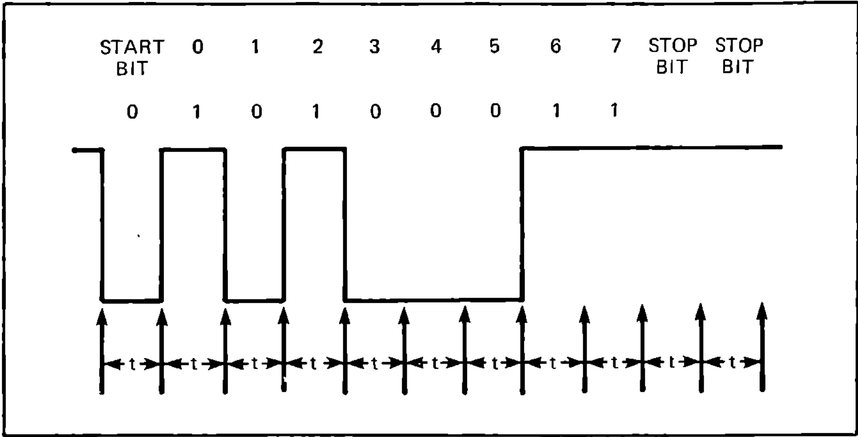
Looking at this application from a more scientific viewpoint, this form of frequency synthesis may be used to generate any number of different waveforms for a multitude of technical applications.

Generating Serial Data

One such technical application is in the generation of asynchronous serial data. Serial data is data that is sent one bit at a time with each bit lasting a specific amount of time before the next bit is output. Asynchronous serial data is a short group of bits output in serial form. Each group of bits generally represents a single character of one of the standard character sets (i.e., ASCII, BAUDOT), although random data patterns may be transmitted in this fashion. It is referred to as asynchronous, because the beginning of the group of bits may occur at any time. However, once started, the timing of each bit in the group must meet the specified time. The timing diagram illustrates the manner in which the ASCII code for the letter "E" (11000101 in binary) is transmitted as asynchronous serial data.

As noted in the timing diagram, the character code for the "E" is preceded by a start bit. This bit is used to inform the receiving device that a character is being transmitted. The character code then follows the start bit, beginning with the least significant bit. The character transmission is completed by adding one or more stop bits to the end of the code. The stop bits are added to allow time for the receiving device to prepare to receive another character.

The timing diagram also indicates that there is a specific amount



of time, "t," for the duration of each bit. This timing is often referred to by the number of bits that could be transmitted in one second at this rate, rather than the amount of time used for each bit. The standard bit per second, or BAUD, rates used for transmitting ASCII code, range from 110 BAUD for many keyboard and printer devices, to 9600 BAUD for high-speed devices.

Programmed Delay Creates the BAUD Rate

The computer may be used to generate serial data in this form, by outputting one bit at a time, and providing a programmed delay between each bit to create the proper timing. The routine listed next outputs eight-bit characters as asynchronous serial data with two stop bits. The timing generated by this routine outputs data at a rate of 110 bits per second. This corresponds to a delay between bits of 9.09 milliseconds. The timing may be calculated by adding up the number of cycles per instructions (indicated in the column of figures to the left of the listing) for each instruction executed between the output of each bit. This timing assumes a cycle time of one microsecond.

This routine may be used to output ASCII characters to a printer or other type of device that receives asynchronous serial data at 110 bits per second. The character to be output must be in the accumulator when this routine is called. The initial contents of the X and Y index registers are pushed onto the stack at the start of this routine, and then pulled from the stack before returning. The output of each bit is accomplished by rotating it into bit zero

of the accumulator, and storing it in the memory location assigned to the output port. One should make special note of the fact that the instructions between the output and rotate operations do not affect the carry flag. This allows the routine to maintain the character for outputting as each bit is transmitted.

```
PRINT       STA TEMP      Save initial character
            TYA           Move Y to A and
            PHA           Save Y on the stack
            TXA           Move X to A and
            PHA           Save X on the stack also
            LDA TEMP      Fetch character
            CLC           Clear carry for start bit
            ROL A         Rotate carry into A
            JSR BITOUT    Output start bit
2           LDY #$08      Set data bit counter
6 PRINT1    JSR BITOUT    Output data bit and delay
2           DEY           Decrement bit counter
3           BNE PRINT1    Not zero, output next bit
2           LDA #$01      Set up stop bit
4           STA PRINTR    Output stop bit
6           JSR TIMER     Delay for one stop bit
6           JSR TIMER     Delay from second stop bit
            PLA           Fetch initial X value
            TAX           Restore in X
            PLA           Fetch initial Y value
            TAY           Restore in Y
            LDA TEMP      Fetch initial character
            RTS           Return
4 BITOUT    STA PRINTR    Output bit to printer
2           ROR A         Position for next output
6           JSR TIMER     Delay one bit time
6           RTS
2 TIMER     LDX #$D2      Set delay counter value
6 TIME1     JSR DUMMY     Jump to return instruction to
6           JSR DUMMY     Provide delay using
6           JSR DUMMY     X index register as delay counter
2           NOP           Added for delay
2           DEX           Decrement delay counter
3           BNE TIME1     Not zero, continue loop
6           JSR DUMMY     Added for delay
6 DUMMY     RTS           Return
```

Shaking Hands with the Computer

The type of peripheral devices discussed require nothing more than a simple input or output instruction to transfer the information. When a transfer is to be made, the program does not care what state the peripheral is in, previous to the transfer. However, for many peripherals, the process of transferring data between it and a computer under program control requires some type of handshaking. This means that a program must check whether the device is ready to make a data transfer, and, when so indicated, perform the logic necessary to make the transfer. In general, there are two methods used to provide the program control. One method is to have the program continuously input the status bit of the peripheral, often referred to as the "programmed data transfer" (or PDT) bit, until it indicates the device is ready for a data transfer. The other method is for the peripheral device to send a signal to the computer when it is ready for a data transfer. This signal is called an interrupt. Once an interrupt is received, the method of data transfer is similar to that for the PDT operation.

The major difference between the two modes is that under PDT operation, the program must continuously check the status of the device. Under interrupt operation, the program is free to perform other operations while waiting for the interrupt from the peripheral.

Utilizing the PDT Bit

Whether a peripheral device is designed to generate interrupts or operate strictly in the PDT mode, there is generally a PDT bit associated with it. A device that generates interrupts will have a PDT bit to provide the option of operating in the PDT mode. When operating under interrupt it is used to identify itself as the device that generated the interrupt, should there be more than one interrupting device in the system. It is, therefore, important to understand how to check the PDT bit of a device. Any peripheral that is designed to operate with a PDT bit will have a status output. This output may contain only the PDT bit, or it may include several other status leads to indicate error conditions that may occur in the peripheral. These status leads are connected to an input port allowing the status to be examined by a program.

There are several ways of checking the PDT bit, depending on its location within the memory byte. If located in the most significant bit of the status byte, loading the accumulator with the status will set the N flag to indicate the condition of the PDT bit.

```
CKPDT    LDA STATUS    Load status byte into A
         BPL CKPDT     If PDT = zero, continue testing it
         . . .         PDT = one, device ready, begin
                         processing
```

If the PDT bit is located in a bit position other than bit seven, the BIT test instruction may be used. The accumulator must be loaded with all zeros except for the bit corresponding to the location of the PDT bit in the device's status byte. Then, by performing the BIT test between the accumulator and the device's status, the Z flag will indicate the opposite condition of the PDT bit. The following routine checks the PDT bit as bit one until it indicates that the device is ready.

```
CKPDT    LDA #$02      Set bit to test the PDT
         BIT STATUS    Condition the Z flag for the PDT test
         BEQ CKPDT     If Z set, device not ready
         . . .         If Z reset, device is ready
```

Anticipate I/O Problems

There are times when it is known that a PDT bit must change within a certain amount of time. For instance, after outputting a character to a display device there is usually a specific maximum time limit for the device to accept it and the PDT bit to come true again. If this time limit is surpassed, it might indicate a problem with the display device. This possible error may be monitored by the program by inserting a counter in the PDT test loop. The counter would be calculated to allow only a given amount of time to elapse before the PDT bit must return. Otherwise an error routine would be entered to inform the operator of a possible problem. The following format may be used to include a timer in the PDT checking routine. The exact timing of this loop may be calculated as discussed in Chapter Three.

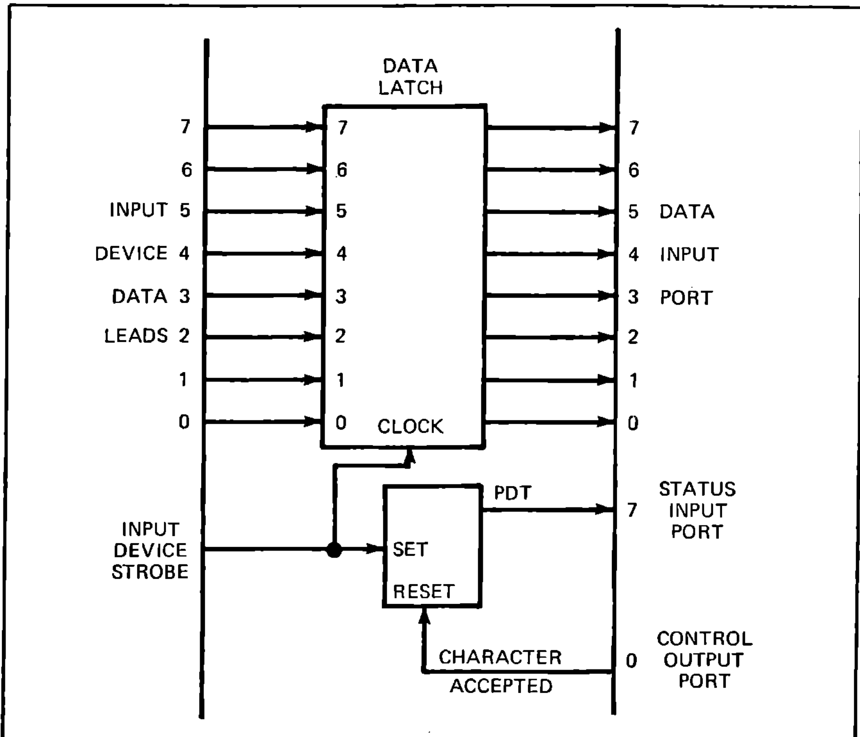
```
LPSET    LDY #$YY      Set up timing loop counter
CKPDT    LDA STATUS    Condition N flag for PDT test
         BMI PDTEST    Have PDT, continue processing
         DEY           No PDT, decrement timer
         BNE CKPDT     Timer ≠ zero, continue testing
         . . .         Time out, possible error
```

Data Input with PDT Control

PDT operation, for most input devices, generally follows the same basic procedure. When a program requires data from an input device, it reads the status of the device and checks the condition of the PDT bit. If the PDT bit indicates the device has data available, the program can proceed to input the data. For some devices, once the data has been read in, a "character accepted" signal from the program may be required to reset the PDT bit.

This procedure is typical of many interfaces that latch the data in from a device and then set a PDT bit. The diagram shown next illustrates this form of interface. The data is entered by setting up the data at the input to the latches and then pulsing the strobe line of the latches. This same strobe signal sets the PDT bit. After the data has been read by the program, the reset line is pulsed by the program outputting a "character accepted" signal.

A program to control this type of interface is listed next. The PDT bit is connected to bit seven of the status input port. This



allows the program to check for the PDT bit by reading the status and testing for the condition of the N flag. The data is entered through the data input port. Once the data has been accepted, the PDT bit is reset by outputting a "character accepted" signal to the control output port.

```
PDTINP   LDA STATUS     Input device status
         BPL PDTINP     N = zero, no PDT, continue testing
         LDA DATAIN     N = one, read data from device
RESET    LDY #$01       Set up output pulse
         STY CHRACC     Output character accepted
         LDY #$00       Clear to create pulse
         STY CHRACC     Reset character accepted
         RET            Return
```

In this program listing, the "character accepted" signal is derived by first loading index register Y with $01 and outputting it to the control port, labeled RESET. Then, Y is cleared and output to the control port. This effectively creates a pulse on the least significant data lead of the control port. Some interfaces are reset by simply writing to the control port. A third possibility is that the interface resets the PDT bit when the input is executed. In this case, the input routine may be exited just after the data is read.

The label RESET has been included in this routine to point out the portion of the routine that resets the PDT bit. This portion may be required as an initial reset for the input device at the start of a program that uses the device to receive data. Quite often when dealing with such devices, it is necessary to output a reset during the initializing stages of the program. This guarantees that the device status will indicate the true status when the device is first called upon to input some data. For the other cases in which the PDT is reset by writing to the control output or reading from the data input, the corresponding instruction should be executed to initialize the input device.

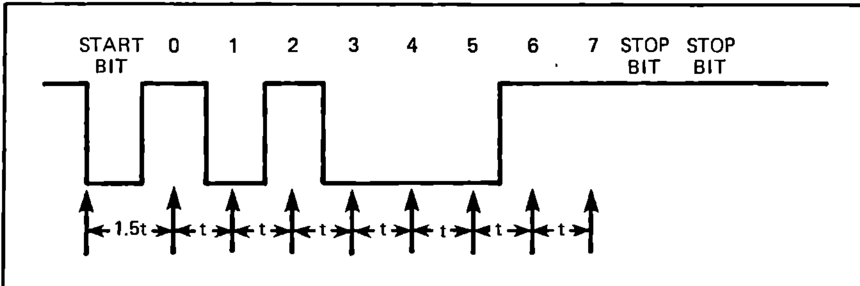
Receiving Serial Data

Another routine that tests the status before inputting the data is one which inputs asynchronous serial data. The start bit of the asynchronous data could be considered its PDT bit. The input routine would test for the presence of the start bit. When detected, the data bits that follow may be read in by sampling the data at the proper time intervals.

Sampling the data is performed by providing a programmed delay until the midpoint of each bit is reached. The bit is then read in on the input data lead. The arrows in the timing diagram shown next indicate when each bit should be sampled by the program. The left arrow indicates when the start bit is first detected. The next arrow indicates a delay time equal to one and a half bits before sampling bit zero of the data. Each subsequent sample is taken after a delay time of one bit.

The next program may be used to receive eight bits of asynchronous serial data. Such data may be generated by the PRINT routine previously mentioned. The timing provided in this routine reads the data at 110 BAUD. By altering the delay, this timing may be changed to input data over a wide range of BAUD rates. The number of cycles for each instruction is indicated in the left-hand column. The delay time between samplings may be calculated by adding up the number of cycles for each instruction executed between inputs. The major portion of the delay is provided by the same TIMER subroutine used in PRINT.

