PRACTICAL MICROCOMPUTER PROGRAMMING:

```
0300                    *
0300                    * CONVERT BINARY TO ASCII-HEX
0300                    *
0300                    CNV2HX  TAX             SAVE INCOMING NUMBER
0300 AA                         RORA            ROTATE HIGH DIGIT
0301 6A                         RORA            INTO POSITION
0302 6A                         RORA            ...
0303 6A                         RORA            ...
0304 6A                         JSR   CVHXDG    CONVERT HIGH DIGIT
0305 201003                     PHA             SAVE ASCII RESULT
0308 48                         TXA             RETRIEVE BINARY NUMBER
0309 8A                         JSR   CVHXDG    CONVERT LOW DIGIT
030A 201003                     TAX             LOW RESULT TO X REGISTER
030D AA                         PLA             RETRIEVE HIGH RESULT
030E 68                         RTS             AND RETURN
030F 60                 *
0310                    * CONVERT LOW FOUR BITS OF A TO AN ASCII-HEX DIGIT
0310                    *
0310                    CVHXDG  AND   #$F        CLEAR UPPER FOUR BITS
0310 290F                       CMP   #$A        GREATER THAN 9?
0312 C90A                       BLS   CVHX2      NO
0314 9003                       ADD   #7         YES, BIAS UP
0316 186907             CVHX2   ADD   #'0       ADD ASCII ZERO
0319 1869B0                     AND   #$B0       CLEAR LOWER FOUR BITS
```

THE 6502

A User's Guide to Application Programming for the 6502

W. J. WELLER

$32.95

PRACTICAL MICROCOMPUTER PROGRAMMING: THE 6502

W.J. WELLER

In 20 chapters and 6 appendices this fourth volume of the PRACTICAL MICROCOMPUTER PROGRAMMING™ series covers the detailed assembly level programming characteristics of the 6502 microprocessor to a depth never before available for this machine. This is not a rehash of manufacturer's data sheets but a fully developed course on assembly language programming for the 6502.

All fundamental areas of assembly language technique are covered, the text explanation being supported by 119 formal programming examples.

Software support is not neglected either. The book contains the full source text of a conversational editor/assembler which will run directly on the Apple II, but can be made to run on any 6502 based computer. A full debug program for the Apple is also included which has such features as self restoring breakpoints, memory search, memory fill, memory windowing and much more.

PRACTICAL MICROCOMPUTER PROGRAMMING is a trademark of Northern Technology Books.

PRACTICAL MICROCOMPUTER PROGRAMMING:

THE 6502

W.J. WELLER

Northern Technology Books

Weller, Walter J.
Practical Microcomputer Programming: The 6502

Library of Congress Catalog Card #80-83327
International Standard Book Number: 0-930594-08-8

Copyright © Walter J. Weller 1980

All rights reserved. No part of this publication may be reproduced, copied or transmitted in any form or by any mechanical or electronic means without written permission of the copyright holder.

To the 24 years...

TABLE OF CONTENTS

PREFACE

CHAPTER 1 STRUCTURE OF THE 6502 COMPUTER 1

Logical elements of a computer. The function of memory. Volatility of memory. The word and wordlength. Address concept. RAM and ROM. The A register. The control unit. I and P registers. Concept of a pointer. The computer program. Fundamental instruction execution cycle. Classes of computer instructions.

CHAPTER 2 THE BINARY SYSTEM AND BINARY OPERATIONS 7

Number bases. The binary system. Logical operations. Inclusive OR. The AND function. Masking. Complementation. Exclusive OR. Powers of two. Binary addition. Binary subtraction. Twos complement arithmetic. Carry, sign and overflow bits. Logical shifts. Rotary shifts. Arithmetic shifts. Octal and hexadecimal.

CHAPTER 3 MACHINE AND ASSEMBLY LANGUAGES 27

Binary instructions and programming. Mnemonic form of an instruction. Mnemonic loader. Assembly language. The assembly program. Definition of symbol. Label, instruction and operand fields. Comment field. The source program. The assembly listing. The symbol table. Error listing. The object program.

TABLE OF CONTENTS

CHAPTER 4 USING THE ASSEMBLY PROGRAM 36

Allocating source buffer memory. System initialization. The A command. The L and D commands. Writing and reading source. Error scan function. Switching peripherals. Assembler error messages. Standard I/O device assignments.

CHAPTER 5 MEMORY REFERENCING ON THE 6502 46

Addressing modes. Scratchpad memory. General memory. Direct addressing. Immediate mode. Indexed addressing. The effective address. Index registers. Indirect addressing. Pre and post indexing.

CHAPTER 6 6502 ARITHMETIC AND LOGICAL OPERATIONS 54

Addition with and without carry. Subtraction with and without borrow. Minuend, subtrahend and difference. The processor status word. Meanings of PSW bits. Jump and branch instructions. Conditional branches. Branch synonyms. Logical operations. Data types. Looping. Block reservation. Half symbol constants. The CDB pseudo-op. Memory increment and decrement. Special pseudo-ops. Shift operations.

CHAPTER 7 DOUBLE AND MULTIPLE PRECISION METHODS 83

Definition of multiple precision. Double precision addition. Double precision subtraction. Triple precision subtraction. N precision addition. Mixed precisions. The IDB operation. Multiple precision shifts. More

assembler pseudo-ops.

CHAPTER 8 SOFTWARE MULTIPLICATION AND DIVISION 104

Multiplication by shift and add. Multiplication by special constants. The product of eight bit magnitudes. Multiplication of signed numbers. Double precision multiplication. Divisor, dividend, quotient and remainder. Binary division. The divide fault condition. Multiple precision division.

CHAPTER 9 SUBROUTINES AND THE STACK POINTER 122

The stack pointer. Pushing and pulling stack data. Loading the stack pointer. PSW bit locations. Subroutines. A double multiply subroutine. A validity check routine. Use of the stack in computed jumps. Argument transfer.

CHAPTER 10 ADDRESSING ARRAYS AND TABLES 144

Definition of an array. Tables. Data generating pseudo-ops. Sum of an array. Inverting an array in memory. Sorting. The "bubble" sort. Simple table lookup. Special table lookup cases. The displacement. Self modifying code.

CHAPTER 11 PROGRAMMED INPUT/OUTPUT 162

Device address. Ready flags. Clearing the ready flag. Symbolic I/O addresses. Peripheral status word. A parallel printer driver. Memory mapped I/O. The Apple screen addressing scheme. Scrolling the screen. Table of ASCII

character codes.

CHAPTER 12 6502 DECIMAL ARITHMETIC 180

Use of SED and CLD. BCD addition. Double precision BCD addition. A decimal mode "trick". BCD subtraction. Double precision BCD subtraction. Tens complement arithmetic. Considerations in BCD multiplication and division.

CHAPTER 13 FLOATING POINT ARITHMETIC 188

Components of a floating point number. Floating point addition. The guard digit. Loss of precision due to pre-shift. Normalization of floating point numbers. Variability of result with order of addition. Binary floating point. A floating addition for the 6502. "Noisy mode" arithmetic. Range of floating point numbers. Alternate floating point forms.

CHAPTER 14 NUMBER BASE CONVERSIONS 205

Conversion of BCD integer to binary. Conversion by successive multiplication by ten. Decimal validity check. Conversion to double precision integer. Decimal fraction to binary. Conversion of signed numbers. Binary integer to BCD. Binary fraction to BCD fraction. Radix deflation method for conversion of floating point numbers.

CHAPTER 15 GENERATION AND USE OF RANDOM NUMBERS 223

Fundamental properties of pseudorandom numbers. Determination of the value of

pi by random number methods. Monte Carlo methods. Power residue method for random number generation. Generator period. A 6502 random number generator. Frequency distribution for the generator. Random fractions. Generation of random BCD digits. A 6502 program to find the value of pi.An application of random number methods to data processing terminology.

CHAPTER 16 SIMPLE GRAPHIC OUTPUT 237

Construction of graph in memory. The image region. The Apple high resolution screen addressing scheme. Computation of screen addresses. Placing the point within the word. Some considerations in fast graphics. Inversion of the Apple screen line system. Erasure of plotted points. Copying the screen to a graphics printer. An application of the graphics scheme to an unusual alphabet.

CHAPTER 17 ELEMENTARY CRYPTOGRAPHY 259

Codes and ciphers. Encipherment and decipherment. Plaintext and ciphertext. The substitution cipher. Monoalphabetic ciphers. Historical use of the monoalphabetic cipher. A 6502 program for monoalphabetic encipherment. Letter frequencies in English text. A 6502 program to find ciphertext frequencies. The polyalphabetic cipher. A program for polyalphabetic encipherment for the 6502. A polyalphabetic decipherment program. Importance of ciphers in recent history.

TABLE OF CONTENTS

CHAPTER 18 INTERRUPT DRIVEN PROCESSES 273

Concept of an interrupt. Enablement and disablement of the interrupt system. Arming and disarming of individual devices. Necessary conditions for interrupt to occur. The interrupt service routine and its obligations. A real time clock interrupt. A 6502 software elapsed time clock. Problems with interrupt during disk transfer. Input under interrupt control. Arming considerations of the output interrupt. Output under interrupt control. Concept of a real time executive program.

CHAPTER 19 PROGRAM DEBUGGING 291

Generation of the object program. Using debug to load the object program. Examination and modification of memory. Setting the jump screen. Dumping memory blocks to the printer. Setting breakpoints. Modifying debug's registers. A sample program containing an error. Tracing the error with debug. Binary patching. Limits on breakpoints. Memory windowing. Window types.

CHAPTER 20 STRUCTURE OF THE ASSEMBLY PROGRAM 313

Purpose of the assembler. Design considerations in the text editor. Consideration of assembly language notation and form. The necessity for three passes. The location counter. Some error considerations. Two symbol definition passes. Final assembly pass. Internal operation of the text editor.

TABLE OF CONTENTS

APPENDIX A	ASSEMBLY LISTING OF THE ASSEMBLY PROGRAM	323
APPENDIX B	ASSEMBLY LISTING OF APPLE II INPUT/OUTPUT SUBROUTINES FOR THE ASSEMBLY PROGRAM	389
APPENDIX C	USING THE ASSEMBLY PROGRAM WITH OTHER 6502 BASED COMPUTERS	403
APPENDIX D	ASSEMBLY LISTING OF THE DEBUG PROGRAM	411
APPENDIX E	SUMMARY OF ASSEMBLER OPERATIONS AND PSEUDO-OPS	439
APPENDIX F	SUMMARY OF DEBUG PROGRAM OPERATIONS	449
INDEX		457

PREFACE

In this, the fourth of the **PRACTICAL MICROCOMPUTER PROGRAMMING** series, the detailed assembly level programming characteristics of the 6502 microprocessor are examined. The purpose of the book is to provide a means by which the 6502 user can, either by self instruction or formal classroom discipline, acquire the skills necessary to program the machine at assembly level.

Though the above stated purpose is congruent with the purpose of any textbook, a special situation exists with respect to assembly level instruction for the 6502 and some other microprocessors. Computer programming, like any skill, is acquired by experience, by doing. This experience may be guided by a book or teacher but the ultimate result in terms of the skill of the student depends most heavily on actually performing meaningful tasks with the machine. In the case of computer programming doing means actually running programs, verifying that they produce the required result, and understanding why this result is produced. This implies the existence of a language processor to support the language used in the textbook.

Though the 6502 is one of the oldest extant microprocessors, it is the remarkable fact that no generally available resident assembly system of a quality suitable for instruction has appeared for it. This is all the more remarkable in view of the large numbers of these devices which have been used as the base of machines like the Apple.