The data is input through bit seven of the data input port. This allows the program to test simply the N flag for the start bit. Each bit is then input by rotating bit seven of the input port into the carry and then rotating the carry into the accumulator. When the last bit has been input, an additional delay of one bit time is added to make sure the input data is into the stop bit before returning to the calling program. If this final delay was not provided, and the last data bit of the input was a "0," the calling program could call SRLINP and would input the last data bit that is still at the input port. If this occurs, SRLINP would assume it to be the start bit of a new character. This would result in the input of erroneous data. The data received is returned to the calling program in the accumulator.



```
SRLINP     TYA             Save initial value of Y
           PHA             On the stack
           TXA             Save initial value of X
           PHA             On the stack
4 SRLOOP   LDA KYBDIN      Input to look for start bit
3          BMI SRLOOP      N flag = one, no start bit yet
2          LDA #$00        Have start bit, clear A
6          JSR HAFBIT      Delay one-half bit time
6          JSR TIMER       Delay one bit time
2          LDY #$08        Set data bit counter
6 NEXBIT   ROL KYBDIN      Move data bit into carry
2          ROR A           Move carry into accumulator
6          JSR TIMER       Delay one bit time
2          DEY             Decrement bit counter
3          BNE NEXBIT      Not zero, input next bit
           JSR TIMER       Delay one bit time
           STA TEMP        Temporarily save data
           PLA             Fetch initial value of X
           TAX             Restore X
           PLA             Fetch initial value of Y
           TAY             Restore Y
           LDA TEMP        Restore data received in A
           RTS             Return
2 HAFBIT   LDX #$6E        Set one-half bit delay time
3          JMP TIME1       Delay one-half bit time
```

Output Data with a Specific Format

PDT operation of a parallel output device is generally straightforward. When the PDT bit is checked and indicates the device is ready to accept data, the data may be output. Upon receipt of the data by the device, the PDT bit will change state to indicate that the device is busy processing the data. Once the processing is completed, the PDT will return to its device ready status and wait for the next output from the program. Therefore, if the program is to output more than one character, the PDT bit must be monitored after each character is output to determine when the device is ready to accept the next character.

The following routine might be used to output a line of text to a printer that accepts ASCII characters as eight-bit parallel data. This routine fetches the characters one at a time from a buffer and outputs them to the printer. When a carriage return is detected in the character string, it is transmitted, followed by a line feed. The program then re-

turns to the calling program. The PDT is assumed to be in bit zero of the status input from the printer. Also, when the routine is called, FMPNT is assumed to be pointing to the start of the character string.

```
LINOUT   LDY #0           Initialize pointer
         JSR CKPDT        Wait for printer PDT
         LDA (FMPNT),Y    Fetch character from message storage
         CMP #$8D         Character = carriage return?
         BEQ FINISH       Yes, complete output
         STA CHROUT       No, output character to printer
         INY              Advance character string pointer
         JMP LINOUT       Wait for PDT
FINISH   STA CHROUT       Output carriage return
         JSR CKPDT        Check PDT before sending line feed
         LDA #$8A         Set line feed
         STA CHROUT       Output line feed
         RTS              Return to calling program
CKPDT    LDA #$01         Set up to test PDT
         BIT STATUS       Test status of printer PDT
         BEQ CKPDT+$2     PDT = zero, wait for printer
         RTS              PDT = one, return to output next char
```

This method of checking the PDT bit of an I/O device is commonly used when it is not required to perform other functions while waiting for a data transfer from a peripheral. In cases where a background program is not necessary while waiting for a peripheral, it is of no consequence for the CPU to dedicate itself to testing the PDT bit. Such constant attention to the PDT bit allows data to be transferred as rapidly as possible. This is also necessary if the data is only available for a given length of time. A card reader is a good example. The program must read a character from the card reader when it is available. Once the reader starts reading a card, it does not stop in between each character while the program reads it. The program must be ready for each character when it is available. Otherwise, the character will be lost.

Data Transfer Using Interrupts

Another method of transferring data under program control is to have the I/O device send an interrupt signal to the computer when it is ready for a data transfer. This signal interrupts the program currently in progress and directs the CPU to an interrupt service routine. The interrupt routine performs the logic necessary to transfer the data to or

from the peripheral and then returns to the original program as though it had not been interrupted at all. This method of operation is known as interrupt processing.

Interrupt processing is analogous to a postman being interrupted while delivering the mail. As the postman is placing the mail in a row of mailboxes, someone walks up to him and taps him on the shoulder. The mailman completes filling the current mailbox, makes a mental note as to which mailbox is to be filled next, and turns to the person. The mailman is given a letter and is asked to send it. The mailman takes the letter and stores it in his mailbag. He then returns to the job of filling the mailboxes, beginning with the box he remembers as the next one to be filled. This is similar to the procedure followed by a computer when an interrupt is received from a peripheral.

When an interrupt signal is received, the current instruction being executed is completed. The address of the next instruction to be executed is saved on the stack. Also, it is necessary to save the information contained in the CPU status flags so that it may be properly restored before returning to the interrupted program. The computer is now ready to perform the steps necessary to transfer the data between itself and the peripheral. Once the transfer is completed, the status flags must be restored to their initial contents at the time the interrupt was received. Execution of the interrupted program is resumed at the instruction that would have been executed next.

The 6502 Interrupt Structure

Before presenting methods of interrupt processing with the 6502, several features should be discussed which make interrupt processing easy and effective. There are two types of hardware interrupts available. One is the nonmaskable interrupt. When received, the nonmaskable interrupt is always acknowledged. For this reason, the nonmaskable interrupt is generally used for very high-speed devices that require immediate attention, or as a power failure interrupt to allow the storage of any critical information.

The other hardware interrupt is the maskable interrupt. Its acknowledgement is dependent on the condition of the I flag. When the I flag is set, the maskable interrupt line is disabled. An interrupt on the maskable interrupt line will not be acknowledged by the 6502. When the I flag is reset, the 6502 will acknowledge a maskable interrupt. The maskable interrupt is generally used by most of the devices that operate under interrupt control. For a maskable or nonmaskable interrupt, the interrupt service routine will be basically the same.

The condition of the interrupt flag may be software controlled by one of three instructions. The SEI instructions sets the I flag, disabling the maskable interrupt. The CLI instruction clears the I flag, enabling the receipt of maskable interrupts. The third instruction, PLP, conditions not only the I flag, but all of the status flags from the contents of the stack. The stack may be loaded from the accumulator using PHA first. Then, the PLP instruction will move that value from the stack into the status register.

Interrupts Aren't Always Desired

When writing a program to operate with interrupts, there are several times when it may not be desired to accept interrupts. One is during the initialization of the program, before all the necessary pointers, counters and tables used by the program have been set up. If an interrupt is received before the program is ready to accept it, the program may receive or transmit erroneous data. To avoid such an occurrence, the first instruction of the program should be the disable interrupt instruction. Then, after the initialization is complete, the interrupts may be enabled. The program now is ready to deal with the interrupts properly.

Another time that interrupts must be disabled is upon receipt of an interrupt. This is to allow the program enough time to respond to the first interrupt before receiving the second. The 6502 automatically disables the maskable interrupt upon receipt of the maskable, nonmaskable and software interrupts. Therefore, it is not necessary to include a SEI instruction in the interrupt service routine. When the interrupt service routine is finished, the return from interrupt instruction will restore the I flag to its initial condition at the time the interrupt was received. If it is desired to allow nesting of interrupts, the interrupt service routine can enable the maskable interrupt after it has completed its initial steps. Any subsequent interrupt on the maskable interrupt line will be recognized and serviced, even if it is the original interrupting device. The process of nesting interrupts will be discussed later.

It may be necessary to disable interrupts when a section of the program is changing information vital to the function of the interrupt routine. This information might be the address for storing or retrieving data to be transferred. Or, a flag indicating the progress of the program to the interrupt routine. For whatever reason, the program must disable interrupts before the change is made. After changing the information, the interrupts may be re-enabled. This will provide the smooth transition of information needed by the interrupt

routine.

Save Data from the Interrupted Program

Another feature of the 6502 is the automatic status register storage that takes place when an interrupt is acknowledged. The contents of the program counter and the status register are pushed onto the stack. It is necessary to save this information so that it may be restored before returning to the interrupted program. When the interrupt service routine has completed its operation, the return from interrupt instruction, mnemonic RTI, pulls the status register and program counter from the stack. This results in the CPU returning to the interrupted program with no change in its execution.

The accumulator and index register may also be stored in the stack if they are used by the interrupt service routine. This is achieved by pushing the accumulator, transferring the index registers to the accumulator and pushing them onto the stack. At the completion of the service routine, the data must be pulled from the stack and transferred to the proper registers. This program sequence is illustrated next.

```
            ...      Interrupt received, program counter
                     And status register pushed onto stack
INTRPT      PHA      Save accumulator contents on stack
            TXA      Move X index reg to accumulator
            PHA      Save contents of X on stack
            TYA      Move Y index reg to accumulator
            PHA      Save contents of Y on stack
            ...      Process interrupt
            PLA      Fetch original contents of Y
            TAY      Restore Y index register
            PLA      Fetch original contents of X
            TAX      Restore X index register
            PLA      Restore accumulator
            RTI      Restore status and program counter
                       to original contents
```

The procedure for receiving interrupts by a 6502-based microcomputer follows the basic steps described above. When an interrupt is received from a peripheral, the CPU automatically pushes the contents of the program counter and status register onto the stack, and sets the I flag. (For the maskable interrupt, this procedure assumes that the I flag is reset at the time the maskable interrupt occurs.) The

CPU vectors to the proper interrupt service routine.

Service the Interrupting Device

The interrupt service routine performs the logic required to service the interrupting device. It is usually a combination of the PDT routine for the device being controlled, and a routine that checks and stores the data for an input, or sets up the data to be output. The interrupt routine is meant to operate independently from the main program. It must perform its own checks and manipulate the data into and out of memory, as well as drive the peripheral. In order to accomplish this, and to provide a flexible interrupt routine, a link between the main program and the operation of the interrupt service routine must be established.

One method of establishing the link is through the use of an interrupt table area. This table area normally includes at least three items, namely, a memory pointer, a data counter, and an in-progress flag. The memory pointer is used by the interrupt routine to indicate where input data is to be stored, or where output data is to be found. As the interrupt routine stores or outputs each byte of data, the memory pointer is advanced to the next location. The data counter indicates to the interrupt routine the amount of data to be received or sent. The routine decrements this counter each time it inputs or outputs some data. When the counter reaches zero, the operation is complete. If necessary, the end of the operation also may be indicated by the receipt or transmission of a terminating character, such as a carriage return or line feed. This would terminate the operation before the data counter reached zero. The completion of the operation then is signaled by resetting the in-progress flag. The in-progress flag is set by the main program when the input or output is initiated. Then, when the interrupt routine is finished with the I/O operation, the in-progress flag is reset. The main program periodically checks this in-progress flag and, when it is reset, the main program knows that the I/O operation is complete.

The in-progress flag may also serve another purpose. The interrupt routine can test this flag when an interrupt is received to determine whether an interrupt from the peripheral is expected. If it is expected, the interrupt routine can service the interrupt normally. If the interrupt is not expected, the interrupt may be ignored by resetting the I/O device, if necessary, and returning to the interrupted program. Or, an error routine may be entered, which informs either the main program or the computer operator of the erroneous interrupt.

After the interrupt service routine completes its operation, it returns control to the interrupted program. This is accomplished by restoring the CPU registers and executing the return from interrupt instruction. The original status and program counter are pulled from the stack, and the return is made to the interrupted program. Restoration of the status and CPU registers before exiting the interrupt service routine allows the interrupted program to continue execution as though the interrupt never occurred.

Interrupts for Input and Output Differ

Interrupt processing for an input device is not exactly the same as that for an output device. The reason for this difference is that an interrupt from an input device indicates that the input device has a character or some data available for the program. The program may read the data in, process it, and then wait for another interrupt. For an output device, an interrupt indicates that the device has accepted the previous output and is ready to receive another character. Therefore, an output device initially must receive an output from the program before it generates an interrupt. Also, after the last character is received by the output device, a final interrupt will be generated, which must be ignored. This difference is further illustrated by the following input and output interrupt routines.

The input interrupt service routine stores characters as they are input into a buffer area in the memory. This routine continues until either the buffer is filled or a carriage return is received. The routine might be used to input characters from a keyboard or data from a paper tape reader. A table area is used which contains the input buffer pointer, data counter and in-progress flag. This table is listed next, followed by the table set-up routine of the main program. The table set-up routine initializes the contents of the table when an input sequence is to begin.

The in-progress flag in the first byte of the table is represented by the sign bit, not the contents of the entire byte. Therefore, the remaining seven bits in this byte may be used to signal error conditions or intermediate program status. This type of information is often required by the interrupt routine or the main program. Next, the input buffer pointer is stored in the second and third bytes of the table, with the low portion of the address in the second byte, and the page portion in the third byte. The address that must be initially loaded into these locations is the start address of the input biffer minus one. Setting this pointer to the location before the start of the input buffer is necessary because the input interrupt routine increments the

input buffer pointer before storing the character received, not after.

Finally, one should note the use of the set and clear the I flag instructions in the SETINT routine before and after the table is set up. This prevents an interrupt from being acknowledged while the contents of the input interrupt table are being initialized. However, disabling interrupts may not be necessary if the data counter is initialized first, followed by the pointer and finally the in-progress flag. In this way, the pertinent data is loaded into the table before the in-progress flag is set.

Interrupt Input Table

```
FLAGIN    .BYTE $1          In-progress flag, sign bit
          .BYTE $1          Low portion, input buffer pointer
          .BYTE $1          Page portion, input buffer pointer
          .BYTE $1          Data counter
          ...               Set up routine for input
SETINT    LDA #$80          Disable mskbl interrupts during setup
          STA FLAGIN        Store in-progress flag
          LDA #INBFLO       Set low portion of buffer address
          STA FLAGIN+$1     Store in interrupt table
          LDA #INBFPG       Set page portion of buffer address
          STA FLAGIN+$2     Store in interrupt table
          LDA #$XX          Set data counter
          STA FLAGIN+$3     Store counter in table
          CLI               Enable maskable interrupts
          ...               Continue main program
```

The input interrupt service routine is listed followed by the flow chart. The input is performed by a single load instruction. This assumes that the input device is reset by reading the data from its input port. When implementing this routine, the instruction marked by the double asterisk should be replaced by those required to operate the specific device being driven.

It is assumed in this routine that only one device in the system can generate an interrupt. Therefore, it is not necessary to check for the PDT bit of the input device. If one desires to check the PDT bit as an error checking measure, this routine should include an instruction sequence which inputs the PDT bit of the input device and tests the status. If the PDT bit is not set properly, an error routine should be entered. Otherwise, the routine should proceed to input the data and continue with the normal interrupt processing.

After the data has been input, the in-progress flag is checked to

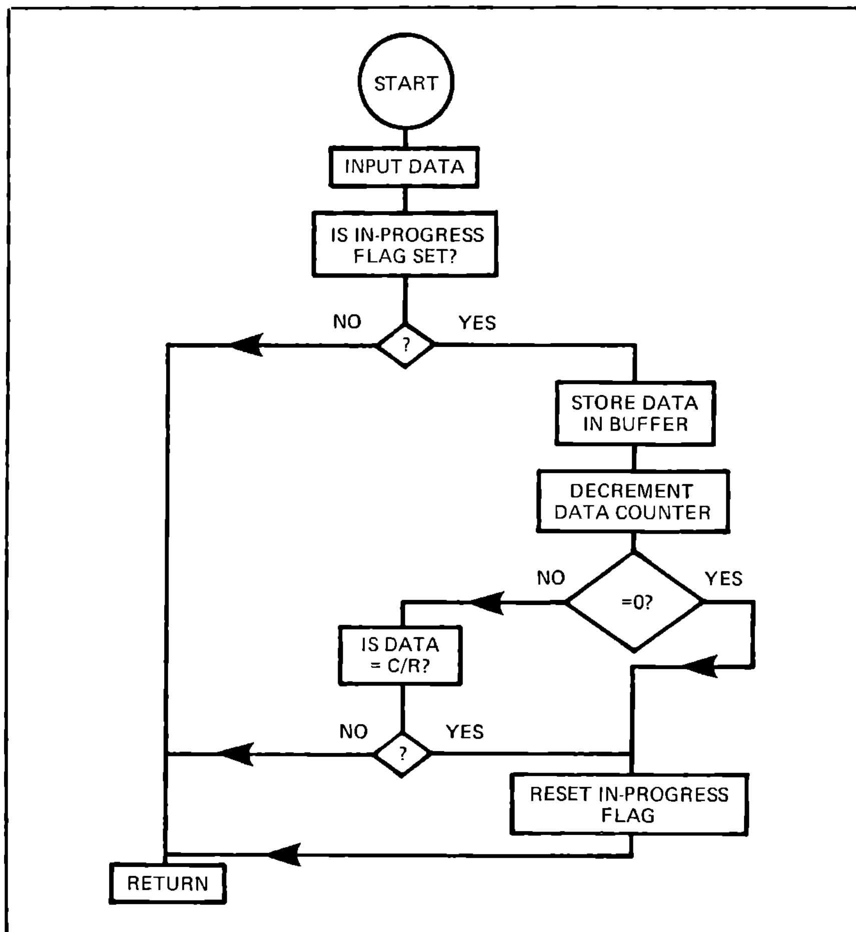
determine whether an input is expected by the interrupt program. This routine ignores an unexpected interrupt by simply returning to the interrupted program without storing the character inputted. It should be noted that by performing the input sequence before checking the in-progress flag, the input device will be properly reset whether the interrupt was expected or not.

Assuming the interrupt was expected, the character received is stored in the input buffer. The new input buffer pointer is then stored in the input interrupt table. The data counter is decremented and, if zero, the in-progress flag is reset and the interrupt service routine is exited. If it is not zero, the character just received is tested for a terminating character. In this routine, the input may be terminated by a carriage return, ASCII code $8D. If it is a carriage return, the in-progress flag is reset to end the input operation and the interrupt routine is exited. If it is not a carriage return, the in-progress flag remains set when the routine is exited.

The short instruction sequence following the interrupt service routine listing may be used by the main program to check for the completion of the input operation. When the sign bit of the in-progress byte is reset, the main program will branch to the appropriate routine, referred to here as CMPTIN, to examine the data received. The contents of the interrupt input table may be used by the main program in examining the data input. The input buffer pointer indicates the location of the last character received. The data counter indicates either the number of unused locations in the input buffer, or, if equal to zero, that the entire buffer is filled.

```
INTINP   LDA INPDAT         ** Input data from input device
         LDY FLAGIN            Check in-progress flag
         BPL EXITIN            Interrupt not expected, ignore
         INC FLAGIN+$1         Increment input buffer pointer
         BNE INTSTR            Not zero, store data
         INC FLAGIN+$2         Increment page portion of pointer
INTSTR   LDX #$00              Clear index pointer
         STA (FLAGIN+1,X)      Store data received in buffer
         DEC FLAGIN+$3         Decrement character counter
         BEQ FININP            If zero, input finished
         CMP #$8D              Is character a carriage return?
         BNE EXITIN            No, exit input routine
FININP   STX FLAGIN            Input complete, clear in-progress flag
EXITIN   ...                   Restore registers and return
         ...                   Sequence to check
```

```
. . .                In-progress flag for end of operation
LDA FLAGIN           Check sign bit of in-progress byte
BPL CMPTIN           Sign reset, input complete
. . .                Sign set, continue other processing
```



Outputting Under Interrupt

Operation of an output device under interrupt control requires a different sequence of events from that for an input device. As pointed out before, the main reason for this difference is that the output device generates an interrupt after a character has been out-

putted by the program. The input device generates an interrupt to indicate that it has a character available. The output routine presented next illustrates the different approach that must be taken for an output device.

The output interrupt routine outputs a string of characters stored in a buffer memory. Such a routine may be used to output messages to a printer or video display, or to output data to a low- or medium-speed storage device. (High-speed devices generally use a method of direct memory access. The data is transferred directly from memory to the storage device, or vice versa, under control of a hardware interface.)

The interrupt output table is the same type of table used to provide the exchange of information between the main program and the input interrupt service routine. The organization of the table is the same as the input table, with the in-progress flag, output buffer pointer, and data counter. However, when the table is initialized, the buffer pointer is set to the actual start address of the output buffer, rather than the start address minus one, as in the input table.

Aside from setting up the table, the initialization routine checks the in-progress flag to determine whether an output is currently being executed. This may occur when a program uses the same output device to display messages from a number of different routines, such as error and advisory messages in a system monitor program. Checking this flag eliminates the possibility of an output being initiated before a previous one is finished. The input routine does not test this flag since it is less likely that two separate inputs will be required at the same time. However, if the possibility does exist, a similar instruction sequence should be added to the input initialization routine before the disable interrupt instruction.

When the in-progress flag is reset, the output may be initiated. First, the output table is set up with the required information. While this table is being loaded, it is not necessary to disable interrupts, since the output device should not generate an interrupt until after the first character has been sent. Once the proper information is contained in the table, the first character is output by this routine. The output is performed by the STA OUTDAT instruction in this listing. This initial output triggers the output sequence which is carried on by the interrupt service routine. For implementation of this routine on one's own system, the instruction in this routine and in the interrupt service routine marked with a double asterisk should be changed to the instruction sequence necessary to drive the specific output device.

Interrupt Output Table

```
FLAGOUT  .BYTE $1            Output in-progress flag
         .BYTE $1            Low portion, output buffer pointer
         .BYTE $1            Page portion, output buffer pointer
         .BYTE $1            Output data counter
         . . .               Output initialization routine
TSTOUT   LDA FLGOUT          Check in-progress flag
         BMI TSTOUT          If output in progress, wait
         LDA #$XX            Set character counter
         STA FLGOUT+$3       Store in output interrupt table
         LDA #OUTBFL         Set low portion of buffer address
         STA FLGOUT+$1       Store in interrupt table
         LDA #OUTBFP         Set page portion of buffer address
         STA FLGOUT+$2       Store in interrupt table
         LDA #$80            Set in-progress flag
         STA FLGOUT          Store in output interrupt table
         LDX #$00            Set up buffer pointer
         LDA (FLGOUT+$1,X)   Fetch first character to output
         STA OUTDAT**        Output character to device
         . . .               Continue main program
```

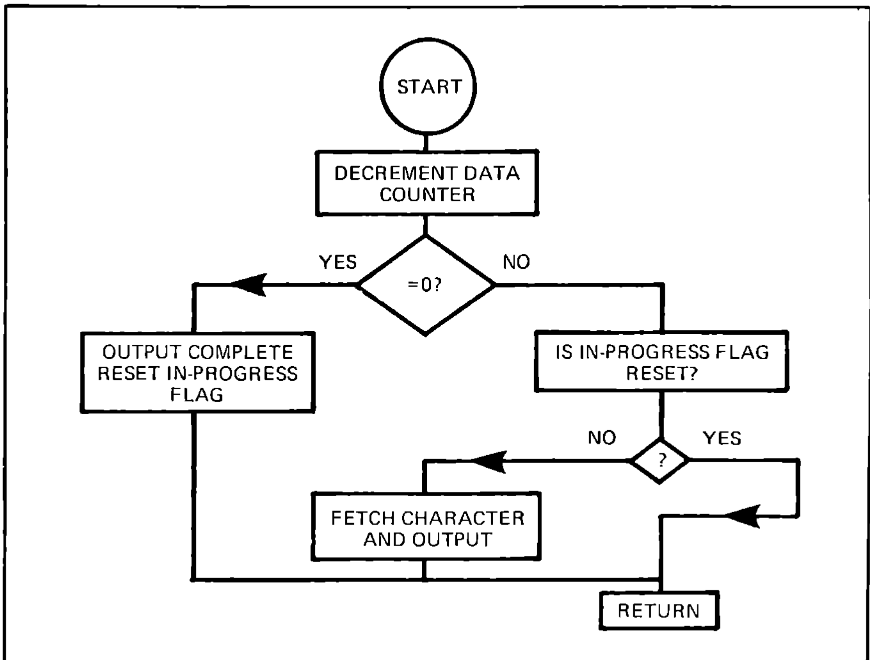
The output interrupt service routine is entered upon receipt of an interrupt from the output device. The data counter is decremented once and checked for zero. When it reaches zero, the last character has been output and the output operation is complete. The in-progress flag is reset, and the routine returns to the interrupted program.

If the counter is not zero, the in-progress flag is checked to make sure that the output routine is expecting an interrupt. As in the input interrupt service routine, this is indicated by the in-progress flag being set. If it is reset, the interrupt may be ignored by simply returning to the interrupted program. Otherwise an error routine may be entered that signals either the main program or the operator that an unexpected interrupt was received.

If the routine makes it by the test, the next character may be outputted. In this routine, it is assumed that the output device is the only device generating interrupts. Thus, a PDT test is not necessary before outputting the character. However, if it is felt that such a test should be performed before outputting the character, the required instruction sequence for testing the PDT bit may be included. The routine then returns to the interrupted program.

```
INTOUT   DEC FLGOUT+$3   Decrement character counter
```

```
            BEQ FLGRST          = zero? Yes, reset in-progress and exit
            LDX #$0             Clear the index register
            LDA FLGOUT          Check in-progress flag
            BPL EXITOT          Reset, ignore interrupt
            INC FLGOUT+$1       Advance output buffer pointer
            BNE XMIT            Not zero, continue
            INC FLGOUT+$2       Advance page portion
XMIT        LDA (FLGOUT+$1,X)   Fetch character to be output
            STA OUTDAT   **     Output character
EXITOT      . . .               Restore registers and return
FLGRST      STX FLGOUT          Reset in-progress flag
            JMP EXITOT          Return to interrupted program
```



Several Devices on One Interrupt

Only one device has been considered to generate an interrupt. When an interrupt is received, the interrupt service routine simply performs the indicated input or output for the single device. This may not always be the case, since an I/O controller quite often controls an input and an output device, and generates an interrupt for

both devices. Or, there may be several interrupt devices connected to the system. In order to control more than one device, the interrupt service routine must determine which device generated the interrupt by polling each device when an interrupt is received.

Polling means that the interrupt service routine checks the status of each device that could have generated the interrupt. This is done by checking the PDT bit of each of the devices. When a PDT bit is found to be set, the appropriate service routine is entered to execute the I/O for that device. At the conclusion of the service routine, the return from interrupt instruction sequence is executed to return to the interrupted program.

The following listing is an example of a polling routine that checks the status of three possible interrupting devices. This instruction sequence should be the initial sequence of the interrupt routine. The labels DVICE1, DVICE2 and DVICE3 refer to the interrupt service routines that perform the I/O logic for the designated device. This routine tests the PDT bit of each device and jumps to the proper service routine when a PDT bit is set. If none of the possible devices have the PDT bit set, this routine ignores the interrupt and returns to the interrupted program. This condition may be treated as an error condition, if necessary, rather than ignoring it.

```
...            Polling routine
LDA PDTDV1     Test status of device 1
BMI DVICE1     If PDT set, service device 1
LDA PDTDV2     Test status of device 2
BMI DVICE2     If PDT set, service device 2
LD PDTDV3      Test status of device 3
BMI DVICE3     If PDT set, service device 3
...            None set, ignore interrupt
```

Nesting Interrupts for Fast Service

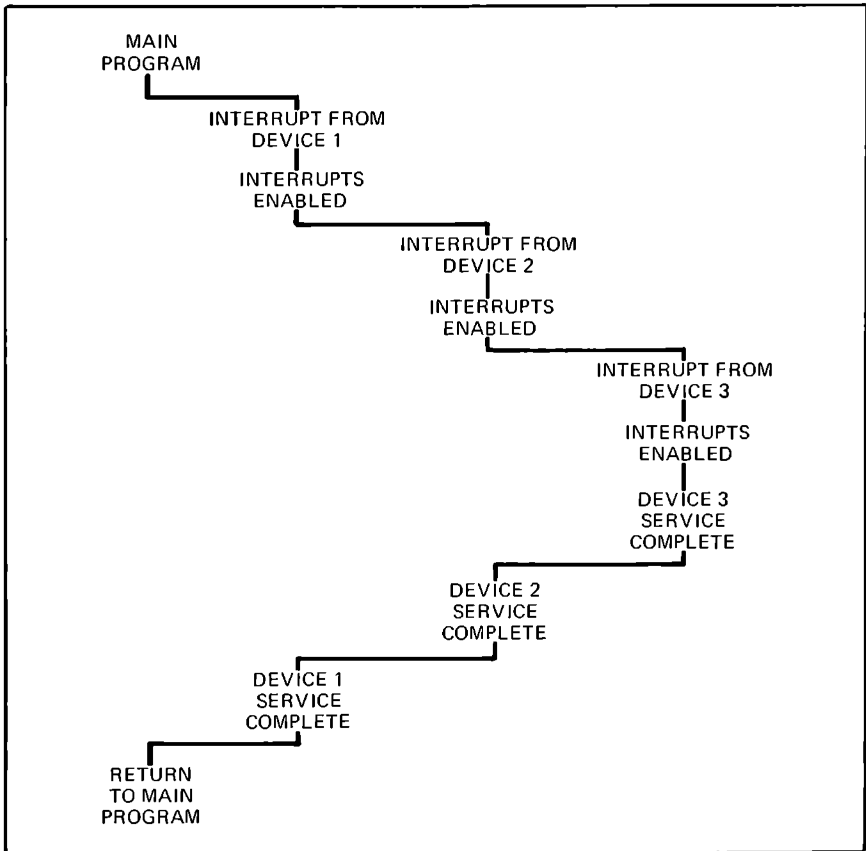
The use of several interrupting devices in a system may require that the interrupt service routines allow receipt of an interrupt from one device while another is being serviced.