In the absence of a suitable language processor, then, a textbook would seem to be only a paper exercise which, if not devoid of worth, is of only theoretical value so far as providing a valid instruction vehicle is concerned. It would seem that a language processor must be part of any attempt at creating a teaching vehicle. To meet this obvious need, therefore, an editor/assembler system has been created specially for this book. The direct reason for the existence of this

programming system is to provide a means by which the examples in the book can be assembled, loaded into the user's 6502 system, and executed. Because of this primary purpose the greatest single emphasis in the design of the programming system has been simplicity of operation. The entire operation of the system is controlled by only nine primary keyboard commands. Since the text editor is to manipulate assembly language text only, rather than general text, it is the simplest line oriented editor which could be devised. The design emphasis in the assembler itself has also been simplicity. While the system will accept all 6502 machine mnemonics as well as a number of new compound and pseudo-operations, all line formatting is automatic and blank is used as the most natural field delimiter rather than semicolon or other special character sometimes used.

Where the requirements of a general purpose assembler do not conflict with the primary educational purpose of the system all reasonable efforts have been expended to make the system serviceable for these needs, but never at the expense of simplicity of operation. There may be users who wish to modify the system by adding special purpose functions or features. The needs of these users are recognized and to allow these modifications and additions to be made the entire source text of the editor/assembler is included in an appendix. The object code for the system is supplied to the book purchaser without charge on either Apple II cassette or paper tape when the licensing agreement at the back of the book is returned to the publisher. Apple disk is available for a moderate surcharge.

A large part of current assembly and machine level activity on the 6502 is now performed using mnemonic loaders, sometimes called "mini-assemblers", and mnemonic disassemblers. While these methods can be made to produce results, they are terribly error prone and properly belong to a past and more primitive era. Many thousands of potentially productive man-hours are wasted in this way, and it is to the elimination of

this waste that part of the thrust of this effort has been directed. The existence of a cheap, reliable assembly programming system for the 6502 will do much to allow the full usefulness of this device to be exploited, and the state of the programming craft will therefore be advanced. Economic forces have, in the past, worked against the availability of this kind of utility software, the need to recover the great cost of software development dictating that software be kept secret. Those competitive forces which usually work to enhance technological progress have the effect of retarding it in an arena such as software where it is difficult or impossible to protect the product from duplication.

Situations like this are not unknown in previous history. The Medici of renaissance Italy, for example, discovered the obstetrical forceps and kept it a family secret for many years, resulting in a handsome profit for the Medici. Progress can hardly be said to have been served in this case, many thousands of women dying in agony while the means for their preservation existed only as a trade secret. The corresponding dilemma with respect to software must be resolved in some way if we are to enjoy the benefits of computer technology with any timeliness.

With regard to the 6502 itself, the device has been called the "overlooked" and "neglected" microprocessor in some circles, and it is worthwhile to examine some of the reasons for this oversight and neglect. Whether correctly or not, the 6502 has come to be viewed as more difficult to program at assembly level than other contemporary microprocessors, which has resulted in a near vacuum in the literature for this machine. The current literature for the 6502 consists largely of rehashes of the manufacturer's manual which explain what the instructions do rather than how to put the instructions together to perform useful work.

This condition is partly the result of the extreme terseness and brevity of the 6502 instruction set which

makes it necessary to write two, three, or four mnemonic instructions to accomplish what can be done on other machines, e.g., the 8080, with a single instruction. The sheer tedium of writing the resulting volume of instructions has caused the software fraternity to turn its efforts elsewhere. This problem of verbosity can be greatly reduced by a thoughtfully constructed programming system which reduces common operations such as pointer creation to a single written line. Creation of such a programming system, however, is in itself a tedious and prosaic task and it has in consequence not been done to date. The problem is a circular one. The machine is too tedious to be properly programmed without the programming system, and the programming system is not created because of the sheer tedium of creating it. We of the software fraternity find ourselves in a situation which parallels that of the Antarcic penguins who mill and jostle around a hole in the ice waiting for one of their number to fall into the water. If the unfortunate one reappears the rest know that no walrus is lurking below, and it is safe to dive in and fish. The programming system presented here represents a voluntary "dive into the hole in the ice".

The verbosity problem of the 6502 is largely due to a few characteristics of its instructions set, i.e., that carry must be cleared before addition and set before subtraction, that it requires four instructions to create an address pointer and three to increment it. Further objections to the instruction set have been that there is no single instruction means for incrementing or decrementing the accumulator or performing ones or twos complementation on it. These objections have all been dealt with in the programming system presented here. All of the above objections have been met by providing pseudo-operations which perform the indicated functions with a single written line.

Yet another body of criticism of the machine comes from a group who would not be happy with anything.

Their attitude was best described in a remark by the sometimes populist philosopher Leo Durocher, who is alleged to have said: "There is a kind of man who could sit on a keg of rum in a harem and still complain".

It is not the writer's intention that this book be a stand alone source of programming information for the 6502. Like any book it is limited in scope and there are topics which are omitted or treated with less emphasis than might be desireable. This is the result of the finite size of books. If the book and its attendant programming system allow beginning assembly level programmers to get far enough into the subject to begin to feel comfortable with it its purpose will have been well served. It is intended that the book be used in conjunction with the 6502 machine manual.

The entire manuscript of the book has been prepared using the WordStar word processing system from MicroPro International of San Rafael, California. This system has contributed greatly to accuracy by eliminating the manual typesetting process entirely, thereby omitting a major source of error in the rekeying of the entire manuscript. WordStar represents a very great advance in both efficiency and convenience. It is a spendid job, one of the best software efforts this writer has ever seen, and its creators are to be commended.

Each of the example programs in the book has been re-entered directly from the final page proofs as insurance against error. However careful the writer and however good the word processing system, though, mistakes do slip through undetected in every book. The writer would be greatly indebted to any reader who would notify him of errors or oversights in either the text or software so that these may be corrected in later printings.

A number of individuals and organizations have contributed to the preparation of this book, either by supplying materials, or by discussion and consultation. They are, alphabetically:

Apple Computer, Inc.
Mr. Marvin Clavey
Mr. Charles Faso
Mr. Lou Haehn
Mr. Howard Herbin
Mr. Harvey Millman
Mr. Harvey Nice
Mr. William Powers
Mr. Phil Roybal
Mr. Harold Scoblow
Mr. Ted Singer
Mr. Gene Sullivan
Victor Comptometer Corporation, Components Div.
Mr. Karl Weller
Mrs. Ruth Weller

Chicago, Illinois
March 1980

Apple and Apple II are trademarks of Apple Computer, Inc. of Cupertino California.

CHAPTER 1

STRUCTURE OF THE 6502 COMPUTER

The purpose of this first chapter is to present the logical structure of the 6502 from the programmer's point of view, as distinct from that of the hardware designer. This will involve viewing some computer elements as logically separate which are not in fact physically separate.

The 6502 and other computers consist of three logical elements: a memory, an arithmetic and logic unit, and a control element. We will describe these elements in turn. **Memory** is a device for holding information. It comes in many physical forms but has the common property that information contained in it can be repeatedly retrieved, i.e., read, without destroying memory contents. Further, placing information into memory destroys the former contents. It is the exact analog of erasing a blackboard to write new information on it.

Memory is said to be **volatile** if its contents are lost when power is removed and **nonvolatile** if its contents are retained. Memory devices like magnetic cores which rely on ferromagnetism to hold information are in general nonvolatile. Semiconductor memory with battery or other power backup is also nonvolatile. Memory is organized into groups of digits known as **words**. The number of digits which form a word is known as the **wordlength**. The wordlength is fixed in the hardware design and cannot be changed by the programmer. The word is the fundamental addressable unit of memory. The wordlength of the 6502 is eight binary digits or bits. The fact that this length happens to coincide with that of a communications unit known as a byte has caused some confusion. We will use the term word throughout this book except when actually dealing with communications data. The eight bits of a 6502 word are numbered from zero through seven

beginning at the right, making bit number seven the highest or most significant numerical bit.

With each memory word is associated a unique numerical identifier known as its **address**. Addresses usually, but not always, begin with zero and extend without gaps to the highest physical memory word in the machine. The contents of memory are stored and retrieved by reference to the address of the required memory word, not by its contents. The system of addresses greatly resembles the system of box numbers in a post office. In the post office system it is not possible to answer a question like: What is the number of the box which contains the largest IRS refund check? To find this information all of the boxes must be searched. A similar situation exists with computer memory and the system of addresses. Communication with memory is always done by referring to the address of the required memory word. To answer a question like: which memory word contains the largest number, just as in the post office box analogy, the entire memory must be searched.

Memory can be further categorized as being **Random Access Memory (RAM)** or **Read Only Memory (ROM)**. The term random access has recently come to have a meaning different from its original one. The term originally meant a memory in which all parts could be accessed in the same time. It has come to mean memory into which data can be written as well as read. The term read only means memory from which data can be read but whose contents are fixed and not alterable by the normal program processes of the computer. ROM memory is normally nonvolatile and is commonly used to hold information which must not be lost when power is removed from the system.

The arithmetic and logical unit, or ALU, resembles a hand calculator in its simplest functions. It contains a **register** or set of registers each capable of temporarily holding a single computer word. Data are transferred between the registers of the computer and memory under control of the computer program. Data

which are moved from computer memory to a register are said to be **loaded** into the register. Data moved from a register to memory are said to be **stored** into memory. The meanings of these two terms are quite specific and imply the direction of data movement, load meaning movement from memory to a register and store meaning movement from a register to memory.

The principal register of the ALU is known as the **accumulator** or **A register**. It is in the accumulator that most of the computational work is done. There may be more than one accumulator in a computer but this is not the case with the 6502. The 6502 contains a single accumulator in which almost all computation is done. The exceptions to this are certain simple operations which can be performed directly on data contained in memory. In addition to the transfer of data by loading and storing, data from memory may be added to or subtracted from the contents of the accumulator. Likewise certain logical operations may be performed between data in the accumulator and data contained in memory. In any of these arithmetic operations between accumulator and memory the result always appears in the accumulator with the contents of the memory word being undisturbed.

The ALU also maintains certain flags or indicators which reflect the outcome of arithmetic and logical operations. If a result is generated, for example, which is too large to be held in a single computer word an indicator known as **overflow** is set. This indicator can then be tested by the program and the future course of the program modified as required. Other indicators reflect a variety of conditions. They will be discussed as the conditions arise in the text.

The third logical computer element is the **control unit** or **CPU**. The CPU also contains registers which are used to direct and control the activities of memory and the arithmetic unit. The principal registers of the CPU are known as the **instruction** or **I register** and the **program counter**, also called the **P register** or **PC**. The 6502 CPU also contains a pair of auxiliary registers

known as X and Y which participate in certain special memory access modes. The uses of X and Y will be covered in a later chapter.

The program counter or **P register** is a double length, i.e., 16 bit, register which contains the address of a memory word. The 16 bit length of the P register allows it to hold a number as large as 65,535. It can thus hold the address of any memory word in the 6502. The P register is said to function as a **pointer**, i.e., the number contained in it "points to" a location in memory. This concept of a pointer is probably the single most important notion required to program a computer at this level. Do not let this slip by without understanding it. If you are at all fuzzy about this idea stop now and go back over it until you understand it. The remainder of this book will be gibberish if you don't grasp the concept of a pointer. The function of the CPU's instruction or I register will be made clear in just a moment.

A **computer program** is a series of coded instructions, the successive execution of which performs some meaningful task. These coded instructions are held in computer memory. They are simply numbers just like the data upon which the program operates. Having said this, we can put the concepts discussed in this chapter together to define the fundamental computer execution cycle. What follows applies to any computer whatever, not just the 6502. By a means not yet specified, a series of coded instructions is entered into computer memory beginning at some address, the successive instructions occupying successively higher numbered memory addresses. The address of the first instruction is entered into the program counter, the 16 bit pointer discussed above. The machine is then started with this value in the program counter. The following events then occur:

1) The contents of the memory word pointed to by the program counter are read from memory by the CPU and loaded into the I or instruction register.