This means that the service routine of the first interrupting device must enable interrupts before it has completed its operation. Then, if an interrupt from a second device occurs before this routine is finished, the current interrupt routine being executed becomes the interrupted program of the second interrupt. Allowing interrupts to overlap in this manner is referred to as nesting interrupts.

The illustration shows how the flow from one interrupt routine

to another would proceed if three interrupting devices generated interrupts within a short period of time, to create the nesting of three levels of interrupts.

The 6502 stack plays an important role in nesting interrupts. Saving the CPU registers and status in the stack allows the interrupt routines to interrupt each other without setting up special pointers and data storage areas for each interrupt level or device. The only restrictions on the number of nesting levels, as far as the stack is concerned, is the amount of memory provided for use by the stack. Each interrupt may use six memory locations in the stack to store the registers. Therefore, for every interrupt nesting level one can expect, there must be six memory locations available in the stack. One must also allow for other uses of the stack by the main program (i.e., subroutine calls, temporary data storage).



Deciding when to enable interrupts in an interrupt service routine to allow nesting is generally determined by the speed of the device being serviced. A low-speed device may allow interrupts to be enabled immediately, since it is likely that a swift data transfer is not required. A medium-speed device, or a device that has a limited amount of time to transfer the data, may require the data transfer to be performed before the interrupts are enabled. Then, the remainder of the service routine may be executed while the interrupts are enabled. If the high-speed device is being operated in the interrupt mode, it is very likely that it should not allow interrupts to be enabled until the end of its service routine. If it were to enable interrupts, a slower device might delay the execution of the high-speed device's service routine to the point that a second interrupt from the high-speed device would be received before the initial interrupt had been serviced completely. Therefore, one should carefully consider which routines, and where in the routines, the interrupts are to be enabled.

This method of selecting when to enable interrupts is a means of setting a priority for the interrupting devices. The high-speed devices would have the highest priority, since they do not allow themselves to be interrupted until the service routine is finished. The medium-speed devices, which may enable interrupts after several operations of the service routine have been completed, would be considered a middle priority. The low-speed devices would be the lowest priority because they may be interrupted at any time during the interrupt service routine.

Such a system of priorities may be augmented by the computer's hardware, if a priority interrupt interface is used. This interface fields the interrupts from the interrupting devices and allows the higher priority interrupts through first, before those of lower priority. The interrupt software for setting up priorities for the interrupts received is greatly simplified by this type of interface.

When deciding whether to operate a computer system's peripherals under PDT control or interrupt, one should consider the type of programs (along with the number of peripherals) to be used in the system. If the programs are the type that receive an input and then output a response to a single terminal, the PDT mode would be the easiest to implement, and would provide sufficient performance. Programs of this type include games, editors and small system monitors. For programs that provide keyboard entry and storage or retrieval from a bulk storage device to enter and store mailing lists, for example, one should consider interrupt processing. This would allow the

data entry and bulk storage to be performed simultaneously. However, if the speed of the storage device is fast enough to store each entry with a minimal delay imposed between entries, the PDT mode may work just as well. For programs that operate a number of peripherals simultaneously and, in essence, are running more than one or two programs at a time, the interrupt mode of operation is a necessity. Such multi-programmed systems might be used to control several terminals at once, while monitoring a burglar or fire alarm system. Therefore, one should carefully consider the overall requirements for the type of programs to be run when setting up the I/O portion of one's system.

Chapter 8

Search and Sort Routines

The capability of a computer to manipulate data stored in its memory is another reason why the computer is such a powerful machine. The speed with which it can search large blocks of data and extract information, or sort the data into alphabetical order, or into common groupings, is far beyond the capability of a human. The information may represent a wide variety of data. For example, the data could consist of names and addresses that are to be searched for those in specific geographic regions. The list may be sorted into alphabetical order, or the data may be numerical information, such as test grades or data gathered for a research project. In order for the computer to perform these tasks, the information to be processed must be arranged in the memory in a specific format. Then, programs must be written to perform the desired operation.

Structure of Tables

The data to be manipulated must be arranged in some form of table in the memory. The table may contain a number of entries. Each entry may consist of one or more bytes of memory, depending on the maximum size of a single entry and the format specified. The two types of tables to be discussed in this chapter are commonly referred to as fixed-format and free-format tables. In a fixed-format table, the data is arranged in a standard fashion for each entry. The same number of bytes is assigned to each entry, no matter how many bytes an entry may actually take up. A free-format table allows the size of each table entry to follow the data pattern of the entry. If the first entry requires four bytes and the next requires six, in a free-format table, the first entry will only use the four bytes and the second will use six. There are advantages to both formats, depending on the application. These will be discussed as the search routines for each format are presented.

In order to provide a means of comparing the two formats, two search routines will be presented that perform the same function. However, one routine utilizes a search table of a fixed format and the other utilizes a free-format search table. The function performed by these search routines is that of receiving a command input from a keyboard and searching a control table for the same command. If the command is found, the start address of the command routine is taken from the table and is used to jump to that command routine. If the command is not found in the table, the search routine returns to wait for another command input. These routines have many practical applications. The entries in the control table may be modified easily to represent as many commands as one may need for a specific program. Any program that allows an operator to input commands to direct its operation may find one of these routines useful.

Same Data — Two Different Formats

The following control tables are used to illustrate the operation of the fixed-format and free-format search routines. The commands in these tables are: GO, LIST, MEDIAN, AVG, COUNT, and ERASE. These commands might be used to direct the computer to aid in conducting an experiment. The GO command could initiate a 10-second sampling interval, during which time a sensor is monitored to detect the occurrence of an event. A count of the number of times the event occurs within the 10-second interval is stored in the computer by the GO command routine. The LIST routine might be used to print out the counts stored for each 10-second interval up to that time. This would allow one to examine the raw data for possible patterns that may develop. The MEDIAN and AVG commands could calculate the median and average values of the counts stored for each interval, and output the value to a printer. The COUNT command might be used to indicate the number of 10-second intervals that have been initiated up to that time. The ERASE command could be used to reset the storage area to allow a new set of tests to begin.

In both the fixed-format and free-format control tables, each entry is divided into two fields. The first field consists of the character string that defines the command name. In the fixed-format entry, this field is set to a fixed length. In this case, it is six characters long. For the command names that do not use all six locations available for the name, the unused locations are filled with zeros. In the free-format entry, the command field contains the characters for the name plus one more location that contains a zero byte. This extra location is used to indicate the end of the name. The second field

is the same for both formats. This field is two bytes long and contains the start address of the command in the entry.

The end of each control table is indicated by a zero byte stored immediately following the last entry. By terminating the tables in this way, the search routines simply may check the first character of each entry for a zero byte to determine when the end of the table is reached. Therefore, the number of entries in the control table is completely independent of the operation of the search routine. Modifying the number of commands in the control table is accomplished by adding or deleting the command entries and moving the zero byte to the end of the new control table. The search routine does not have to be changed at all.

The control table for the fixed-format search routine is presented here, followed by the control table for the free-format routine. Note the difference in length between the two tables caused by the extra zeros that must be added to the fixed-format entries.

Fixed-Format Control Table

0200	C7	Code for letter G
0201	CF	Code for letter O
0202	00	Not used for this command
0203	00	Not used for this command
0204	00	Not used for this command
0205	00	Not used for this command
0206	40	Location on page where GO starts
0207	03	Page where GO routine starts
0208	CC	Code for letter L
0209	C9	Code for letter I
020A	D3	Code for letter S
020B	D4	Code for letter T
020C	00	Not used for this command
020D	00	Not used for this command
020E	60	Location on page where LIST starts
020F	03	Page where LIST routine starts
0210	CD	Code for letter M
0211	C5	Code for letter E
0212	C4	Code for letter D
0213	C9	Code for letter I
0214	C1	Code for letter A
0215	CE	Code for letter N
0216	80	Location on page where MEDIAN starts
0217	03	Page where MEDIAN routine starts

0218	C1	Code for letter A
0219	D6	Code for letter V
021A	C7	Code for letter G
021B	00	Not used for this command
021C	00	Not used for this command
021D	00	Not used for this command
021E	A0	Location on page where AVG starts
021F	03	Page where AVG routine starts
0220	C3	Code for letter C
0221	CF	Code for letter O
0222	D5	Code for letter U
0223	CE	Code for letter N
0224	D4	Code for letter T
0225	00	Not used for this command
0226	C0	Location on page where COUNT starts
0227	03	Page where COUNT routine starts
0228	C5	Code for letter E
0229	D2	Code for letter R
022A	C1	Code for letter A
022B	D3	Code for letter S
022C	C5	Code for letter E
022D	00	Not used for this command
022E	E0	Location on page where ERASE starts
022F	03	Page where ERASE routine starts
0230	00	**End of table marker**

Free-Format Control Table

0200	C7	Code for letter G
0201	CF	Code for letter O
0202	00	*End of command word marker*
0203	40	Location on page where GO starts
0204	03	Page where GO routine starts
0205	CC	Code for letter L
0206	C9	Code for letter I
0207	D3	Code for letter S
0208	D4	Code for letter T
0209	00	*End of command word marker*
020A	60	Location on page where LIST starts
020B	03	Page where LIST routine starts
020C	CD	Code for letter M
020D	C5	Code for letter E
020E	C4	Code for letter D
020F	C9	Code for letter I

0210	C1	Code for letter A
0211	CE	Code for letter N
0212	00	*End of command word marker*
0213	80	Location on page where MEDIAN starts
0214	03	Page where MEDIAN routine starts
0215	C1	Code for letter A
0216	D6	Code for letter V
0217	C7	Code for letter G
0218	00	*End of command word marker*
0219	A0	Location on page where AVG starts
021A	03	Page where AVG routine starts
021B	C3	Code for letter C
021C	CF	Code for letter O
021D	D5	Code for letter U
021E	CE	Code for letter N
021F	D4	Code for letter T
0220	00	*End of command word marker*
0221	C0	Location on page where COUNT starts
0222	03	Page where COUNT routine starts
0223	C5	Code for letter E
0224	D2	Code for letter R
0225	C1	Code for letter A
0226	D3	Code for letter S
0227	C5	Code for letter E
0228	00	*End of command word marker*
0229	E0	Location on page where ERASE starts
022A	03	Page where ERASE routine starts
022B	00	**End of table marker**

As mentioned before, the lengths of the two tables differ because of the variation in the number of characters for each command name. If, however, all of the names were six characters long, the fixed-format table would be shorter than the free-format. The command field name in the free-format table would require seven bytes to store each name — six for the name and one for the terminating zero byte.

Fixed-Format Input Routine

Another consideration when deciding which format to use is the type of input programming required to enter the commands. There are several different methods that may be used to input and store the command to be searched for in the control table.

One method is to initially clear out the input buffer area by filling it with zero bytes. Then, as each character is entered, it is stored in the input buffer. When a carriage return is entered, the input is terminated and the contents of the input buffer may be used to search for the command. If the command entered does not fill the input buffer, the unused locations will contain zero bytes. This method is best suited for the fixed-format, since the input buffer will contain the same contents as the command name field of the matching command in the control table.

The following routine could be used to clear the input buffer and store the characters in the buffer as discussed above. This routine uses the CLRMEM subroutine in Chapter Three to clear the input buffer. The INPUT routine that is called must input a character from the input device (such as an ASCII keyboard), echo it to a display and return with the character in the accumulator. Along with the test for the carriage return, to terminate the input and return, the character count is checked. When the input buffer is full, any additional characters that may be inadvertently entered before the carriage return are ignored. The initial instruction sequence may be used as a control routine to call the individual routines, including the search routine to be presented later.

When a match is found, the FOUND routine is entered. This routine takes the address from the address field of the matching control entry and uses it to jump to the command routine. The address is moved to FMPNT on page zero. The jump indirect instruction directs the CPU to the appropriate command routine. After the command routine completes its operations, it may return to the main control program simply by executing a return instruction.

Since this routine compares the entire input buffer against the command name field of the control table entries, it is not necessary for it to test the input buffer or command name for a terminating character. However, a counter must be set to the number of characters in the command name field so that the routine will know when all of the characters have been compared. In this routine, this counter is set to six. If one changes the length of the command name field, this counter must also be changed to reflect the new length.

The listing for this fixed-format search routine is presented next, followed by the flow chart. The flow chart also includes the logic flow of the main control routine when used in conjunction with this search routine.

```
NEXCMD    LDA #$0        Set page portion of input pointer
```

```
        STA TOPNT+1     Store in TOPNT
        LDX #INPBFR     Set low address of input pointer
        STX TOPNT       Store in TOPNT
        LDX #$06        Set precision counter
        JSR CLRMEM      Clear input buffer storage
        JSR INCMND      Fetch command string from input
        JSR SRCHFX      Search table and execute command
        JMP NEXCMD      Repeat loop for next command
INCMND  LDX #INPBFR     Set pointer to input buffer
        LDY #$06        Set counter to buffer size
INCHAR  JSR INPUT       Call routine to input character
        CMP #$8D        Is character a carriage return?
        BNE CHECK       No, continue input
        RTS             Yes, return, input complete
CHECK   CPY #$00        Is character counter = zero?
        BEQ INCHAR      Yes, ignore new character
        DEY             Else, decrement counter
        STA $0,X        Store character in buffer
        INX             Advance input buffer pointer
        BNE INCHAR      Fetch next character
```

Free-Format Input Routine

Another method of inputting the characters is to leave the input buffer contents as is at the start of the input routine. As each character is received, it is stored in the input buffer. When a carriage return is received, the input is terminated by storing the carriage return in the input buffer and returning. Thus, the input buffer area must be assigned one byte more than the maximum of characters assigned for a command name. This method is more advantageous for the free-format search routine. It sets up the command entered in a similar format to that used in the command name field of the free-format control table entries.

The only real diference between this input routine, labeled INCTRL, and the previous INCMND routine is the instruction sequence that stores the carriage return as the terminating character in the input buffer before returning. Also, one should note the absence of the routine that clears the input buffer before inputting the command. This saves quite a few memory locations. The initial instruction sequence is a sample control routine for directing the operation of the command search function.

```
NEXCMD  JSR INCTRL      Input command from input device
```

```
         JSR SRCHFR     Search table and execute command
         JSR NEXCMD     Repeat loop for next command
INCTRL   LDX #INPBFR    Set pointer to start of input
         LDY #$06       Set counter to buffer size
INCHAR   JSR INPUT      Call routine to input character
         CMP #$8D       Is character a carriage return?
         BNE CHECK      No, check for full buffer
         STA $0,X       Yes, store carriage return in buffer
         RTS            Return, input complete
CHECK    CPY #$00       Character count zero?
         BEQ INCHAR     Yes, ignore current input
         DEY            Decrement character counter
         STA $0,X       Store character in buffer
         INX            Advance buffer pointer
         BNE INCHAR     Loop to input next character
```

Searching the Fixed-Format Table

The search routine for a fixed-format control table compares the contents of the input buffer to the command name field on a character-by-character basis. This is done by calling the CPRMEM subroutine, which is presented in Chapter Three. This subroutine may be included in the SRCHFX routine if it is not used elsewhere in one's program.

If the characters in the input buffer do not match any command name fields in the control table, the NXWORD routine is entered to advance the control table pointer to the start of the next entry. At this point the first character of this entry is checked for the zero byte, which indicated the end of the control table. If the zero byte is not found, the routine jumps to the compare routine to check for a match between the new control entry and the input buffer. When the zero byte is encountered, it indicates that the entire table has been searched and no match has been found. The routine then returns to the control routine to initiate a new command entry.

It may be desirable at this point to have the search routine print out a message, and if no match is found, to inform the operator of the error. This may be done by changing the RTS instruction in the SETNXW routine to a branch or jump instruction, which jumps to a message output routine to print the message before returning to the main control program.

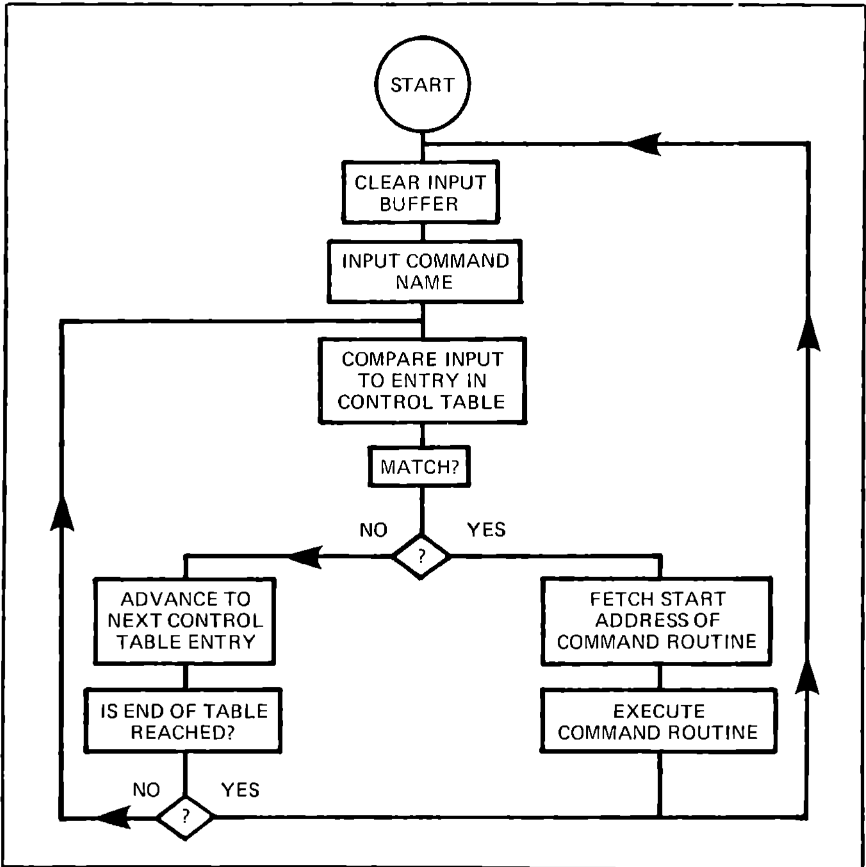
When a match is found, the FOUND routine is entered. This routine takes the address from the address field of the matching

control entry and uses it to jump to the command routine. The address is moved to FMPNT on page zero. The jump indirect instruction directs the CPU to the appropriate command routine. After the command routine completes its operations, it may return to the main control program simply by executing a return instruction.

Since this routine compares the entire input buffer against the command name field of the control table entries, it is not necessary for it to test the input buffer of command name for a terminating character. However, a counter must be set to the number of characters in the command name field so that the routine will know when all of the characters have been compared. In this routine, this counter is set to six. If one changes the length of the command name field, this counter must also be changed to reflect the new length.

The flow chart below includes the logic flow of the main control routine when used in conjunction with this search routine.

```
SRCHFX   LDX #>CMDTBL   Set pointer to start of table
         STX FMPNT 1    Store page portion in FMPNT
         LDX #CMDTBL−1  Set low portion of table pointer
         STX FMPNT      Store in FMPNT
         LDA #INPBFR−1  Set pointer to input buffer
         STA TOPNT      Store in TOPNT
INITBF   LDY #$06       Initialize byte counter
CMATCH   JSR CMPMEM     Compare table entry to input
         BEQ FOUND      Both equal, process command
NXWORD   LDA FMPNT      Fetch table pointer
         CLC            Clear carry for addition
         ADC #$08       Advance pointer to next entry
         STA FMPNT      Restore in FMPNT
         LDY #$01       Set index pointet to first character
         LDA (FMPNT),Y  Is end of table reached?
         BNE INITBF     No, continue table search
         RTS            Yes, entry not found, input command
FOUND    LDY #$07       Set pointer to command address
         LDA (FMPNT),Y  Fetch low address
         TAX            Save in X
         INY            Advance table pointer
         LDA (FMPNT),Y  Fetch page portion of address
         STX FMPNT      Store start address of command
         STA FMPNT 1    Routine in FMPNT
         JMP (FMPNT)    Jump to command routine
```



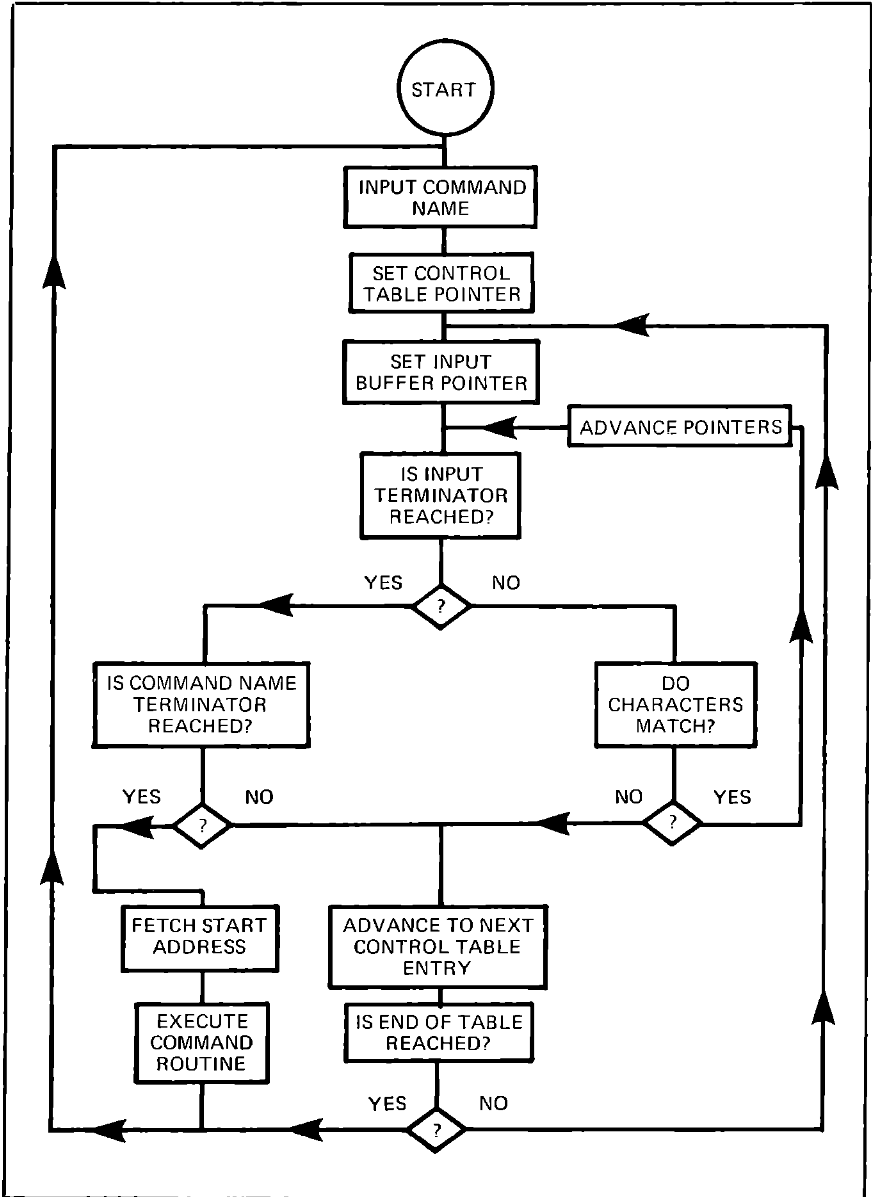
Free-Format Search Routine

The free-format search routine that follows has the same basic flow of the fixed-format routine. That is, it compares the input buffer to an entry in the control table. If they do not match, it advances the control table pointer to compare the next entry against the input buffer. This continues until either a match is found, or the end of the buffer is reached. But, instead of comparing a fixed number of characters to get a match, this routine compares the contents of the input buffer, up to the terminating carriage return, against each command name field in the control table. If the input buffer and command name field match to the carriage return in the input buffer, the corresponding location in the command name field is checked for its terminating character, a zero byte. If the zero byte

is found, the FOUND routine is entered to fetch the address from the control table entry and jump to the proper command routine. Here again, if the suggested main control routine for the free-format is used, the command routines may return to the main control routine by executing a return instruction.

```
SRCHFR   LDX #INPBFR      Set pointer to input buffer
         STX TOPNT        Store in TOPNT
         LDX #<CMDTBL     Set pointer to start of command table
         STX FMPNT        Store in FMPNT
         LDX #>CMDTBL     Set page portion of table address
         STX FMPNT+1      Store in FMPNT
         LDY #$00         Initialize index pointer
         STY TOPNT+1      Store page portion of input address
CMATCH   LDA (TOPNT),Y    Fetch entry from input buffer
         CMP #$8D         Is character a carriage return?
         BEQ LCHAR        Yes, check for end of command
         CMP (FMPNT),Y    No, is entry equal to command?
         BNE NXWORD       No, advance to test next command
         INY              Yes, advance index pointer
         BNE CMATCH       Check next character
LCHAR    LDA (FMPNT),Y    Is end of control field here?
         BEQ FOUND        Yes, found matching control word
NXWORD   LDA (FMPNT),Y    Test for end of control field
         BEQ SETNXW       If found, advance to next block
         INY              Otherwise, advance command pointer
         BNE NXWORD       And continue looking
SETNXW   INY              Advance pointer over the
         INY              Address field to start of
         INY              Next control word field
         CLC              Clear carry for addition
         TYA              Move index pointer to accumulator
         ADC FMPNT        Advance command table pointer
         STA FMPNT        Restore pointer in FMPNT
         LDY #$00         Reset index pointer
         LDA (FMPNT),Y    Is next control word the end?
         BNE CMATCH       No, compare to input buffer
         RTS              Yes, entry not found, return
FOUND    INY              Advance pointer to address field
         LDA (FMPNT),Y    Fetch low portion of command
         TAX              Save in X
         INY              Advance index pointer
         LDA (FMPNT),Y    Fetch page portion of pointer
```

```
STX FMPNT       Store pointer to start of
STA FMPNT+1     Command routine in FMPNT
JMP (FMPNT)     Jump to command routine
```



Why Sort Out Data?

The process of sorting data into some specified order, or into groups with common attributes, is another of the computer's powerful capabilities. For example, it is often desired to arrange a list of names and addresses in alphabetical order according to the last names. Or, one may want to sort out persons living in a common geographical location by sorting the addresses according to Zip code. In order to program these sort functions, the data first must be organized into carefully structured tables.

Another Data Structure

Proper structuring may be accomplished by creating fields in which specific items are to be located. These fields are set up in, a similar manner to the fields in the search table entries. For this sort routine, a fixed-format table is used. This table contains a list of names which are to be arranged in alphabetical order. For illustrative purposes, the following names are used as table entries.

BROWN, L.R.
DALEY, D.R.
ANDERSON, B.
DARBY, P.
MATTOX, R.T.
MATTHEWS, K.D.
JONES, A.M.

Each element of the names in this list must have a specific field assigned to it. Looking at the last names, one may observe that the longest name is eight characters long. However, there are many names that use more than eight characters. Therefore, a field of 14 bytes for the last name will be used to accommodate the longer names. One-byte fields will be set up for each of the initials. This makes a total of 16 bytes for each entry. The delimiters (the comma and period) are not assigned any location in the tables since a fixed format is to be used. The inclusion of these punctuation marks would only serve to take up table space. The delimiters are used, however, when entering the names to be stored in the table.

A Data Entry Routine

The following program is one possible means of entering the names into the table in the properly formatted fields. This routine accepts the names as keyboard entries in the format illustrated

above. Each field is accepted and stored in its proper location in the table for that entry. A name input is terminated by entering a carriage return. The delimiters terminate a field entry and advance the pointer to the next field in the entry. If a field is not filled by the name, such as a last name with less than 14 characters, or one with no middle initial, the unused locations in the field are filled with zero bytes. After the final name has been entered, the operator may input an asterisk to indicate the completion of the input operation.

ASCII Helps Keep Sorting Simple

The character input is expected to be ASCII characters. The ASCII character set is a well-ordered set of character codes. The letters A through Z are coded in consecutive order from C1 to $DA and the numbers zero through nine are coded in order from $B0 to $B9. This is especially useful when sorting into alphabetical or numerical order, since the lower the code for the character, the lower it will place the entry in the sorted list.

Before listing this routine, several comments about its operation must be given. First, the characters are received by calling a routine referred to by the label INPUT. This routine must be provided by the user to accept characters from the input device associated with one's system. This routine must return the ASCII code for the character input in the accumulator upon returning. As a visual indication for the operator to verify the characters entered, this INPUT routine should also output the characters received to the system display device before returning. The contents of the index registers must be used to point to a conversion table or whatever, they should be saved and then restored before returning. This INPUT routine should also check for the receipt of a carriage return character. When received, the input routine should also output a line feed character to the printer device, since this is not provided for in the name input routine.

Before this routine is called, the table area must be properly set up. This is accomplished by storing a zero byte in the first location in the table. The zero byte is used to indicate the location for storing the next name entered. Initially, this byte must be set up at the start of the table by the calling program. Then, as each name is entered into the table, the zero byte is moved up to the location immediately following the last name entered in the table. Before each name input is initiated, the location of this byte in the table is checked. If the zero byte is not found within the limits of the

table area, it is assumed that the table is filled. At this point, a routine, to be supplied by the user and referred to here as TOMUCH, would be entered. TOMUCH should output a message to the operator indicating that the table is filled. The limits of the table area in this sample routine begin at page $04, and end at page $07 location $FF. Thus, when the table pointer is advanced to the start of page $08, the table is full.