2) The number just loaded into I is interpreted by the CPU as a command to perform some action. If the action implied by the coded command requires more information than is contained in the command in the I register, one or more additional words may be fetched from memory.

3) An amount is added to the program counter sufficient to make it point to the next instruction.

4) The action implied by the coded command interpreted in 2) above is carried out. This may be any of a large number of different actions. It is especially important to note that this execution of the command takes place only after the program counter has been updated to point to the next instruction.

5) The entire sequence of steps is repeated beginning with 1) above.

If you have followed this through you now understand the fundamental cycle of computer instruction execution. It is especially important to come away from this with one fact about the cycle firmly in mind, i.e., that at the time the instruction is being executed the program counter points to the next instruction, the one which follows the instruction currently in progress.

We have not up to this point been specific about what kinds of operations are performed by the instructions. The instruction set will be treated in detail in the coming chapters. For now it will suffice to know something about the broad classifications of instructions.

The instruction set can be divided in a number of ways but the most meaningful from the point of view of the programmer is by function. The principal class of instructions, the one in which the real power of the

computer is resident is known as the **memory reference** class. Broadly speaking, these instructions require that a memory location be accessed to perform their functions, e.g., the addition of a pair of numbers requires that the addend be fetched from a memory location and added to an augend held in the machine's accumulator. The memory address of the word to be fetched is contained in the instruction in this case. A subclass of memory reference instructions is the control transfer class, instructions which modify the flow of the program by changing the program counter, either conditional upon some computational event or unconditionally.

A second distinct instruction class contains the **register operate** instructions. These instructions perform operations directly on quantities held in the 6502's registers and do not require that memory be referenced.

Yet a third class is concerned with the transfer of information between the computer and the outside world. These are normally known as the **input/output** instructions. The 6502 has no distinct input/output instructions, these functions being handled as a special case of memory reference.

If you are at all uncertain about any of the material in this chapter you are urged not to continue until the uncertainty is cleared up. All of these ideas are fundamental programming concepts and there is no possibility of understanding the remainder of this book without mastering them.

CHAPTER 2

THE BINARY SYSTEM AND BINARY OPERATIONS

As creatures who have evolved with ten fingers we tend to think of the decimal system of computation as being somehow a "natural" one, and experience a feeling of discomfort when required to deal with another system. This has not been true at all times in history, and our languages show abundant evidence of this. The English words dozen and gross are the remnants of the widespread use of twelve as a number base in the Middle Ages in Europe. Twelve has a good deal to recommend it for computation, being divisible by 2, 3, 4, and 6. Words like score testify to early use of twenty as a number base in climates in which the weather permitted people to go barefoot or wear open sandals. Becoming proficient with another number base is simply a matter of getting used to it.

Computers are built of components which have two possible stable states, on and off. Information to be processed by a computer must therefore be represented in some way capable of being represented by these devices. This is the base two or **binary system**. When dealing with numerical information the on and off states are usually called 1 and 0. With information which is not numerical they are sometimes referred to as set and reset.

While the idea of representing information in this form may seem strange, it is not so odd after a little thought. You have already seen information represented by two state codes many times. The red-green system of traffic signals is one example. The yellow is not a state at all, but a warning that a change of state is about to take place. The telegrapher's dot-dash code is another example.

Information is handled by the 6502 in groups of

eight binary digits or **bits**. With the exception of certain testing operations these eight bit groups are moved and manipulated in parallel. All of the operations so performed can be reduced to a small number of fundamental operations involving comparison of bits. The simplest of these are called the **logical** operations. In a logical operation a pair of bits is compared and the result set to reflect the outcome of the comparison. The commonest of these operations is known as the **inclusive OR**, also sometimes called the **logical sum**.

In an inclusive OR operation two bits are compared and the result set to one if either or both bits are ones. If and only if both bits are zero is the result zero. This is shown in tabular form below.

0 V 0 = 0
0 V 1 = 1
1 V 0 = 1
1 V 1 = 1

The "V" is the inclusive OR operator, originating from the Latin word vel, meaning or. It is used in the same sense that we use "+" as an addition operator. The inclusive OR function is used to merge two binary quantities. The inclusive OR operation is shown applied to eight bit quantities in example 2-1.

EXAMPLE 2-1

Perform the inclusive OR operation on the binary words 10111101 and 00101010.

First we write the two operands, one beneath the other, just as we would for an arithmetic operation.

```
  10111101
V 00101010
```

It makes no difference whether we perform the OR from right to left or left to right, but for the sake of consistency with arithmetic practice we will begin on the right. Recall from the last chapter that the bits of the 6502 word are numbered from 0 through 7 from right to left. In the 0 bit position we have 1 V 0 = 1. In the bit 1 position is 0 V 1 = 1. For bit 2, 1 V 0 = 1 and in the bit 3 position 1 V 1 = 1. In the 4 and 5 bit positions are 1 V 0 = 1 and 1 V 1 = 1. In the bit 6 position is 0 V 0 = 0 and in the leftmost bit 7 we have 1 V 0 = 1. The inclusive OR or logical sum of the two binary quantities is therefore:

```
  10111101
V 00101010
  10111111
```

Another important logical function is called the **AND** function, also sometimes known as the **logical product**. In the AND function the result is a one if and only if both of the bits being ANDed are ones. If either or both are zeros the result is a zero. This is shown in table form below:

$$0 \wedge 0 = 0$$
$$0 \wedge 1 = 0$$
$$1 \wedge 0 = 0$$
$$1 \wedge 1 = 1$$

the AND operation being indicated by the inverted V.

It may help your memory to think of the AND as a bit by bit product, a one appearing if both bits are ones and a zero if either of the multiplier bits is zero.

The AND operation performs a very important function called **masking**. This use of AND is shown in example 2-2.

EXAMPLE 2-2

The ASCII code for the decimal digit 6 is 10110110. Use the AND function to isolate the low four bits by masking out the high four bits.

To do this a mask must be constructed with ones in the positions we want to preserve and zeros in the positions to be cleared. Since the high four bits are to be cleared and the low four preserved this mask is 00001111. Therefore:

```
  10110110  the ASCII code
Λ 00001111  the mask
  00000110  the result
```

The reader already familiar with the binary number system may notice that the low four bits of the result, 0110, are the binary representation of the decimal number 6.

A logical operation which is performed on a single binary quantity is **complementation**, also known as the NOT function and **ones complementation**. It is extremely simple. Each bit of the number to be complemented is simply inverted. All ones are changed to zeros and all

zeros are changed to ones. The operation is shown in example 2-3.

EXAMPLE 2-3

Find the complement (ones complement) of 10101010.

The complement is found by inverting every bit, i.e., ones change to zeros and zeros change to ones. The complement of 10101010 is therefore 01010101.

An extremely useful logical function is the **exclusive** OR. Exclusive OR is not a fundamental operation since its effect can be achieved with AND, inclusive OR and complementation. The function is so useful, however, that we define it separately. The exclusive OR is sometimes called the **logical difference**. The result of exclusive OR is a one if and only if the operand bits are different. If they are both ones or both zeros the result is a zero. An easy way to remember it is to think of the way in which the sign of a product or quotient is fixed in algebra. If the signs of the operands are the same the result is positive. If they are different the result is negative. This function is one of the uses of exclusive OR in computers. The signs of multiplier and multiplicand or dividend and divisor can be exclusive ORed together to find the sign of the product or quotient without ever performing the multiplication or division. The exclusive OR operator is ⊻. The exclusive OR is shown in table form below.

$$0 \veebar 0 = 0$$
$$0 \veebar 1 = 1$$
$$1 \veebar 0 = 1$$
$$1 \veebar 1 = 0$$

Note again that the result is a one only when the operand bits are different. The exclusive OR function is illustrated in example 2-4 below.

EXAMPLE 2-4

Find the logical difference (exclusive OR) of the two binary quantities 10111011 and 10100101.

$$\begin{array}{r} 10111011 \\ \veebar\ 10100101 \\ \hline 00011110 \end{array}$$

The operand bits in each column are exclusive ORed and the result written below. Before leaving this example notice that it follows from the rules for the exclusive OR function that the exclusive OR of two quantities will be all zeros if and only if they are identical. This has an application in the comparison of data. Suppose an eight bit word in computer memory contains the last status of a switch register read from an external device. When the new reading is taken exclusive ORing it with the old reading will yield a nonzero result if any of the eight switches has changed status. Further, the switch or switches which changed status will be indicated by ones in the result word.

THE BINARY SYSTEM AND BINARY OPERATIONS

We must now deal with arithmetic in the binary system. As in decimal, the successive leftward digits of a binary number have place values. These place values, again like decimal, represent higher and higher powers of the number base. In decimal these are 10^0, 10^1, 10^2, etc. In binary they are 2^0, 2^1, 2^2, and so forth. A table of the first few powers of 2 is shown below.

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$
$2^{11} = 2048$
$2^{12} = 4096$
$2^{13} = 8192$
$2^{14} = 16384$
$2^{15} = 32768$
$2^{16} = 65536$

Conversion of a binary number to its decimal equivalent is straightforward using the above table. Wherever a one occurs in the binary number the appropriate power of two is added. Thus the binary number 00001101 has the decimal value 13 since it contains ones in the 2^0, 2^2 and 2^3 columns.
Its value is therefore the sum of these powers of two, i.e., 1 + 4 + 8.

The number 00001111 has the decimal value 15, the sum of the first four powers of two, as indicated by ones in the low four positions. This number shows a special case of a very useful general rule. Since it contains an unbroken chain of ones in its low four

positions, the low four bits represent the largest number which can be held in four bits. Adding one to this number would result in a carry into the 24 position. This leads to a very useful general rule, namely:

The largest number which can be held in N bits is 2^N-1.

Thus the largest number which can be held in ten bits is $2^{10}-1$ or 1023. This rule is useful when estimating the length of binary precision necessary to achieve a given decimal precision. An approximate form of this rule is also very useful. Note that 2^{10} above is just a little greater than 10^3 or 1000. The following rule of thumb thus emerges:

Each ten bits of binary precision corresponds to about three decimal orders of magnitude.

We are used to thinking about the process of addition as being a serial one, beginning at the right and adding column after column until the leftmost column has been processed. A binary addition of this type is shown in example 2-5.

EXAMPLE 2-5

Perform the binary addition:

```
  00001001
+ 00001100
```

Beginning at the bit 0 position at the right we add the units digits to get:

```
      0
  00001001
+ 00001100
         1
```

The sum of the units column being one. The zero written above the twos column represents the carry from the units to the twos column. It is useful to think of the carry as always being present, even if its value is zero. The twos column is next added to yield:

```
      0
  00001001
+ 00001100
        01
```

the zero carry from the twos to the fours column again being indicated above. Adding the fours column we have:

```
      0
  00001001
+ 00001100
       101
```

Adding the eights column gives:

```
      1
  00001001
+ 00001100
      0101
```

with the carry being a one this time since the binary sum 0 + 1 + 1 = 10, i.e., the 0 is written into the sum and the one carried into the sixteens column. Proceeding to the sixteens column we get:

```
    0
  00001001
+ 00001100
    10101
```

The sum of the carry from the eights column and the two sixteens column bits is 1 + 0 + 0 = 1. Since the carry is a zero and there are no more significant bits to the left the addition is finished, the final sum being 00010101.