After the zero byte is found and the program is ready to accept a name input, it calls the input routine to fetch the first character. There are a number of special codes checked when the first character is entered. One is an asterisk, which is to be entered when the list contains all of the names desired. When this character is received, the routine returns to the calling program, which may then call upon the sort routine to sort the names into alphabetical order. The other special codes checked are the carriage return, comma, or period. If any of these codes are received as the first character, they are ignored. This is because they indicate the end of either an entry or a field. They would not be valid at this time since there are no characters stored as yet for this name.

Once a valid character is entered for the first character, the characters that follow are checked for a comma or carriage return. If the comma is received, the remainder of the last name field is filled with zero bytes, and the portion of the routine that accepts the initials is entered. If a carriage return is received, the remainder of the entire entry is filled with zeros and a new name input is initiated. If the character entered is neither of these two characters, it is entered as the next character in the last name, up to the fourteenth character. If more than 14 characters are entered for the last name, the excess characters are simply ignored.

If, when the first initial entry is to be entered, a carriage return is received, the two initial fields are zeroed and a new name input is begun. If a comma is received, it is ignored. Otherwise, the character is stored as the first initial and the routine jumps to input the second initial. When the second initial is to be entered, receipt of a period is ignored, and a carriage return results in a zero byte being stored for the second initial. Any other input is stored as the second initial. The routine then checks for a full table, and, if not filled, begins a new name input sequence.

```
ACCEPT   LDX #<SRTTBL  Initialize sort table pointer
         STX TOPNT     Store low address in TOPNT
         LDX #>SRTTBL  Set page portion of address
```

```
         STX TOPNT+1     Store in TOPNT
NOTFND   LDY #$00        Clear index pointer
         LDA (TOPNT),Y   Is first entry = zero?
         BEQ FNDEND      Yes, begin input routine
         JSR CKPAGS      No, advance sort table pointer
         JMP NOTFND      Check for last table entry
FNDEND   LDY #$00        Initialize index pointer
         LDX #$0E        Set up last name field counter
         JSR INPUT       Fetch character from input
         CMP #$AA        Check for * code
         BNE NOTDON      Proceed if not *
         LDA #$00        End of input
         STA (TOPNT),Y   Store end of table marker
         RTS             Return to main program
NOTDON   CMP #$8D        Test for carriage return
         BEQ FNDEND      Ignore if first character in field
         CMP #$AE        Test for period
         BEQ FNDEND      Ignore if first character in field
         CMP #$AC        Test for comma
         BEQ FNDEND      Ignore if first character in field
STRCHR   STA (TOPNT),Y   Store character in field
         INY             Advance index pointer
         DEX             Decrement character counter
         BEQ FULFLD      If zero, field is full
         JSR INPUT       Otherwise, input next character
         CMP #$8D        Test for carriage return
         BEQ HAVECR      Finish entry if carriage return
         CMP #$AC        Test for comma
         BEQ HAVECM      Finish last name field
         BNE STRCHR      Jump to store character
HAVECR   INX             Increment counter twice to
         INX             Clear initial fields
         LDA #$00        Set up zero byte
         STA (TOPNT),Y   Store in remaining field area
         INY             Advance index pointer
         DEX             Decrement byte counter
         BNE HAVECR+2    Not zero, continue clearing
         JSR CKPAGS      See if end of boundary
         LDA #$00        Not out of bounds
         TAY             Store zero byte at start
         STA (TOPNT),Y   Of next entry
         BEQ NOTFND      Begin process for next entry
```

```
HAVECM   LDA #$00        Comma entered, clear
         STA (TOPNT),Y   Rest of last name field
         INY             Advance index pointer
         DEX             Decrement field counter
         BNE HAVECM      Continue clearing field
FULFLD   JSR INPUT       Get first initial
         CMP #$AC        Test for comma
         BEQ FULFLD      Ignore comma at this point
         CMP #$8D        Test for carriage return
         BNE SAVIN1      Not CR, store character
         LDA #$00        If carriage return, store
         STA (TOPNT),Y   Zero byte for both initials
         INY             Advance index pointer
         JMP SAVIN2      Jump to clear second initial
SAVIN1   STA (TOPNT),Y   Store character for first initial
         INY             Advance index pointer
INITF2   JSR INPUT       Input next character
         CMP #$AE        Test for period
         BEQ INITF2      Ignore a period at this point
         CMP #$8D        Test for carriage return
         BNE SAVIN2      Not CR, store zero byte
         LDA #$00        If CR, store zero byte
SAVIN2   STA (TOPNT),Y   Store second initial character
         JSR CKPAGS      Check if out of bounds
         JMP FNDEND      If not, process next entry
CKPAGS   CLC             Clear carry for addition
         LDA #$10        Set increment to next entry
         ADC TOPNT       Add to indirect pointer
         STA TOPNT       Save in TOPNT
         BCC CKPAGS      If no carry, return
         INC TOPNT+1     Increment page portion of TOPNT
         LDA TOPNT+1     Fetch page portion of pointer
         CMP #$08        Test for boundary exceeded
         BEQ TOMUCH      Display message if table full
CKPGRT   RTS             Otherwise, return to continue input
```

It may be desired to provide some kind of entry correction capability to this main input routine. One way to accomplish this is to designate another special entry code. When entered, this would cause the program to reset the table pointer to the start of the current entry and initiate a new name input. The routine listed next

may be used by the input routine to check for a control zero character, $8F. This routine checks for the control zero and, if entered, resets the table pointer, essentially erasing the current name input from the table. The start of the name input sequence at the FNDEND label then is entered. Otherwise, it simply returns to continue the input.

```
CHKRUB   CMP #$8F       Check for control zero
         BEQ RESET      If control zero, start new entry
         RTS            Otherwise, return to check character
RESET    PLA            Advance stack pointer to
         PLA            Eliminate return address
         LDY #$00       Reset index pointer to start
         JMP FNDEND     Begin name entry again
```

Now that one has defined the format for storing the data, and developed a means of entering it, a routine may be written to sort the data as desired. The main objective of the sort routine is to examine the contents of the field (or fields) that contain the information pertinent to the sort operation. Then it must rearrange the table contents into the desired order or groups. There are a number of techniques used to do this. The choice generally depends on the type of data and sorting operation to be performed. The sort routine arranges the table contents into alphabetical order by using a ripple sorting technique.

How the Ripple Sort Operates

The term ripple is derived from the manner in which the routine moves through the table to sort the entries into alphabetical order. Beginning with the first entry (N), the sort routine compares it to the next entry (N+1) in the table, If the first entry is lower in alphabetical order than the second, the two entries are left as it. The routine advances to check the order of the second entry (new N) against the third (new N+1). If the first entry is greater than the second, the routine will swap the two entries so that the entry that was initially the second entry would now be the first.

As the procedure continues, if the Nth entry is found to be greater than the N+1 entry, the two entries are exchanged in the table. Then, rather than advancing to the next entry, the routine backs up to compare the N—1 entry against the new N entry. This is because if the initial N+1 entry was lower than the N entry, it may also be lower than the N—1 entry. Therefore, the routine will

continue to transfer the lower entry down through the table until the entry before it is lower in alphabetical order, or the entry is moved to the beginning of the table once again. The routine then starts back up through the table once again. This type of movement up and down through the table gives a ripple effect as the routine compares and shifts the entries around.

The operation of the sort routine may be illustrated by examining its procedure for arranging the sample list of names into alphabetical order, given at the start of this section. The routine initially compares the first entries and finds the order to be correct, since the B in BROWN comes before the D in DALEY. When the next pair of entries is compared, however, it is found that ANDERSON should go before DALEY. The routine will therefore shift the second and third entries around, as illustrated in the table below.

BROWN, L.R.
ANDERSON, B.
DALEY, D.R.
DARBY, P.
MATTOX, R.T
MATTHEWS, K.D.
JONES, A.M.

Now the second and third entries are in the proper order with respect to each other. The routine backs up to compare the first entry against the second. The second entry is found to be less than the first. The routine swaps these two entries and begins comparing the entries once again, starting with the first two.

ANDERSON, B.
BROWN, L.R.
DALEY, D.R.
DARBY, P.
MATTOX, R.T.
MATTHEWS, K.D.
JONES, A.M.

On this pass through the table, the routine will proceed all the way up to MATTOX, R.T. before finding another entry out of order. Note that in comparing MATTOX, R.T. and MATTHEWS, K.D., the routine must work up to the fifth character in the last names to determine the proper order. If the last names were the

same, it must go up to the initials to check whether the two entries are in order. Upon finding these names in the wrong order, the routine will exchange them:

ANDERSON, B.
BROWN, L.R.
DALEY, D.R.
DARBY, P.
MATTHEWS, K.D.
MATTOX, R.T.
JONES, A.M.

The routine then backs up in order to compare DARBY, P. and MATTHEWS, K.D. Finding these to be in order, it moves forward again until MATTOX, R.T. is compared to JONES, A.M. These two entries are swapped and the routine again backs up and compares MATTHEWS, K.D. to JONES, A.M. It finds also they must be swapped. Finally, after determining that DARBY, P. and JONES, A.M. are in the right order, the routine advances until the end of the table is reached. The resulting table will contain the names in the following order:

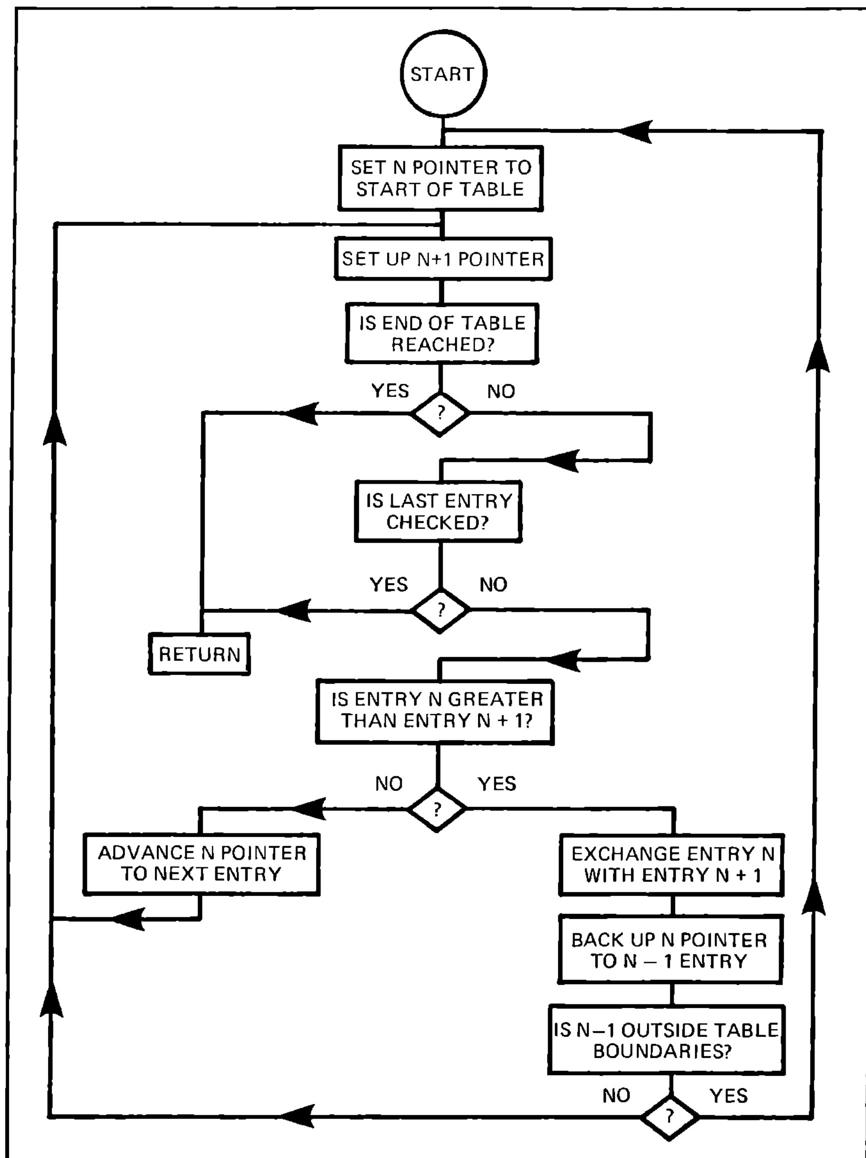
ANDERSON, B.
BROWN, L.R.
DALEY, D.R.
DARBY, P.
JONES, A.M.
MATTHEWS, K.D.
MATTOX, R.T.

In checking for the start and end of the allowable table, this routine assumes that the table begins at page $04 location $00, and ends at page $07 location $FF. If the entire table is not filled, the last entry must be followed by a zero byte. The instruction sequence used here to compare the two entries is similar to the CPRMEM routine presented in Chapter Three.

```
SORT     LDX #<SRTTB1    Initialize pointer to second table entry
         STX FMPNT       Store low address portion in FMPNT
         LDX #<SRTTB1    Set page portion of address
         STX FMPNT+1     Store in FMPNT
         LDX #<SRTTBL    Initialize pointer to start of sort table
```

```
           STX TOPNT        Store low address portion in TOPNT
           LDX #>SRTTBL     Set page portion of address
           STX TOPNT+1      Store in TOPNT
INITBK     LDA TOPNT+1      Fetch page portion of table pointer
           CMP #$07         Is last page of table indicated?
           BNE CKEND        No, check for last table entry
           LDA TOPNT        Yes, check low address portion
           CMP #$F0         Is the end of table reached?
           BEQ SRTRET       Yes, sort complete, return
CKEND      LDY #$10         Set index pointer
           LDA (TOPNT),Y    Does N+1 entry start with zero?
           BNE CKNEXT       No, compare two entries
SRTRET     RTS              Yes, end of table entries, return
CKNEXT     LDY #$0          Initialize index pointer
           LDA (TOPNT),Y    Fetch character from N entry
           CMP (FMPNT),Y    Compare N to N+1 entry
           BNE CKGTLT       Not equal, check for greater than
           INY              Equal, advance index pointer
           CPY #$10         All characters checked?
           BNE CKNEXT+$2    No, continue comparison
FINEND     CLC              Clear carry for addition
           LDA FMPNT        Fetch N+1 pointer
           ADC #$10         Advance it to next entry
           STA FMPNT        Restore in FMPNT
           BCC TOADV        Not across page, advance TOPNT
           INC FMPNT+1      Next page, increment page portion
TOADV      CLC              Clear carry for addition
           LDA TOPNT        Fetch pointer to N entry
           ADC #$10         Advance to next entry
           STA TOPNT        Restore in TOPNT
           BCC INITBK       Not across page, check end of table
           INC TOPNT+1      Advance page portion of TOPNT
           BNE INITBK       Jump to test for end of table
CKGTLT     BCC FINEND       N < N+1, advance to next entry
           LDY #$0          N > N+1, exchange entries
NOTYET     LDA (TOPNT),Y    Fetch character from N
           TAX              Save temporarily
           LDA (FMPNT),Y    Fetch character from N+1
           STA (TOPNT),Y    Store N+1 character in N
           TXA              Fetch N character
           STA (FMPNT),Y    Store N character in N+1
           INY              Advance index pointer
```

```
CPY #$10        Complete entries swapped?
BNE NOTYET      No, continue exchange
SEC             Set carry for subtraction
LDA FMPNT       Fetch N+1 pointer
SBC #$10        Backup to N entry
```



```
        STA FMPNT      Restore in FMPNT
        BCS TODEC      Page not crossed, decrement TOPNT
        DEC FMPNT+1    Page crossed, decrement page portion
TODEC   SEC            Set carry for subtraction
        LDA TOPNT      Fetch N pointer
        SBC #$10       Back up to N-1 entry
        STA TOPNT      Restore in TOPNT
        BCS INITBK     Page not crossed, compare next entries
        DEC TOPNT+1    Page crossed, decrement page portion
        LDA TOPNT+1    Fetch new N page pointer
        CMP #$03       Is pointer backed up too far?
        BNE INITBK     No, test next pair of entries
        BEQ SORT       Yes, reset pointers to start
```

Ways to Shorten Sort Operations

This method of sorting may be aided in a number of ways to increase the efficiency of its operation. For example, the name input routine could be revised to separate the table into several sections, one for names beginning with the letters A through J, another for K through R, and another for S through Z. As each name is entered, the first letter could be checked, and the name stored in the proper section of the table.