Subtraction can also be done in a manner parallel to decimal pencil and paper subtraction but a problem occurs when this is attempted. Consider the subtraction:

```
  00001000
- 00000001
```

In subtraction the upper number is called the **minuend** while the lower one is called the **subtrahend**. Attempting to subtract the units digit of the subtrahend from the units digit of the minuend shows the problem immediately. Since the minuend digit is a zero a borrow must be made from the subtrahend column to the left. Both this column and the next ones are zero, however, and no signifiant subtrahend digit occurs until the eights column. To get around this we modify the subtraction process a bit. The operation we call borrowing amounts to decrementing, i.e., decreasing by one, the minuend column to the left of the current one. The same thing can be accomplished by incrementing the subtrahend column to the left of the current one. This gets around the problem of the borrow completely and is much more attractive from the hardware point of view. A subtraction done by this process is shown in example 2-6.

EXAMPLE 2-6

Perform the binary subtraction 00001000 - 00000001 by the method of subtrahend incrementation.

```
  00001000
- 00000001
```

Attempting to subtract the units digits requires a borrow. Instead of attempting the borrow we increment the twos digit of the subtrahend. This allows the units digit of the minuend to be considered a binary 10 for computational purposes. The subtraction of the units digits can now be performed and the result looks like this:

```
  00001000
- 00000011
         1
```

Attempting to subtract the twos digits we encounter the same problem, so the subtrahend fours digit is incremented and the twos digit subtracted from 10 as before. The result is:

```
  00001000
- 00000111
        11
```

Subtracting the fours digit involves the same process, whose result is:

```
  00001000
- 00001111
       111
```

The subtraction of the eights column involves no borrowing, being a simple 1 - 1 = 0. The 0 is

written in the result. Since there are no more significant bits to the left of the eights column the subtraction is finished and the result is:

```
  00001000
- 00001111
  00000111
```

Yet another method of subtraction is by the addition of negatives. This presents us with the problem of how to represent a negative number.

The 6502 represents negative numbers by means of a system known as the **twos complement system.** To form the negative of a number in the twos complement system, the ones complement, as defined earlier in this chapter, is first found, and then one is added to it. To find the twos complement of 00001000, first write out the ones complement which is formed by inverting every bit as shown in example 2-3, i.e.:

```
11110111
```

Now add one to this:

```
  11110111
+ 00000001
  11111000
```

This result, 11111000, represents -8 in the twos complement system. If this seems doubtful it can easily be tested. If 11111000 is really -8 then adding +8 to it ought to give zero, i.e.:

```
  11111000
+ 00001000
  00000000
```

The reader who has actually gone through the above addition may find something troublesome about it.

While the eight bit result is correct the addition of the two high bits resulted in a carry which would have gone into the ninth bit if there were one. Fortunately, there is a ninth bit specifically for this purpose. The 6502 catches and preserves carries out of the high bit of a sum in an indicator known as the **carry bit**. This carry bit also functions as a borrow indicator, i.e., after a subtraction the state of this bit tells whether a borrow would have been necessary from a ninth bit if it existed. More will be said about this in chapter 6.

When the twos complement of the number in the example above was formed the leftmost bit became a one. When dealing with signed numbers this leftmost bit is known as the **sign bit**. If the sign bit is zero the number is positive, if a one the number is negative. This is a convenient way to view signed numbers, but there is more to it than this, which will become apparent when we deal with arithmetic shifts later in this chapter.

Before leaving the subject of the twos complement system, there is one point which must be made. In any twos complement system, whatever the wordlength, there is always one more number available in the negative direction than in the positive. The largest possible <u>positive</u> eight bit number is:

01111111

or +127 decimal. The largest possible <u>negative</u> number is:

10000000

or -128 decimal. This is not peculiar to the 6502 or any other computer. It is a characteristic of twos complement systems that must be kept in mind when working with them. It is instructive to attempt to find the negative of the largest negative number, i.e., its positive counterpart. Following the procedure for

twos complementation, we begin by finding its ones complement, i.e.:

```
01111111
```

which is formed by simply inverting every bit. Adding one to this ones complement:

```
  01111111
+ 00000001
  10000000
```

In other words, attempting to find the negative of the largest negative number yields the largest negative number as a result. We have negated a negative number and gotten a negative number as a result. This is algebraic nonsense, of course, and the programmer must be on guard lest this occur. The computer hardware offers some assistance here by maintaining a facility known as the **overflow bit**. The overflow bit is set automatically by the computer hardware if an attempt is made to generate a result too large to be held in a single computer word. Attempting to negate the largest negative number is a special case of this error. More will be said of the overflow bit in chapter 5.

The final logical functions to be considered in this chapter are the shifts. To understand what a shift is imagine a conveyor with eight spaces on it. This conveyor can be moved to the right or left, only in units of a whole space, not part of a space. A one is represented by a box in a space and a zero by the absence of a box. Now move the conveyor one space left in your imagination. If there was a box in the leftmost space it fell off the end of the conveyor, while an empty space entered from the right. Now load up your imaginary conveyor again and move it one space to the right. If there was a box in the rightmost position it fell off the end of the conveyor, and an empty space entered from the left. This is the most fundamental kind of shift and is known as a **logical**

shift. Let's examine the consequences of shifting a number in this way. Take the number:

00010101

which is a decimal 21. Shift this number left one place and we have:

00101010

a decimal 42. The effect of shifting a number left is to multiply its value by two. If we shift left again we have:

01010100

a decimal 84. This rule only applies, of course, as long as you don't shift a significant bit out the left end of the word. A logical right shift has just the opposite effect, it divides by two, but the effect is a little more complicated than that. Starting with the same number, a decimal 21:

00010101

we shift right once, and get:

00001010

a decimal 10. A significant bit was shifted out the right end and lost, having the effect of dividing by two and discarding the remainder. The exact description of what happens in the general case is a little more complicated than this, i.e., the effect of a right shift is to divide by two and round the result toward the next more negative number.

When the right shift above was performed a one was shifted out and lost. Strictly speaking, it is not lost beyond retrieval, since bits shifted out of either end of a word are preserved momentarily in the carry

bit. This is a property of the carry bit which is much exploited by programmers, since it offers a way of testing bits in a word by testing the status of the carry bit after a shift.

A second kind of shift, and the one most commonly available on microprocessors is the **rotary shift.**

In a rotary shift bits shifted out one end of a word re-enter through the other end after being held in carry. The eight bit word and the carry bit together thus form a nine bit register. The effect of a rotary shift with carry is easily shown. Suppose the word contains all ones and carry is a zero, i.e.

0 11111111

the carry being shown as detached at the left. Executing a rotary left shift yields:

1 11111110

and a second shift gives:

1 11111101

and so forth. Right shifts simply rotate in the opposite direction. Note here that any rotary shift is the same thing as a logical shift in the same direction if the carry bit is cleared to zero before each shift.

The final type of shift, and the most complex, is the **arithmetic shift.** In an arithmetic shift only the rightmost seven bits of the 6502 word are shifted. The sign bit is undisturbed. In an arithmetic right shift the sign bit value is copied into the vacated space to its right. Take the number:

10000000

a decimal -128. If this number is shifted right we get:

11000000

a -64. The negative sign bit has been filled into the vacated space. Shifting right again we get:

11100000

a -32. If we continue to shift right in this way, after 5 more shifts we have:

11111111

which is a -1. If you are in doubt about this try finding its twos complement as an exercise. If we shift this number again we get:

11111111

the same thing, a -1 ! Now think back to the definition of the effect of a right shift. Shifting right divides the number by two and rounds toward the next most negative number. Minus one divided by two should give -.5, but when this is rounded down we get the minus one again.

An arithmetic left shift simply moves the rightmost seven bits of the word to the left. The sign bit does not participate in the shift. The effect of such a shift is to multiply the number by two except when a significant bit is shifted out of bit 6 and lost. The 6502 hardware provides no warning that this has happened. It is the programmer's responsibility to check for this condition.

The specific types of shifts available vary from machine to machine. On the 6502 both logical and rotary shifts are available. There are no arithmetic shifts on the 6502.

The final topic for this chapter is hexadecimal notation. Writing or reading long strings of binary digits is a tedious and error prone process. For this reason shorthand methods for writing binary data have been developed, the **octal** and **hexadecimal** systems. We

will have no need for octal here, but understanding hexadecimal is necessary for working with the 6502.

The hexadecimal, or base 16, system is used as a shorthand for binary because of the ease of conversion between the two systems. In base 16 or any number system which uses an integral power of two as a base the conversion between that number base and binary is a simple matter of substituting a symbol for a bit group and vice versa. To convert a binary number to hexadecimal the number is divided into four bit groups and the hexadecimal equivalent symbols substituted for the groups. For numbers less than ten the hexadecimal symbols used are simply the decimal numbers. A four bit group, however, can contain a number as large as fifteen, and the decimal system has no single symbols for these numbers. Arbitrarily we use the first six letters of the alphabet for this purpose. The binary equivalents of the hexadecimal symbols are shown in the table below.

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Using this table, conversion from binary to

hexadecimal is easy. Take the binary number 11100111. First it is divided into two groups of four bits, like this:

1110 0111

and then the hexadecimal equivalent of each group is read from the table. The left group has value E and right group value 7. The hexadecimal equivalent of the binary number 11100111 is therefore E7. Another such conversion is shown in example 2-7.

EXAMPLE 2-7

During programming of microprocessors a structure called a pointer is commonly encountered. A pointer is a 16 bit number which contains the address of a memory word. Pointer values are usually expressed in hexadecimal. Find the hexadecimal value of the 16 bit binary pointer value

1010111000111011

The number is first divided into four bit groups:

1010 1110 0011 1011

and the hexadecimal equivalent of each group read from the table. The left group is a binary ten whose symbol is A. The next group is a binary fourteen, symbol E. The last two groups, a three and an eleven, have symbols 3 and B. The hexadecimal value of the number is therefore AE3B.

Conversion from hexadecimal to binary is equally simple. The appropriate four bit binary value from the table is substituted for the hexadecimal digit. Such a conversion is shown in example 2-8.

EXAMPLE 2-8

Convert the hexadecimal number D7F9 to binary.

Read the binary values of the successive digits directly from the table. The D has value 1101, the 7 value is 0111, the F value is 1111 and the 9 has value 1001. The binary equivalent of D7F9 is therefore:

1101 0111 1111 1001

CHAPTER 3

MACHINE AND ASSEMBLY LANGUAGES

Since computer instructions are simply numbers which occupy computer memory along with the numbers and other data upon which they operate, it ought to be possible to construct a purely numerical program, enter the program into memory and cause it to be executed. This is indeed possible and is sometimes done. Such a program is shown below. It compares the three numbers in hexadecimal locations 319, 31A and 31B, finds the greatest of the three and places it into location 31C.

location	contents
300	AD
301	19
302	03
303	CD
304	1A
305	03
306	B0
307	03
308	AD
309	1A
30A	03
30B	CD
30C	1B
30D	03
30E	B0
30F	03
310	AD
311	1B
312	03
313	8D
314	1C
315	03

The program above is certainly valid and will

produce the results quoted if executed. The problem is its form. This program certainly talks to the machine in its own language, but the language is largely gibberish to the human programmer. Being written in hexadecimal, it is a bit easier to view than if it were written directly in binary, but there is still nothing about AD that suggests loading the 6502's A register, which is what AD does. Likewise there is nothing about CD and 8D which suggests what their functions might be. It is certainly possible to write programs this way but the process is so inefficient that a project of any size is impossible. Programmers have recognized this difficulty since very early in the programming craft, of course, and did something about it very quickly.

What is obviously required is some means of representing computer instructions in a form more meaningful to human programmers. Since AD represents an instruction which causes the 6502's A register to be loaded from a memory location, why not construct a program which will accept some abbreviated form of "load A" and translate it automatically to AD or whatever other machine instruction is appropriate? This is exactly what was done in the early days. Programs were constructed which recognized abbreviations of the meanings of instructions and translated (or transliterated) them to the binary machine form. Thus we have LDA for Load A register, CMP for Compare A register and STA for Store A register. There is such an abbreviation for every instruction in the 6502. These abbreviations are known as **mnemonic instructions** or just **mnemonics** (pronounced ni-monic, with a silent m).

A program which could translate mnemonics into machine instructions was called a **mnemonic loader**. Using a mnemonic loader the programmer was allowed to write a program which looked something like this:

```
LDA  $319
CMP  $31A
BGE  $30B
```

```
LDA  $31A
CMP  $31B
BGE  $313
LDA  $31B
STA  $31C
...
```

This is certainly a very large improvement over programming in hexadecimal or binary in that it frees the programmer from having to remember the numerical form of each instruction. It still requires that the programmer keep exact track of the location of everything in memory. To understand what a problem this can be consider the first BGE instruction in the above program. BGE means branch if greater than or equal. It is a synonym for another instruction and is used to alter the program flow based on the results of comparison of two program quantities. The BGE must specify the address of the next instruction to be executed, however, and since the jump goes toward a higher memory address, i.e., one which has not been filled in by the programmer at the time the BGE is written, the programmer must either anticipate the exact distance to be jumped or go back and fill in the operand of the BGE after the jump point has been located. This is worse than a nuisance. It is so burdensome that developing a program of any size with a mnemonic loader is impractical.

Mnemonic loaders have been revived recently under names like "mini-assembler" and such. This is pure verbal inflation. A mnemonic loader is no more an assembler than a barnacle is a ship.

What is required here is a system in which most or all of the housekeeping chores are done by the machine. Ideally such a system would allow names to be given to memory locations with subsequent references to these names rather than the numerical locations. The program would be written purely in symbols, with the programming system translating these symbols into the appropriate machine instructions and addresses.

A program capable of doing this is known as an **assembler** or **assembly program.**

Perhaps it is best to start the subject by showing you the same program we have been considering, but written in the symbolic language of the assembly program, called the **assembly language.** The program is shown below:

```
        ORG   $300      START AT $300
TEST    LDA   N1       GET 1ST NUMBER
        CMP   N2       COMPARE TO 2ND
        BGE   T2       SKIP IF 1ST GREATER
        LDA   N2       LOAD 2ND NUMBER
T2      CMP   N3       & COMPARE TO 3RD
        BGE   T3       SKIP IF GREATER
        LDA   N3       LOAD 3RD NUMBER
T3      STA   MAX      SAVE RESULT
T4      JMP   T4       AND WAIT HERE
N1      ***            1ST NUMBER
N2      ***            2ND NUMBER
N3      ***            3RD NUMBER
MAX     ***            BIGGEST OF 3
        END
```

It is not important to understand just what all of this means at this point. Do note that there are no references in the above program to absolute numerical addresses. The numerical addresses are handled entirely by the assembler. The assembler keeps automatic track of the location of T3, for example, and adjusts the BGE instruction to jump to the correct memory location.

It is important to realize just how large a step has been taken here. When programming in absolute numeric or with the mnemonic loader we were dealing directly with the machine. Each instruction written in these systems corresponds exactly to a memory word or small group of words whose size is known exactly. In using an assembler we are not dealing directly with the machine but with an intermediate program which

interprets our symbolic commands. Unlike an instruction written for a mnemonic loader, an assembly language instruction may generate a number of program words unknown to the programmer, or no words at all. A case in point is the ORG in the program above. ORG generates no program words. It is an instruction to the assembler rather than to the machine. It instructs the assembler to translate the symbolic instructions which follow as if they were to be loaded and executed starting at hexadecimal address 300. The dollar sign ($) specifies that the 300 which follows it is to be considered hexadecimal. It must be kept crystal clear in your mind that you are dealing with an intermediate translator rather than the computer itself.

Each line of an assembly language program has four possible fields, only three of which are shown in the program above. The leftmost field, containing the symbols TEST, T2, etc., is known as the **label** field. The label field may be blank or may contain a symbol. If it contains a symbol the effect is to name the memory location with which it is associated. This gives the symbol in the label field a value. Since the program above starts at location $300 and the TEST label is attached to the first instruction of the program, the LDA N1, the value of the symbol TEST is $300. Labels have other uses as well, which will be explained as the need arises. The use of a symbol in the label field of a line is said to **define** that symbol.

The second field, the one containing LDA, CMP, BGE, etc., is known as the **instruction field.** It contains the symbolic specification of the instruction itself. It may alternately contain a command to the assembler, as in the case of the ORG. Note the instruction fields near the end of the program which contain ***. This is not an instruction but a command to the assembler to reserve a memory location in this place under whatever name may be in the label field.

The third field, containing N1, N2, T2, etc., is called the **operand** field. It may or may not contain

anything, depending on the instruction. While all of the instruction operand fields are occupied in the program shown, there are instructions which require no operand. In these cases the operand field is left blank. Note that the operand fields contain no symbols which have not been defined in a label field. The final field occupies the space to the right of the operand field. It is called the **comment field** and plays no part whatever in the program itself. It is simply a place for the programmer to makes notes about the instructions he is writing. This field is the primary documentation space for the program being created.

To use an assembler the programmer must first create the program to be assembled. He writes the string of assembly language lines required to solve the problem at hand and then puts them into machine readable form. The symbolic program so created is known as the **source program**. The source program can be punched onto Hollerith cards or paper tape or, as is the case with our assembler, keyed directly into the computer. The assembly program then scans the source and produces one of three possible results. Probably the most important from the programmer's point of view is the **assembly listing**. This is a side by side listing of the symbolic source program created by the programmer and the numerical result produced by the assembly program, i.e., the actual machine language program. This side by side listing is followed by an alphabetic list of the symbols used by the programmer in the source program. This alphabetic list of symbols is called the **symbol table**. The full assembly listing of the sample program is shown in example 3-1.

Let's look at this assembly listing in a little more detail. The leftmost column is the line number. This plays no part in the assembly process. It is simply for editing convenience when the program is to be changed. Program editing will be taken up in the next chapter. The next column is the memory location where the first word of the instruction on this line is

located. The line which reads: TEST LDA N1, for example, generates three words, the first of which is in location $300, the second in $301 and the last in $302. The next instruction thus begins in $303, as shown by the location column of the next line. The third column from the left shows the actual machine language result of the processing of this line, AD, 19, and 03 in this case. This column of the listing may show from one to four words, which accounts for its ragged appearance.

EXAMPLE 3-1

Sample Assembly Listing

```
PAGE 001

001 0300                    ORG  $300      START AT $300
002 0300 AD1903     TEST    LDA  N1        GET 1ST NUMBER
003 0303 CD1A03             CMP  N2        COMPARE TO 2ND
004 0306 B003               BGE  T2        SKIP IF 1ST GREATER
005 0308 AD1A03             LDA  N2        LOAD 2ND NUMBER
006 030B CD1B03     T2      CMP  N3        & COMPARE TO 3RD
007 030E B003               BGE  T3        SKIP IF GREATER
008 0310 AD1B03             LDA  N3        LOAD 3RD NUMBER
009 0313 8D1C03     T3      STA  MAX       SAVE RESULT
010 0316 4C1603     T4      JMP  T4        AND WAIT HERE
011 0319 00         N1      ***            1ST NUMBER
012 031A 00         N2      ***            2ND NUMBER
013 031B 00         N3      ***            3RD NUMBER
014 031C 00         MAX     ***            BIGGEST OF 3
015 031D                    END

******************************
*  S Y M B O L   T A B L E  *
******************************

MAX    031C  N1    0319  N2    031A  N3    031B  T2    030B  T3    0313
T4     0316  TEST  0300
```

A secondary output of the assembly program, but

very important from the point of view of programmer convenience and efficiency is the **error listing**. As will become apparent in the chapters to come, the assembly language has a "grammar" all its own, and violations of the laws of this grammar are considered to be errors, and are so treated by the assembly program. The assembler is capable of performing a scan of the source program for such errors, printing only lines which contain errors. This allows for efficient editing of the program being developed. After an error scan the problems can be corrected by the editor program and another scan executed, this process being repeated until all errors have been eliminated.

Last, but most important from the operational point of view, the assembler produces an **object program**. The object program is the machine language version of the source program in machine readable form. The object program may punched onto cards or paper tape or written onto magnetic tape, depending on the peripheral devices available to the computer.

The assembler we will be using for the examples in the chapters will follow was created especially for this book. It adheres, within the limits of good sense and design, to previously established language conventions for the 6502. A number of features have been added to the language to enhance clarity and reduce the great volume of writing required which have hobbled this processor in the past.

While this assembler will run on any 6502 based computer, the specific configuration supplied here is for the Apple II. Detailed instructions for installing it on other 6502 machines are given in an appendix. If you have a machine other than the Apple II it is suggested that you install the assembler on your machine now. The remainder of the book is written on the presumption that you are able to run the example programs on an actual computer and observe what happens. While you can certainly learn something from just reading the text, nothing can replace the experience of actually running programs. It is only in

this experience that any real learning takes place. Good programmers are made, not born, and they are made by the same process that produces good brain surgeons, baseball players and engineers - plenty of experience. Only by actually performing a task do you know anything about it.

In the next chapter we will take up the actual mechanics of using the editor/assembler. If you need to install your assembler in a non-Apple II machine do it now!

CHAPTER 4

USING THE ASSEMBLY PROGRAM

In this chapter we will study the actual mechanics of creating and editing a source program and assembling it for listing and object program. If you have not heeded the advice at the end of chapter 3 and installed the editor/assembler this chapter and the remainder of the book are likely to be of limited value to you. Turn to the appendix and install the editor/assembler now! Do it!

The editor/assembler system is operated by a set of eleven simple commands, six of which control editing, three are associated with assembly functions, and two with memory and peripheral management. The system indicates its readiness to accept a keyboard command by displaying a caret or "greater than" symbol (>). All system commands are terminated by a period. If the command is to be aborted before the period is hit, just type any illegal character. In the Apple II version of the system the screen position of the next command character is indicated by a flashing period.

The programmer enters his source program into the 6502 system by typing at the system terminal device. In the Apple II this is the keyboard. The source program is held in an area of memory known as the **source buffer**. This source buffer begins immediately after the end of the editor/assembler and its associated input/output routines and continues to the end of high available memory. Since the programmer may have utility programs of his own located in high memory the system offers him a way of preventing the source buffer from overlaying these utilities. Setting the upper memory limit for the source buffer is accomplished by the M command, i.e., type:

>M.

and the system will respond with:

ddddd LEFT, EOB = hhhh.

The ddddd is a decimal number indicating the number of memory words left in the buffer. For a 48K Apple II this number is about 25,000, enough to accomodate a huge program. The EOB means end of buffer. The hhhh is the hexadecimal address of the last available source buffer word. The system will pause with the period at the end of the line flashing. If a hexadecimal number is typed followed by a period, its value will replace the one shown. If the upper buffer limit is not to be changed type a period and the prompt character will reappear, meaning that the system is waiting for another command.

The system is initialized and the buffer cleared by the slash (/) command. This function is performed automatically on cold start. Just type:

>/.

to initialize. DO NOT do this if there is a program in buffer that you want. Initializing clears the buffer and your program will be lost.

The basic source buffer unit is the line. Text is added to and deleted from the buffer in whole lines or groups of lines. It may help to think of the buffer being organized something like a stack of old fashioned printer's type slugs. Each slug represents one line of type. Imagine that there are twenty of these slugs in a stack and that slug number 17 is removed. The former slug 18 now becomes 17, the former 19 becomes the new 18 and the former 20 becomes 19. Now imagine a new slug being added after the third one in the stack. The old number 4 slug becomes the new number 5, the old number 5 is the new number 6 and so forth, each slug after the one added being renumbered to reflect its new position. The text editor works in an exactly analogous way.