Another possibility is to revise the ripple sequence in the following manner. When a name is found to be out of alphabetical order, the start address of the current N+1 entry could be saved. Then, after the entry is backed up to the proper location in the table, the sort may resume by recalling the saved N+1 address and using it as the N address of the next entry to compare. This would avoid the time-consuming process of retracing the sort up through the section already known to be in the proper order.

The sort function also could be revised to limit itself to the contents of just one field in an entry. By setting the pointer and field length counter to a specific field within each entry, the sort operation could arrange the entries according to some classification such as the Zip code of an address, or a special code set up by the programmer to classify each entry.

The techniques and routines discussed in this chapter may be utilized to create rather sophisticated programs designed specifically to fill one's requirements. By combining these with other programming functions, one may develop programs that give the computer the capability to perform various operations for a wide variety of applications.

Appendices

Appendix A

This table presents the entire instruction set of the 6502 CPU. The first column contains the menmonic used for each instruction. The machine code, presented as hexadecimal digits, is given in the second column. For the instructions that use one or more of the possible addressing modes, the third column indicates the mode for the machine code of that row. The fourth and fifth columns indicate the number of bytes required for the instruction cycles and the number of machine cycles, respectively. Instructions are defined in Chapter 1. (* — add one if page boundary is crossed.)

Mnemonic	Machine Code	Addressing Code	No. of Bytes	No. of Cycles
ADC	69	Immediate	2	2
ADC	65	Zero Page	2	3
ADC	75	Zero Page, X	2	4
ADC	6D	Absolute	3	4
ADC	7D	Absolute,X	3	4*
ADC	79	Absolute,Y	3	4*
ADC	61	Indirect,X	2	6
ADC	71	Indirect,Y	2	5*
AND	29	Immediate	2	2
AND	25	Zero Page	2	3
AND	35	Zero Page	2	4
AND	2D	Absolute	3	4
AND	3D	Absolute,X	3	4*
AND	39	Absolute,Y	3	4*
AND	21	Indirect,X	2	6
AND	31	Indirect,Y	2	5*
ASL	0A	Accumulator	1	2
ASL	06	Zero Page	2	5
ASL	16	Zero Page,X	2	6
ASL	0E	Absolute	3	6
ASL	1E	Absolute,X	3	7
BCC	90	Relative	2	3*
BCS	B0	Relative	2	3*
BEQ	F0	Relative	2	3*

BIT	24	Zero Page	2	3
BIT	2C	Absolute	3	4
BMI	30	Relative	2	3*
BNE	D0	Relative	2	3*
BPL	10	Relative	2	3*
BRK	00		1	7
BVC	50	Relative	2	3*
BVS	70	Relative	2	3*
CLC	18		1	2
CLD	D8		1	2
CLI	58		1	2
CLV	B8		1	2
CMP	C9	Immediate	2	2
CMP	C5	Zero Page	2	3
CMP	D5	Zero Page,X	2	4
CMP	CD	Absolute	3	4
CMP	DD	Absolute,X	3	4*
CMP	D9	Absolute,Y	3	4*
CMP	C1	Indirect,X	2	6
CMP	D1	Indirect,Y	2	5*
CPX	E0	Immediate	2	2
CPX	E4	Zero Page	2	3
CPX	EC	Absolute	3	4
CPY	C0	Immediate	2	2
CPY	C4	Zero Page	2	3
CPY	CC	Absolute	3	4
DEC	C6	Zero Page	2	5
DEC	D6	Zero Page,X	2	6
DEC	CE	Absolute	3	6
DEC	DE	Absolute,X	3	7
DEX	CA		1	2
DEY	88		1	2
EOR	49	Immediate	2	2
EOR	45	Zero Page	2	3
EOR	55	Zero Page,X	2	4
EOR	4D	Absolute	3	4
EOR	5D	Absolute,X	3	4*
EOR	59	Absolute,Y	3	4
EOR	41	Indirect,X	2	6
EOR	51	Indirect,Y	2	5*
INC	E6	Zero Page	2	5
INC	F6	Zero Page,X	2	6

INC	EE	Absolute	3	6
INC	FE	Absolute,X	3	7
INX	E8		1	2
INY	C8		1	2
JMP	4C	Absolute	3	3
JMP	6C	Indirect	3	5
JSR	20	Absolute	3	6
LDA	A9	Immediate	2	2
LDA	A5	Zero Page	2	3
LDA	B5	Zero Page,X	2	4
LDA	AD	Absolute	3	4
LDA	BD	Absolute,X	3	4*
LDA	B9	Absolute,Y	3	4*
LDA	A1	Indirect,X	2	6
LDA	B1	Indirect,Y	2	5*
LDX	A2	Immediate	2	2
LDX	A6	Zero Page	2	3
LDX	B6	Zero Page,Y	2	4
LDX	AE	Absolute	3	4
LDX	BE	Absolute,Y	3	4*
LDY	A0	Immediate	2	2
LDY	A4	Zero Page	2	3
LDY	B4	Zero Page,X	2	4
LDY	AC	Absolute	3	4
LDY	BC	Absolute,X	3	4*
LSR	4A	Accumulator	1	2
LSR	46	Zero Page	2	5
LSR	56	Zero Page,X	2	6
LSR	4E	Absolute	3	6
LSR	5E	Absolute,X	3	7
NOP	EA		1	2
ORA	09	Immediate	2	2
ORA	05	Zero Page	2	3
ORA	15	Zero Page,X	2	4
ORA	0D	Absolute	3	4
ORA	1D	Absolute,X	3	4*
ORA	19	Absolute,Y	3	4*
ORA	01	Indirect,X	2	6
ORA	11	Indirect,Y	2	5*
PHA	48		1	3
PHP	08		1	3
PLA	68		1	4
PLP	28		1	4

ROL	2A	Accumulator	1	2
ROL	26	Zero Page	2	5
ROL	36	Zero Page,X	2	6
ROL	2E	Absolute	3	6
ROL	3E	Absolute,X	3	7
ROR	6A	Accumulator	1	2
ROR	66	Zero Page	2	5
ROR	76	Zero Page,X	2	6
ROR	6E	Absolute	3	6
ROR	7E	Absolute,X	3	7
RTI	40		1	6
RTS	60		1	6
SBC	E9	Immediate	2	2
SBC	E5	Zero Page	2	3
SBC	F5	Zero Page,X	2	4
SBC	ED	Absolute	3	4
SBC	FD	Absolute,X	3	4*
SBC	F9	Absolute,Y	3	4*
SBC	E1	Indirect,X	2	6
SBC	F1	Indirect,Y	2	5*
SEC	38		1	2
SED	F8		1	2
SEI	78		1	2
STA	85	Zero Page	2	3
STA	95	Zero Page,X	2	4
STA	8D	Absolute	3	4
STA	9D	Absolute,X	3	5
STA	99	Absolute,Y	3	5
STA	81	Indirect,X	2	6
STA	91	Indirect,Y	2	6
STX	86	Zero Page	2	3
STX	96	Zero Page,Y	2	4
STX	8E	Absolute	3	4
STY	84	Zero Page	2	3
STY	94	Zero Page,X	2	4
STY	8C	Absolute	3	4
TAX	AA		1	2
TAY	A8		1	2
TSX	BA		1	2
TXA	8A		1	2
TXS	9A		1	2
TYA	98		1	2

Appendix B

Octal to Hexadecimal

	0	1	2	3	4	5	6	7
00	0	1	2	3	4	5	6	7
01	8	9	A	B	C	D	E	F
02	10	11	12	13	14	15	16	17
03	18	19	1A	1B	1C	1D	1E	1F
04	20	21	22	23	24	25	26	27
05	28	29	2A	2B	2C	2D	2E	2F
06	30	31	32	33	34	35	36	37
07	38	39	3A	3B	3C	3D	3E	3F
10	40	41	42	43	44	45	46	47
11	48	49	4A	4B	4C	4D	4E	4F
12	50	51	52	53	54	55	56	57
13	58	59	5A	5B	5C	5D	5E	5F
14	60	61	62	63	64	65	66	67
15	68	69	6A	6B	6C	6D	6E	6F
16	70	71	72	73	74	75	76	77
17	78	79	7A	7B	7C	7D	7E	7F
20	80	81	82	83	84	85	86	87
21	88	89	8A	8B	8C	8D	8E	8F
22	90	91	92	93	94	95	96	97
23	98	99	9A	9B	9C	9D	9E	9F
24	A0	A1	A2	A3	A4	A5	A6	A7
25	A8	A9	AA	AB	AC	AD	AE	AF
26	B0	B1	B2	B3	B4	B5	B6	B7
27	B8	B9	BA	BB	BC	BD	BE	BF
30	C0	C1	C2	C3	C4	C5	C6	C7
31	C8	C9	CA	CB	CC	CD	CE	CF
32	D0	D1	D2	D3	D4	D5	D6	D7
33	D8	D9	DA	DB	DC	DD	DE	DF
34	E0	E1	E2	E3	E4	E5	D6	E7
35	E8	E9	EA	EB	EC	ED	EE	EF
36	F0	F1	F2	F3	F4	F5	F6	F7
37	F8	F9	FA	FB	FC	FD	FE	FF

Appendix C

Hexadecimal to Decimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
10	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
20	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
30	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
40	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
50	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
60	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Appendix D

ASCII Character Set

Characters Symbolized	Hexa Rep	Characters Symbolized	Hexa Rep
A	C1	!	A1
B	C2	"	A2
C	C3	#	A3
D	C4	$	A4
E	C5	%	A5
F	C6	&	A6
G	C7	'	A7
H	C8	(	A8
I	C9	)	A9
J	CA	*	AA
K	CB	+	AB
L	CC	,	AC
M	CD	-	AD
N	CE	.	AE
O	CF	/	AF
P	D0	0	B0
Q	D1	1	B1
R	D2	2	B2
S	D3	3	B3
T	D4	4	B4
U	D5	5	B5
V	D6	6	B6
W	D7	7	B7
X	D8	8	B8
Y	D9	9	B9
Z	DA	:	BA
[	DB	;	BB
\	DC	<	BC
]	DD	=	BD
↑	DE	>	BE
←	DF	?	BF
SPACE	A0	@	C0
CAR RET	8D	RUBOUT	FF
LINE FEED	8A	CONTROL O	8F

Appendix E

Baudot Character Set

Characters LC	Characters UC	5 Level Code Bit Position 5	4	3	2	1	Hexa Codes LC	Hexa Codes UC
A	-	0	0	0	1	1	03	23
B	?	1	1	0	0	1	19	39
C	:	0	1	1	1	0	0E	2E
D	$	0	1	0	0	1	09	29
E	3	0	0	0	0	1	01	21
F	!	0	1	1	0	1	0D	2D
G	&	1	1	0	1	0	1A	3A
H	#	1	0	1	0	0	14	34
I	8	0	0	1	1	0	06	26
J	'	0	1	0	1	1	0B	2B
K	(	0	1	1	1	1	0F	2F
L	)	1	0	0	1	0	12	32
M	.	1	1	1	0	0	1C	3C
N	,	0	1	1	0	0	0C	2C
O	9	1	1	0	0	0	18	38
P	0	1	0	1	1	0	16	36
Q	1	1	0	1	1	1	17	37
R	4	0	1	0	1	0	0A	2A
S	BELL	0	0	1	0	1	05	25
T	5	1	0	0	0	0	10	30
U	7	0	0	1	1	1	07	27
V	;	1	1	1	1	0	1E	3E
W	2	1	0	0	1	1	13	33
X	/	1	1	1	0	1	1D	3D
Y	6	1	0	1	0	1	15	35
Z	"	1	0	0	0	1	11	31
SPACE		0	0	1	0	0	04	04
CAR RET		0	1	0	0	0	08	08
LINE FEED		0	0	0	1	0	02	02
NULL		0	0	0	0	0	00	00
FIGURES		1	1	0	1	1	1B	1B
LETTERS		1	1	1	1	1	1F	1F

Appendix F

Floating Point Program

The floating point program presented in Chapter Five has been assembled and is presented below as a memory dump. The left-hand column contains the location of the first memory byte on that line. Each row of data indicates the contents of 16 memory locations. The symbol table that immediately follows this memory dump indicates the location of the instruction referenced by that symbol within this dump.