After you have given the slash (/.) command there is nothing in the source buffer. Just to see how it is done, let's enter the program from chapter 3 into the buffer. In response to the system prompt we type:

```
>A.
  ORG $300 START AT  $300@
 TEST LDA N1 GET 1ST NUMBER@
  CMP N2 COMPARE TO 2ND@
  BGE T2 SKIP IF 1ST GREATER@
  BGE T2 SKIP IF 1ST GREATER@
  LDA N2 LOAD 2ND NUMBER@
 T2 CMP N3 & COMPARE TO 3RD@
  BGE T3 SKIP IF GREATER@
  LDA N3 LOAD 3RD NUMBER@
 T3 STA MAX SAVE RESULT@
 T4 JMP T4 AND WAIT HERE@
 N1 *** 1ST NUMBER@
 N2 *** 2ND NUMBER@
 MAX *** BIGGEST OF 3@
  END@
 .
>
```

in which the character @ means the RETURN or ENTER key on the keyboard. The editor knew we were finished entering text when it saw a line which began with a period (.), and responded with the prompt character, meaning it is ready to accept a new command. Notice that what we typed in doesn't match exactly with the listing in chapter 3. We didn't type the program in all neatly spaced as it appeared there. Where there was a blank field, like empty label fields, we left just one blank. Don't let this worry you. The system has enough intelligence to know that a blank in the first space of a field implies that the field is all blank.

After typing the program in we'd like to see what we've typed. This is done by the L command. The L command needs to know how much of the program we want

to see, so we have to provide the numbers of the first and last lines to be listed. If you've forgotten how many lines you typed in, don't go back and count them. Using the symbol F instead of a line number means the final buffer line, whatever it is. To list the entire buffer we type:

```
>L1 F.
```

and the system responds with:

```
001             ORG  $300       START AT $300
002  TEST       LDA  N1         GET 1ST NUMBER
003             CMP  N2         COMPARE TO 2ND
004             BGE  T2         SKIP IF 1ST GREATER
005             BGE  T2         SKIP IF 1ST GREATER
006             LDA  N2         LOAD 2ND NUMBER
007  T2         CMP  N3         & COMPARE TO 3RD
008             BGE  T3         SKIP IF GREATER
009             LDA  N3         LOAD 3RD NUMBER
010  T3         STA  MAX        SAVE RESULT
011  T4         JMP  T4         AND WAIT HERE
012  N1         ***             1ST NUMBER
013  N2         ***             2ND NUMBER
014  MAX        ***             BIGGEST OF 3
015             END
>
```

and we can examine what we have done. The first thing we notice is that the line with label N3 got lost somewhere, i.e., we forgot to type it in. This is easy to fix. The line should go after the current line 13, so we type:

```
>A13.
N3 *** 3RD NUMBER@
.
>
```

and this inserts the missing line. The effect of

insertion of a line after line 13 is to renumber all of the lines which follow it in the buffer, i.e., the line beginning with MAX becomes line 15 and the line containing END becomes the new line 16.

Now we notice something else wrong. We somehow managed to type the BGE T2 SKIP IF FIRST GREATER line twice, a common enough typing mistake. Lines 4 and 5 are duplicates. This is easy enough to fix. We can get rid of either line with the D command. Just typing:

```
>D4.
```

does the job. Now we have only one copy of the duplicate line. Note that after deleting this line all of the lines following the deleted one are renumbered.

If an entire block of lines were to be deleted the D command would be used with two line numbers after it. If we wanted to delete lines 7 through 13 inclusive, it could be done with:

```
>D7 13.
```

Remember that you can use F for a line number if you want to specify the final line of the buffer. The command:

```
>D1 F.
```

does the same thing as a slash (/.) command. It clears the entire buffer.

Now that you've gone to all the trouble to type this program in, it would be a good idea to save it in case there is a power failure or somebody plugs in the toaster, rotisserie, and electric iron all at the same time and blows the fuse. The source program can be saved on an external medium with the W (write) command. W works just like L except that the writing takes place to cassette or other medium instead of the listing device. To save the program, type:

```
>W1 F.
```

The system will respond with a message telling you to turn on the tape machine or whatever you are using to hold source externally. After that just tap the space bar and the writing will begin. When it is finished the system will issue a prompt, and it is ready for another command.

When you want to read the program back from the external medium, use the R command, i.e., type:

```
>R.
```

The system will display a message telling you to turn the cassette or other device on. Press the space bar and the reading begins. When the program has been read the system issues another prompt, and you are ready to go on to other things. Note that text read in this way is added to any text that may already be in the buffer. This is an easy way to put two or more segments of a long program together. If you want just the program you are reading from the external medium use a slash command to clear the buffer first.

The actual assembly functions of the system are controlled by # command. The # command is always used with a modifier, E, 0, or L. Typing:

```
>#E.
```

results in an error scan of the program. What kinds of errors can be detected will be discussed later in this chapter. The #E mode of assembly saves substantial amounts of time during program development in that it performs very rapid scans of the program under development, displaying only those lines which contain assembler detectable errors. This saves generating a listing of the entire program.

The #0 (0h, not zero) mode of assembly generates the object program and writes it onto an external

medium like cassette. The object program is later reread from the external medium by a program known as a **loader**. The loader loads the object program into 6502 memory in preparation for its trial execution. The loader program is part of the debugging monitor which will be discussed in a later chapter.

The #L mode of assembly performs assembly of the program and produces the assembly listing, the fundamental document which describes the program. It requires that the beginning and ending lines to be listed be specified. This is to allow part of an assembly listing to be generated, a real convenience when something has to be changed toward the end of a long program, since it saves the redundant listing of the correct early portion. To generate an assembly listing of the entire program type:

```
>#L1 F.
```

the line numbers having the same meaning as in the L command.

Finally, it is a convenience to be able to switch peripherals and for this purpose the U (use) command is provided. The U command requires two modifiers. The first of these is L, O, R, or W. The second modifier is a number. The scheme works like this. Typing:

```
>UL1.
```

switches all listing, whether originating from an L or #L command to the primary listing device, while:

```
>UL2.
```

switches it to the secondary listing device. In the case of the Apple II implementation of the system the primary listing device is the screen and the secondary device a parallel printer. A full listing of all of the standard devices is given at the end of this chapter.

The UOn command switches the device to which the object output of the assembler is to be written, the n again being the device number, while UWn and URn control the device to which the source buffer is written with a W command and from which source text is read into the buffer with an R command. Do not take this idea of "standard" devices too seriously. The system is flexible enough to accomodate almost any set of peripherals. A full specification of the procedures necessary to accomodate "nonstandard" peripherals is given in appendix B. The Apple II user can operate with the system as provided.

The assembler is capable of detecting certain kinds of clerical programming errors. These are flagged in the assembly listing with a one character error code on the line immediately below the offending one. These codes are:

U

The operand field of the line contains some element which has not been defined by appearance in a label field, or a numerical element contains a nonsense character.

M

The label field of this line contains a symbol which is defined more than once. The symbol is said to be **multiply defined**. The ambiguity which results must be resolved be removing one of the definitions.

L

The operand field of the line contains some element which cannot be held in the specified space, e.g., attempting to specify the decimal number 500 in eight bits, or the number 100,000 in sixteen bits.

O

The operation specified in the instruction field is not a standard 6502 instruction or one of the compound or pseudo-instructions of the system. Typing nonsense in the instruction field produces this error.

R

An attempt has been made to specify a branch target address for one of the 6502's conditional branches which is out of range of the branch.

I

An improper indexing character has been specified in the operand, e.g., the instruction:

```
ASL  SYM,Y
```

will produce this error, since ASL cannot be indexed by Y.

STANDARD INPUT/OUTPUT DEVICE ASSIGMENTS

L1 - Apple II screen.
L2 - Parallel printer in slot 3.
L3 - Writer's TTY.

O1 - Apple II cassette.
O2 - Writer's TTY.

W1 - Apple II cassette.
W2 - Writer's TTY.

R1 - Apple II cassette.
R2 - Writer's TTY.

CHAPTER 5

MEMORY REFERENCING ON THE 6502

In this chapter we will be concerned with the various methods used by the 6502 memory reference instruction set to access computer memory. Not all of these methods, known as **addressing modes**, are available to all instructions, nor would this be appropriate. The specific limitations upon each instruction class will be noted as the class comes up in the text.

The 6502 may be thought of as having two different kinds of memory, addressable by two parallel memory reference instruction sets. This is true of the machine instructions, though the assembler frees the programmer from having to keep track of this. The same mnemonic is used in both cases.

One block of memory consists of the first 256 locations, addresses 0 through $FF. This block is called **page zero** or **scratchpad** and references to it generate a different type of binary instruction than references to any other part of memory, this difference being one of both form and length. The term page zero is not entirely appropriate here since the 6502 does not have what is commonly regarded as a true paged memory, at least as far as the programmer is concerned. The term **scratchpad** will be used through the remainder of this book. Memory reference instructions which refer to scratchpad are two words long, the first word being the instruction itself and the second the scratchpad address. Memory reference instructions which refer to memory other than scratchpad, which we will call **general memory**, are three words long, the first again being the instruction, and the last two being the address. The mnemonic used by the programmer is the same in both cases, but the binary machine instructions are different. A reference to general memory is also known as an **absolute** reference.

These two addressing modes have the common feature that the address of the memory operand is directly specified by the instruction. Memory references made in this way are called **direct** references. Direct addressing is specified in the assembly language by the sole appearance of the operand address, either symbolic or numeric, in the operand field of the instruction, e.g.:

```
LDA  SYMBOL
```

or

```
LDA  $3AF7
```

which load the A register with the contents of the word whose label is SYMBOL and the contents of memory address $3AF7 respectively.

In the **immediate** mode of addressing the operand itself, rather than an address, is contained in the instruction. In the 6502 immediate mode instructions are always two words long, the first word specifying the instruction and the second being the operand itself. This mode provides a very useful way to generate small program constants. The immediate mode is specified by the appearance of the pound sign (#) as the first character of the operand. The operand itself may be either a number, a symbol with value less than 256, or a "half symbol" specification. The half symbol specification will be discussed later in this chapter. Examples of instructions using the immediate addressing mode are:

```
LDY  #-1
```

which loads the Y register with the binary value of the decimal constant -1, and:

```
LDX  #'S'
```

which loads the X register with the ASCII code for the letter S. Immediate mode addressing is available only for load type instructions. Now let's put these

addressing modes together into something tangible. An application of the direct and immediate addressing modes is shown in example 5-1.

EXAMPLE 5-1

Use the direct and immediate addressing modes to place the constant $F3 into memory location $3C9.

```
label   inst  operand

        LDA   #$F3      this loads the constant
                        $F3 into the accumulator
        STA   $3C9      and this stores the
                        accumulator into memory
                        location $3C9.
        ...
```

The immediate addressing mode was specified by the appearance of the pound sign (#) in the operand field of the LDA. The constant $F3 is contained in the instruction itself in this case. The direct mode of addressing was specified by the absence of # or other special symbol in the operand field of the STA. Remember that prefixing a number with a dollar sign ($) specifies that number to be hexadecimal.

A slightly more complex situation is shown in example 5-2.