The program is split into two parts. The first half contains the floating point routines. The second contains the input and output routines. If one desires to use only the floating point arithmetic routines, the second half may be deleted, beginning at $475.

```
0200   A9 00 A8 91 02 C8 CA D0 FA 60 18 36 00 88 D0 01
0210   60 E8 4C 0B 02 18 76 00 88 D0 01 60 CA 4C 16 02
0220   38 A9 FF 55 00 69 00 95 00 E8 88 D0 F4 60 18 A0
0230   00 B1 02 71 00 91 02 C8 CA D0 F6 60 38 A0 00 B1
0240   02 F1 00 91 02 C8 CA D0 F6 60 A0 00 B1 00 91 02
0250   C8 CA D0 F8 60 A2 05 A5 0A 30 07 A0 00 94 00 4C
0260   6B 02 95 00 A0 04 A2 07 20 20 02 A2 0A A0 04 B5
0270   00 D0 07 CA 88 D0 F8 84 0B 60 A2 07 A0 04 20 0A
0280   02 B5 00 30 05 C6 0B 4C 7A 02 A2 0A A0 03 20 15
0290   02 A5 05 F0 E4 A0 03 4C 20 02 A5 0A D0 13 A2 10
02A0   86 00 A2 08 86 02 A9 00 85 01 85 03 A2 04 4C 4A
02B0   02 A5 12 D0 01 60 A2 0B B5 00 C5 13 F0 37 38 A9
02C0   00 F5 00 65 13 10 07 38 85 2C A9 00 E5 2C C9 18
02D0   30 08 38 A5 13 F5 00 10 C5 60 A5 13 38 F5 00 A8
02E0   30 0B A2 0B 20 15 03 88 D0 F8 4C F5 02 A2 13 20
02F0   15 03 C8 D0 F8 A9 00 85 07 85 0F A2 0B 20 15 03
0300   A2 13 20 15 03 A2 0F 86 00 A2 07 86 02 A2 04 20
0310   2E 02 4C 55 02 F6 00 CA 98 A0 04 48 B5 00 30 06
0320   20 15 02 4C 2A 03 38 20 16 02 68 A8 60 A2 08 A0
```

0330	03	20	20	02	4C	9A	02	20	A5	03	A5	13	18	65	0B	85
0340	0B	E6	0B	A9	17	85	04	A2	0A	A0	03	20	15	02	90	0D
0350	A2	0D	86	00	A2	15	86	02	A2	06	20	2E	02	A2	1A	A0
0360	06	20	15	02	C6	04	D0	DF	A2	1A	A0	06	20	15	02	A6
0370	17	B5	00	2A	10	13	18	A0	03	A9	40	75	00	85	17	A9
0380	00	75	00	95	00	E8	88	D0	F6	A2	07	86	02	A2	17	86
0390	00	A2	04	20	4A	02	20	55	02	A5	06	D0	07	A2	08	A0
03A0	03	20	20	02	60	A9	00	85	03	85	01	A9	14	85	02	A2
03B0	08	20	00	02	A9	0C	85	02	A2	04	20	00	02	A9	01	85
03C0	06	A5	0A	10	09	C6	06	A2	08	A0	03	20	20	02	A5	12
03D0	30	01	60	C6	06	A2	10	A0	03	4C	20	02	20	A5	03	A5
03E0	0A	F0	23	A5	13	38	E5	0B	85	0B	E6	0B	A9	17	85	04
03F0	20	4E	04	30	16	A2	10	86	02	A2	14	86	00	A2	03	20
0400	4A	02	38	4C	0C	04	A9	BF	4C	06	04	18	A2	18	A0	03
0410	20	0B	02	A2	10	A0	03	20	0A	02	C6	04	D0	D2	20	4E
0420	04	30	1E	A9	01	18	65	18	85	18	A9	00	65	19	85	19
0430	A9	00	65	1A	85	1A	10	09	A2	17	A0	03	20	15	02	E6
0440	0B	A2	07	86	02	A2	17	86	00	A2	04	4C	93	03	A2	14
0450	86	02	A2	08	86	00	A2	03	20	4A	02	A2	14	86	02	A2
0460	10	86	00	A0	00	A2	03	38	B1	00	F1	02	91	02	C8	CA
0470	D0	F6	A5	16	60	A9	00	85	01	85	03	D8	A2	1C	86	02
0480	A2	0C	20	00	02	20	80	07	C9	AB	F0	06	C9	AD	D0	08
0490	85	1C	20	C5	07	20	80	07	C9	8F	D0	0B	A9	BC	20	C5
04A0	07	20	C0	07	4C	75	04	C9	AE	D0	12	24	1E	10	02	30
04B0	2C	85	1E	A0	00	84	04	20	C5	07	4C	95	04	C9	C5	D0
04C0	42	20	C5	07	20	80	07	C9	AB	F0	06	C9	AD	D0	08	85
04D0	1D	20	C5	07	20	80	07	C9	8F	F0	C1	C9	B0	30	56	C9
04E0	BA	10	52	29	0F	85	2C	A2	27	A9	03	D5	00	30	46	B5
04F0	00	18	36	00	36	00	75	00	2A	65	2C	95	00	A9	B0	05
0500	2C	D0	CE	C9	B0	30	2E	C9	BA	10	2A	A8	A9	F8	24	25
0510	D0	83	98	20	C5	07	E6	04	29	0F	48	20	AB	05	A2	23
0520	68	18	75	00	95	00	A9	00	75	01	95	01	A9	00	75	02
0530	95	02	4C	95	04	A5	1C	F0	07	A2	23	A0	03	20	20	02
0540	A9	00	85	22	A9	07	85	02	A9	22	85	00	A2	04	20	4A
0550	02	A0	17	84	0B	20	55	02	A5	1D	F0	08	A9	FF	45	27
0560	85	27	E6	27	A5	1E	F0	05	A9	00	38	E5	04	18	65	27
0570	85	27	30	1D	D0	01	60	20	7D	05	D0	FB	60	A9	04	85
0580	13	A9	50	85	12	A9	00	85	11	85	10	20	37	03	C6	27
0590	60	20	97	05	D0	FB	60	A9	FD	85	13	A9	66	85	12	85
05A0	11	A9	67	85	10	20	37	03	E6	27	60	A9	00	85	26	A2
05B0	1F	86	02	A2	23	86	00	A2	04	20	4A	02	A2	23	A0	04
05C0	20	0A	02	A2	23	A0	04	20	0A	02	A2	1F	86	00	A2	23

```
05D0  86 02 A2 04 20 2E 02 A2 23 A0 04 4C 0A 02 A9 00
05E0  85 27 A5 0A 30 04 A9 AB D0 09 A2 08 A0 03 20 20
05F0  02 A9 AD 20 C5 07 A9 B0 20 C5 07 A9 AE 20 C5 07
0600  C6 0B 10 0F A9 04 18 65 0B 10 0E 20 7D 05 A5 0B
0610  4C 02 06 20 97 05 4C 0E 06 A2 23 86 02 A2 08 86
0620  00 A2 03 20 4A 02 A9 00 85 26 A2 23 A0 03 20 0A
0630  02 20 AB 05 E6 0B F0 0A A2 26 A0 04 20 15 02 4C
0640  34 06 A9 07 85 04 A5 26 F0 11 A5 26 09 B0 20 C5
0650  07 C6 04 F0 1A 20 AB 05 4C 4A 06 C6 27 A5 25 D0
0660  F0 A5 24 D0 EC A5 23 D0 E8 A9 00 85 27 F0 E2 A9
0670  C5 20 C5 07 A5 27 30 05 A9 AB 4C 85 06 49 FF 85
0680  27 E6 27 A9 AD 20 C5 07 A0 00 A5 27 38 E9 0A 30
0690  06 85 27 C8 4C 8C 06 98 09 B0 20 C5 07 A5 27 09
06A0  B0 4C C5 07 A9 8D 20 C5 07 A9 8A 20 C5 07 20 75
06B0  04 20 C0 07 A2 28 86 02 A2 08 86 00 A2 04 20 4A
06C0  02 20 80 07 C9 AB D0 09 20 01 07 20 9A 02 4C F5
06D0  06 C9 AD D0 09 20 01 07 20 2D 03 4C F5 06 C9 D8
06E0  D0 09 20 01 07 20 37 03 4C F5 06 C9 AF D0 0C 20
06F0  01 07 20 DC 03 20 DE 05 4C A4 06 C9 8F D0 C2 F0
0700  A3 20 C5 07 20 C0 07 20 75 04 20 C0 07 A9 BD 20
0710  C5 07 20 C0 07 A2 10 86 02 A2 28 86 00 A2 04 4C
0720  4A 02
```

Label	Address	Label	Address	Label	Address
ACCMIN	0262	COMPL	0203	EXOUTN	067D
ACCSET	028A	COMPLM	0220	EXPFIX	0577
ACNONZ	027A	CROUND	037F	EXPINP	0404
ACZERT	026B	DECBIN	05AB	EXPOK	056D
ADDER	022E	DECEXD	0613	EXPOUT	066F
ADDEXP	033A	DECEXT	0602	FINAL	06F5
ADOPPP	0350	DECOUT	0619	FINPUT	0540
ADDR1	0231	DECRDG	0651	FMPNT	0000
AHEAD1	05F3	DECREP	060E	FNDEXP	04C1
AHEAD2	0685	DERROR	0406	FOLSWE	000F
BRING1	0326	DIVIDE	03F0	FOPEXP	0013
CKEQEX	02B6	DVEXIT	0441	FOPLSW	0010
CKSIGN	03A5	ECHO	07C5	FOPMSW	0012
CLRM1	0203	ENDINP	0535	FOPNSW	0011
CLRMEM	0200	ERASE	049C	FPACCE	0008
CNTR	0004	EXECHO	0401	FDADD	029A
COMPEN	0634	EXMLDV	0393	FPCONT	06A4

Symbol	Address
FPD10	0597
FPDIV	03DC
FPINP	0475
FPLSW	0008
FPLSWE	0007
FPMSW	000A
FPMULT	0337
FPNORM	0255
FPNSW	0009
FPOUT	05DE
FPSUB	032D
FPX10	057D
FSHIFT	031B
INEXPS	001D
INMTAS	001C
INPRDI	001E
INPUT	0780
IOEXP	0022
IOEXPD	0027
IOLSW	001F
IOMSW	0021
IONSW	0020
IOSTR	0023
IOSTR1	0024
IOSTR2	0025
IOSTR3	0026
ISLAND	04DD
LINEUP	02DA
LOOK0	026F
MCAND0	000C
MCAND1	000D
MCAND2	000E
MINEXP	0591
MORACC	02E2
MORRTL	0211
MORRTR	021C
MOVIN1	024C
MOVIND	024A
MOVOP	029E
MULTEX	03A4
MULTIP	0347
NADOPP	0350
NEGFPA	03C5
NEGOP	0303
NINPUT	0495
NOEXPS	0407
NOGO	040B
NONZAC	02B1
NORMEX	0279
NOTADD	06D1
NOTDIV	06FB
NOTMUL	06EB
NOTPLM	0498
NOTSUB	06DE
NVALID	06C1
OPERAT	0701
OPSGNT	03CE
OUTDGS	064A
OUTDIG	0642
OUTNEG	05EA
PER1	04B1
PERIOD	04AB
POSEXP	0564
PREXFR	0389
QUOROT	040C
RESCNT	032A
ROTATL	020A
ROTATR	0215
ROTL	020B
ROTR	0216
SECHO	0492
SERASE	04A7
SETDCT	03EC
SETMCT	0343
SETSUB	044E
SFNDXP	0503
SHACOP	02F5
SHIFTO	02ED
SHLOOP	0315
SIGNS	0006
SKPNEG	02CE
SPACES	07C0
SPRIOD	04BD
SUB12	068C
SUBB1	023F
SUBBER	023C
SUBEXP	03E3
SUBR1	0468
TEMP1	002C
TOMUCH	0697
TOPNT	0002
TPEXP	002B
TPLSW	0028
TPMSW	002A
TPNSW	0029
TSIGN	0005
WORK0	0014
WORK1	0015
WORK2	0016
WORK3	0017
WORK4	0018
WORK5	0019
WORK6	001A
WORK7	001B
ZERODG	065B

Index

Accumulator: 9, 10
Addressing modes: 16—22
ASCII: 71, 72
Asynchronous: 143, 149
BAUD: 144, 150
BAUDOT: 71, 73
BCD: 127
Binary, exponent: 92
  mantissa: 92
  point: 92
  to-decimal conversion: 87
Bit, start: 143, 150
  stop: 143, 150
Borrow: 57
Character code: 72
Complement, One's: 56
  Two's: 56
Conditional branch: 46
Conversion, binary-to-decimal: 87
  code: 71
  decimal-to-binary: 85
  numeric: 84
Decimal, addition: 126
  division: 135
  mode flag: 12, 54, 125
  subtraction: 126
  to-binary conversion: 85
Delay time: 150
Dividend: 106
Division: 106
EBCDIC: 71
Encode: 74
Error detection: 68
Execution time: 61
Field: 170, 181
Fixed format: 170
Flags, break: 12
  carry: 11, 53
  condition: 12
  decimal mode: 12
  in-progress: 156
  interrupt disable: 12
  negative: 11
  overflow: 12, 57
  status: 9, 11
  zero: 12
Floating point, accumulator: 93
  addition: 96
  format: 91
  input routine: 113
  multiplication: 101
  operand: 93
  output routine: 118
  subtraction: 100
Free format: 170
Handshaking: 146
HOLLERITH: 72
Interrupt: 14
  maskable: 15, 153
  nonmaskable: 14, 153
  processing: 153
  service routine: 152, 158, 162
  software: 15
  vector: 15
I/O driver: 139
Limits, checking: 60
Look-up table: 77
Mantissa: 91
Memory: 9, 10
  access: 61
  clearing: 50
  transferring: 51
Multiple precision: 91
  addition: 57
  comparison: 58
  decrementing: 54
  rotating: 55
  routine: 52
Multiplicand: 101
Multiplier: 101
Nesting: 154, 164
Parallel data: 140
Parity: 68
Partial-product: 101
PDT operation: 146
Pipelining: 62
Pointer: 80
  stack: 9, 10, 13
Polling: 164
Program counter: 9
Programmed delay: 61, 144
Quotient: 106
RAM, dynamic: 62
  static: 62
Random number generator: 66
Register, countdown: 46
  index: 9, 10
  status: 13
Reset: 15
ROM: 62
Routine, input: 113
  multiple precision: 52
  multiplication: 133
  output: 118
  service: 152, 158, 162
Serial data: 143, 149
Signed, addition: 128
  subtraction: 131
Sort: 181
  ripple: 186
Status byte: 147
Strobe: 148
Subroutine: 49
Timing diagram: 143
Transmission: 143
True logic: 140

Now you can put together programs without having to start from scratch. You'll have the most useful routines at your command — already programmed and ready to use. You'll get a plain-talk explanation of how the entire 6502 instruction set works. And that's a big value to everyone, 6502 owner or not! All in one easy-to-use cookbook.

Why is it called a cookbook?

Because it's a book of recipes. It contains routines, subroutines and short programs. These are the ingredients. All you do is take a pinch of this, a pinch of that. Combine the ingredients, and voila — your own masterpiece! Just the program to suit your taste.

Time-tested recipes.

Although the 6502 cookbook is brand new, SCELBI's software cookbook idea has been around for years. The recipes are really time-tested! Tens of thousands of our Z80, 6800 and 8080 cookbooks have been used throughout the U.S. and in countries around the world.