EXAMPLE 5-2

Use immediate and direct addressing to place the

binary constant 10111101 into location BCON and the ASCII code for the character "+" (plus) into the location following BCON.

```
label   inst    operand

        LDA     #%10111101      the pound sign causes
                                immediate addressing and
                                the percent prefix means
                                that the number is binary.
        STA     BCON            store it in BCON.
        LDA     #'+'            enclosing the character
                                in apostrophes means that
                                it is ASCII.
        STA     BCON+1          BCON+1 means the next word
                                in memory after BCON.
        ...
```

A memory addressing mode which becomes very important in handling tables of data is the **indexed** mode. Indexed addressing resembles direct addressing in that an address is given in the instruction. It differs from direct addressing in that the address in the instruction is modified by the addition of the contents of one of two eight bit registers, the X and Y registers. The memory address which is finally accessed by an instruction using this mode is the sum of the address given in the instruction and the contents of the X or Y register. The final address which results from this computation is known as the **effective address**.

Indexed addressing provides a powerful method for accessing arrays and tables of data in the 6502, as will become clear in the chapters to come. It is specified by appending the character X or Y, separated by a comma, to the operand, e.g.:

```
LDA  BERT,X
```

or

```
LDA  ERNIE,Y
```

When the X and Y registers are used in this way they are known as **indexing registers** or **index registers.** Only a simple example of indexed addressing will given here. It is shown in example 5-3.

EXAMPLE 5-3

Use indexed addressing to move the third word after location GROVER to the seventh location following OSCAR. This done by first loading the required displacements into the index registers X and Y, then using these indices to load A with the required word and store it into the target location.

```
label   inst    operand
        LDX     #3          loads X with 3
        LDY     #7          loads Y with 7
        LDA     GROVER,X    accumulator is loaded
                            with the third word
                            following GROVER
        STA     OSCAR,Y     and stored into the
                            seventh word following
                            OSCAR
        ...
```

The power of indexed addressing lies in its ability to address an entire block of data with the same instruction by simply modifying the index register. It will appear many times in the chapters to

come.

The 6502 is unusual among current microprocessors in having facilities for performing **indirect addressing** through pointers held in memory. As discussed previously, a pointer is a sixteen bit number which is the address of a memory location. Since a 6502 memory word is eight bits long, a sixteen bit pointer is held in two successive memory words, the low significance half in the word with the lower address. Memory pointers in the 6502 are restricted to scratchpad, i.e., they must be contained entirely within the first 256 words of memory. To illustrate the use of a pointer in scratchpad, suppose locations $50 and $51 contained $A6 and $3B respectively. These two words are said to contain a pointer to location $3BA6. An indirect access to location $50 accesses location $3BA6.

The only 6502 instruction which is capable of pure indirect addressing is JMP, meaning jump, an unconditional program branch. For all other memory reference instructions the indirect and indexed modes are combined. The combined addressing mode works in one of two ways, depending on which of the index registers, X or Y, is being used. This is of necessity somewhat complex, so bear with it as it is necessary to understand these addressing modes.

Recall that the X and Y index registers perform exactly the same function when using simple indexed addressing. Their functions in that case were interchangeable. This is not so when the compound indexed-indirect mode is used. We will take up the use of X as the first case.

When X is used as the index in an indirect access, an event known as **preindexing** takes place. In preindexing the value contained in the index register is added to the nominal address given in the instruction. This sum, the instruction address plus the value in the X register, forms the address of the indirect pointer. Recalling that a pointer is two words long, this generated address and the one

following it are used as a pointer. The contents of the memory location pointed to by this pair of words is then accessed. This kind of addressing is of strictly limited usefulness.

A far more useful compound mode, using the Y register, is known as **postimdexing**. In postindexing, the address given in the instruction is taken as the address of a pointer to whose contents the number in the Y register is added to form the final effective address. This postindexing capability is one of the 6502's most powerful assets. It allows a pointer to be created in the scratchpad area which points to the base of a table. The successive entries in the table can then be accessed by manipulating the value in the Y register. The 6502 would be a significantly more powerful device than it is if both of its index registers functioned in the postindex mode.

With regard to the assembly language format of indirectly addressed instructions, there has in the past been a practice of using parentheses to indicate indirection, e.g.:

```
LDA  (PTR,X)
```

indicating the preindexed mode with X, and:

```
LDA  (PTR),Y
```

indicating the postindex mode with Y.

This notation has been rejected here as a piece of fundamentally bad language design. The pre or post index condition is specified completely by the X or the Y, given that indirection has already been specified by the first parenthesis. The second parenthesis is therefore superfluous. What is worse, it imposes the duty upon the programmer of remembering where the second parenthesis goes. The programmer is thus, in a way, obliged to prove to the assembly program that he knows the difference each time he attempts to use indirect addressing. This is, of course, a hideous

nuisance and an unnecessary and unreasonable burden on the user.

To clear this obstacle out of the way, therefore, another method of specifying indirect addressing, much used in the past, has been resorted to. In the examples in this book indirect addressing will be indicated by prefixing an asterisk to the operand field for both pre and post indexing, e.g.:

```
LDA  *PTR,X
```

indicates indirect preindexed, while:

```
LDA  *PTR,Y
```

indicates indirect postindexed, the pre or post condition being completely specified by the X or Y. For those who need to use the parenthesis notation, the assembler will accept and correctly assemble this notation. It processes only the left parenthesis and the actual index character X or Y however. No note is taken of the right parenthesis. It may be placed anywhere or omitted.

CHAPTER 6

6502 ARITHMETIC AND LOGICAL OPERATIONS

The 6502 is a binary twos complement computer. It has a limited decimal arithmetic capability as well, which will be examined in chapter 12. This chapter will examine its arithmetic and logical operations as applied to binary quantities.

The principal arithmetic register of the 6502 is the A register, or accumulator. Addition, subtraction and various logical operations are performed in the A register. Unlike most microprocessors the 6502 is also capable of some limited manipulation of data held in memory. The accumulator operations will be examined first.

Addition in the 6502 is performed by adding a number in memory to one which is already contained in the A register. The mnemonic for addition is ADC, meaning ADd with Carry. The contents of the 6502's carry bit are automatically added to the sum. This means that if a simple addition is to be performed the carry bit must be cleared to zero before the addition. Simple addition of two numbers is shown in example 6-1.

EXAMPLE 6-1

Two numbers are in LOC1 and LOC2. Find their sum and store it in location TOTAL.

```
label   inst    operand
SUM     LDA     LOC1        load first number into A
        CLC                 this clears the carry bit to
                            zero. This instruction needs
                            no operand.
```

```
ADC    LOC2     form the sum in the A
                register. Note that LOC2 in
                memory is not disturbed.
STA    TOTAL    store the sum in TOTAL. The
                sum is now in the A register
                and in location TOTAL.
...
```

Note that it was necessary to clear the carry bit before addition. Failing to do so would have done no harm if carry had been zero before the addition, but would have resulted in a sum too large by one if carry had been a one. This kind of simple addition is an extremely common operation in a program of any size, so the requirement for clearing carry before any addition can become a burdensome clerical duty. This very feature and its counterpart in subtraction have been the basis of some considerable criticism of the 6502 in popular literature. The assembly program can provide easy relief from this nuisance. If the mnemonic **ADD** is written instead of **ADC** the assembler will automatically include the necessary CLC instruction. The use of ADD is shown in example 6-2.

EXAMPLE 6-2

Find the sum of the numbers in LOC1 and LOC2 as in example 6-1, but use the ADD instruction instead of CLC and ADC. Leave the sum in TOTAL.

```
label  inst   operand

ADAM   LDA    LOC1     get first number from LOC1
       ADD    LOC2     form sum
       STA    TOTAL    save result in TOTAL
```

...

It is important to understand here that ADD does not do anything that you could not do for yourself. It is not a 6502 machine instruction, but a convenience provided by the assembly system. It assembles to exactly the same thing as:

```
CLC
ADC   LOC2
```

Subtraction is accomplished by the SBC instruction, meaning SuBtract with Carry. In the case of subtraction the carry bit is used as a borrow bit. For simple subtraction, the nature of 6502 arithmetic requires that the carry bit be set to one before the subtraction takes place. If the carry bit is zero the difference will be too small by one. In subtraction the number to be subtracted is known as the **subtrahend** and the number from which the subtrahend is to be subtracted is known as the **minuend**. The result of a subtraction is called the **difference**. A simple 6502 subtraction is shown in example 6-3.

EXAMPLE 6-3

Two numbers are in locations BOYS and GIRLS respectively. Using BOYS as the minuend, find the difference between BOYS and GIRLS. Save the difference in VIVE.

```
label  inst   operand

       LDA    BOYS     minuend to A register
       SEC             SEC sets the carry bit to 1
       SBC    GIRLS    form difference
```

```
        STA   VIVE     save difference
        ...
```

The location named VIVE must, of course, be defined elsewhere in the program by its appearance as a label. A space can be reserved in a number of ways. For ease of reading of the program, the assembler provides a pseudo-instruction mnemonic consisting of three asterisks. This assembles to a word of all zeros. It provides a convenient way to flag locations whose contents will be changed during program execution. Location VIVE may thus be defined by:

```
VIVE    ***            1a difference
```

As with ADC the need to set carry before an ordinary subtraction imposes an uncommon writing burden on the programmer. The assembler therefore provides a means of avoiding this. Use of the mnemonic SUB causes the assembler to produce both SEC and SBC. The use of SUB is shown in example 6-4.

EXAMPLE 6-4

Solve the same problem as in example 6-3, but use the assembler instruction SUB instead of the SEC - SBC sequence.

```
label  inst   operand

       LDA    BOYS     minuend to A register
       SUB    GIRLS    form difference
       STA    VIVE     save result
```

...

As with ADD, SUB does nothing for you that you could not do for yourself, it simply saves writing. As you become more familiar with the 6502 you will come to appreciate the convenience. A number of these compound instructions will be introduced as the need arises, each of them representing a commonly used standard series of machine instructions.

We have briefly discussed the carry bit in this and earlier chapters. The 6502 maintains a register each of whose bits represents some CPU condition. The carry bit is one of these. The register is known as the **processor status word**. The bits of the processor status word have the individual designations N, V, B, D, I, Z, and C, the eighth bit being unused. The meaning of the individual bits is as follows:

N - Set to one if the result of the last operation left a one bit in the high position of the affected register or memory word. This is the case when a numerical operation produces a negative result. The N bit is also known as the sign flag.

V - Set to one if the result of the last operation was a number too large to be held in a signed 6502 word, i.e., a result greater than decimal 127 or less than decimal -128. The exact mechanics of this operation are disussed later in this chapter. The V bit is also called the **overflow flag**.

B - Deferred to later discussion.

D - This flag is set by the SED and CLD instructions of the 6502. It is associated with the performance of decimal arithmetic and will be covered in chapter 12.

I - This flag is associated with the interrupt system. It will be discussed in chapter 18.

Z - This flag is set to a one if the previous operation produced a zero result, and to zero if the operation produced a nonzero result. Don't let this apparent contradiction bother you. You will very seldom have to deal directly with the Z flag. It will be interpreted for you by the 6502's branch instructions.

C - The carry bit as previously discussed.

The bits of the processor status word are set by the outcome of arithmetic, logical and other operations. They may be tested and the flow of the program altered by the **conditional branch** instructions. To this point we have discussed program execution as a serial execution of instructions located one after the other in ever higher memory addresses. This serial execution can be modified by means of a class of instructions known as **jumps** or **branches**. Though this class of instruction does not transfer data in the way that normal memory reference instructions do, they are considered to be part of the memory reference class. In this context it may help to think of this instruction group as being a sort of load instruction. The program counter is what is loaded with a new value rather than one of the operating registers.

The 6502 has only one jump instruction, JMP. The JMP occupies three words of memory, one for the instruction and two for the address specification. With a full 16 bit address the JMP instruction can reach anywhere in 6502 memory. JMP uses a full 16 bit address even if its target is in scratchpad. Its form is:

JMP TARGET

where TARGET is the address to which control is to be transferred. JMP can also be indirectly addressed, but without indexing. It is the only 6502 instruction that can do this. The indirect form of JMP looks like this:

```
JMP  *TPTR
```

in which TPTR is a 16 bit pointer in memory which contains the address of the location to which control is to be transferred. Note that this pointer can be held anywhere in memory, unlike the pointers for other memory reference instructions which can be held only in scratchpad. We have said nothing so far about how these pointers come into existence. This will be covered a little later in this chapter.

The 6502 branch instructions form their target addresses in a completely different way. The branch instructions are two words long. The first word specifies the instruction and the second a signed twos complement displacement. The addressing scheme works like this. Remember the heavy stress laid upon the value of the program counter during instruction execution in chapter 1. The program counter always points to the instruction following the one being executed. The displacement portion of a branch is computed by subtracting the value of the program counter during the branch execution from the target address. Since the target address may be either higher or lower than the value of the program counter, the result of the subtraction may be either positive or negative, a number between -128 and +127 inclusive. The range of a branch instruction is therefore limited to these values. At attempt to branch out of range will be flagged by the assembler as an error. Suppose you wrote:

```
ORG  $3000
BCC  $4000
```

When you attempt to assemble the program which contained this the assembler would list the line like this:

```
ddd 3000 9000                BCC  $4000
    A
```

where ddd is the decimal line number. The A is the error flag, meaning addressing range error.

The branch instructions of the 6502 are all conditional, i.e., the branch is only executed if some specified condition is met. If the condition is not met the branch is ignored and control passes to the instruction which follows the branch. The condition for the branch is specified in the 6502 program status word. Each of the branch instructions tests one of the status word bits. The 6502's branch instructions and their conditions are:

BCC - Branch if the carry bit is zero (Carry Clear).

BCS - Branch if the carry bit is one (Carry Set).

BEQ - Branch if equal. The branch occurs if the last operation set the Z bit to one. The Z bit is set to one when the result is zero.

BNE - Branch if not equal. The branch occurs if the last operation set the Z bit to zero, i.e., if the result of the last operation was <u>not</u> zero.

BPL - Branch if positive. The branch occurs if the last operation produced a positive result. This branch is used after operations with signed numbers.

BMI - Branch if negative. The branch occurs if the

last operation produced a negative result, i.e., a one in the sign bit. This branch is used after operations with signed numbers.

BVS - Branch if overflow set. The branch is taken if the last operation produced a result too large to be held in a signed word.

BVC - Branch if overflow clear. The branch is taken if the last operation produced a result which can be represented in a signed eight bit word.

As a convenience, a number of synonyms are provided for the above branches. As we will see later in the chapter, branch instructions are used after comparison operations as well as after arithmetic and logical operations. After arithmetic and logical operations it is more natural to think of testing for the zero or nonzero condition rather than equality or nonequality. The following synonyms are therefore recognized by the assembly program:

BZ - Branch if the result was zero. This assembles as BEQ.

BNZ - Branch if the result was nonzero. This assembles as BNE.

The above mnemonics will be used hereafter for tests after arithmetic operations.

The comparison of two quantities is a very common operation in computer programs. It can be done by subtracting one quantity from the other and examining the sign and magnitude of the result via the condition bits. This kind of comparison is necessary in some small computers without hardware to help. The method is clumsy in that it destroys the minuend in the A register, requiring reloading if any further operation

with the minuend is required. It is shown in example 6-5.

EXAMPLE 6-5

Two signed eight bit numbers are located in JIMMY and FRITZ. Compare the two numbers by subtraction and send control to JBGR if JIMMY is larger, to FBGR if FRITZ is larger, and to EQUAL if JIMMY and FRITZ are equal.

```
label   inst   operand
CMPARE  LDA    JIMMY    load JIMMY into A
        SUB    FRITZ    find JIMMY - FRITZ
        BZ     EQUAL    if JIMMY and FRITZ were
                        equal the result of the
                        subtraction is zero.
        BPL    JBGR     if JIMMY was greater than
                        FRITZ the difference is
                        positive.
        BMI    FBGR     if FRITZ was greater than
                        JIMMY the result is negative.
        ...
```

Comparison of eight bit unsigned magnitudes is a bit different, in that different flags must be tested to determine the outcome. Such a comparison is shown in example 6-6.

EXAMPLE 6-6

A pair of eight bit unsigned numbers is in locations ALPHA and BETA. Compare the two by

subtraction, sending control to ALFGTR if ALPHA is greater, BETGTR if BETA is greater and SAME if ALPHA and BETA are equal.

```
label   inst    operand

        LDA     ALPHA   ALPHA to A register
        SUB     BETA    form ALPHA - BETA
        BEQ     SAME    this tests the Z bit
        BCS     ALFGTR  carry/borrow is one if the
                        minuend was greater
        BCC     BETGTR  carry/borrow is zero if the
                        subtrahend was greater
        ...
```

If this seems confusing, it is, but don't let it discourage you. A better way to do this will emerge in a moment.

The 6502 and most other modern computers provide an alternate and much more convenient way to do comparison, a hardware compare instruction. When a compare instruction is executed the operation performed is identical to the one shown in examples 6-5 and 6-6, i.e., a subtraction is done, with a difference being generated and the status bits set to reflect the outcome. The only difference between comparison and subtraction is that the machine's A register is not disturbed. To relieve the programmer of the clerical nuisance of memorizing the settings of the bits in the status word the assembler has been written to include branch synomyms for use after comparison operations. After comparison of signed numbers the branch mnemonics are:

B>= - Branch if greater than or equal. The branch is executed if the number held in the register was greater than or equal to the one in memory. B>=

assembles as BPL.

B< - Branch if less. The branch is executed if the number in the register was less than the one in memory. B< assembles as BMI.

After comparison of unsigned magnitudes the branch mnemonics are:

BGE - Branch if greater than or equal. The branch is executed if the unsigned magnitude held in the register was greater than or equal to the one held in memory. Assembles as BCS.

BLS - Branch if less. The branch is executed if the unsigned magnitude in the register is less than the one in memory. Assembles as BCC.

Remember that an unsigned magnitude does not mean that the sign bit is vacant or zero. It only means that the highest bit is regarded as simply an eighth magnitude bit instead of an algebraic sign.

Using these branch mnemonics and the compare instruction, whose mnemonic is CMP, the comparison of example 6-5 is shown in example 6-7.

EXAMPLE 6-7

The signed numbers of example 6-5 are to be compared using the CMP instruction and control transferred using the branch mnemonics described above.

```
label   inst    operand

CMPARE  LDA     JIMMY    JIMMY to A register
        CMP     FRITZ    compare to FRITZ
        BEQ     EQUAL    JIMMY = FRITZ
        B<      FBGR     JIMMY < FRITZ
        B>=     JBGR     JIMMY > FRITZ
        ...
```

Before leaving this example, there is something important to note about it. The BEQ test was done first and there is a reason for this. The number 0, while it has no sign in the algebraic sense, is considered to be positive by the computer because its leftmost bit is a zero. The test for zero had to come before the test for greater than or equal to zero. The case of the unsigned numbers of example 6-6 is shown in example 6-8.

EXAMPLE 6-8

Solve the problem posed in example 6-6, but use the CMP instruction and the branch synonyms for unsigned comparison.

```
label   inst    operand

        LDA     ALPHA    ALPHA to A register
        CMP     BETA     compare to BETA
        BEQ     SAME     ALPHA = BETA
        BGE     ALFGTR   ALPHA > BETA
        BLS     BETGTR   ALPHA < BETA
        ...
```

BEQ and BNE are used after both signed and unsigned comparisons. The branch synonyms used after a signed comparison contain algebraic symbols, <, >, and =. Branch synonyms used after unsigned comparisons contain only alphabetic characters.

The contents of the X and Y registers may be compared to memory with the CPX and CPY instructions. These instructions work exactly the same way that CMP does, but the contents of X or Y are compared rather than those of A. While the compare instructions as a group perform an actual subtraction, carry/borrow is not included as in a normal subtraction, so it is not necessary to set carry before a compare. Also, compare instructions never affect the overflow bit. Only the N, Z and C flags are changed.

Besides the ADC and SBC instructions of the 6502 and their assembler enhancements ADD and SUB, the 6502 has instructions for performing logical operations between memory and the A register. They are:

ORA - Forms the inclusive OR (V) of the contents of the A register and the contents of the memory location named in the operand field. The result is left in A. Memory is not disturbed.

EOR - Forms the exclusive OR (¥) of the contents of the A register and the contents of the memory location named in the operand field. The result is left in A. Memory is not disturbed.

AND - Forms the AND or logical product (Λ) of the contents of the A register and the memory location named in the operand, the result being left in A. Memory is not disturbed.

The use of ORA is shown in example 6-9.

EXAMPLE 6-9

A number less than ten is in location BCD. This number is to have its high four bits set to 1101, The result being stored in ASCII.

```
label   inst    operand

        LDA     BCD         get BCD number
        ORA     #%10110000  OR on the ASCII code bits
        STA     ASCII       save the result
        ...
```

The ORA #%10110000 performed the inclusive OR operation (logical sum) on the contents of A and the binary number 10110000. The pound sign (#) specified immediate mode addressing and the percent sign (%) specified that the number which followed was binary. The operand could as easily been specified as hexadecimal, i.e.:

```
ORA   #$B0
```

or octal:

```
ORA   #@260
```

or decimal:

```
ORA   #176
```

or ASCII:

```
ORA   #'0'
```

with the same effect. The assembler will process

all of these forms to the same thing. Use of the prefixes $ and @ for hexadecimal and octal, and the apostrophes for ASCII allow the programmer to choose the most convenient form. If there is no prefix the assembler assumes that the operand is decimal.

Another very common use of ORA is shown in example 6-10.

EXAMPLE 6-10

Four signed numbers are in locations UN, DEUX, TROIS and QUATRE. If any of these numbers is negative control is to be sent to NEG. Otherwise send control to ALLPOS.

```
label   inst    operand

PTEST   LDA     UN        get first number
        ORA     DEUX      OR with second
        ORA     TROIS     OR with third
        ORA     QUATRE    OR with last number
        BMI     NEG       if any of the numbers was
                          negative the sign of the A
                          register will be one here.
        BPL     ALLPOS    otherwise go to ALLPOS
        ...
```

A procedure like the one in example 6-10 could be used for any bit position, not just the sign. For example, a binary number is odd if its low bit position is a one. The procedure above could be used to test whether any of the numbers was odd by testing the low

bit after ORing them all together. There are a number of ways to do this. Perhaps the simplest way to do this is to move the bit to be tested into the carry bit and then test for a one in the carry with a BCS instruction. The low bit can be moved into carry with a Rotate Right A instruction, i.e.:

```
RORA
```

To test for odd the BMI instruction would be replaced with:

```
RORA
BCS   ODD
```

The use of EOR is shown in example 6-11.

EXAMPLE 6-11

Two readings of an eight bit switch register are in locations OLD and NEW. Determine whether any of the switch values has changed, either from one to zero or zero to one. If none has changed send control to RETURN. If any change has taken place determine which bit position changed, leaving this number, zero through seven, in the Y register.

```
label  inst   operand
CTEST  LDA    OLD      get old register value
       EOR    NEW      form logical difference
                       with new register value. At
                       this point the A register is
                       zero if and only if NEW and
                       OLD were identical. This can
                       be tested by BZ.
       BZ     RETURN   control goes to RETURN if NEW
```

```
                        and OLD are identical.  If
                        not we must find out which
                        bit position changed.  This
                        bit position will contain a
                        one.
        LDY   #0        clear the Y register
MOVEIT  RORA            and rotate a bit into carry.
        BCS   FOUND     if carry is a one this is the
                        bit.  If carry is a zero we
                        increment the count in Y and
                        try again.
        INY             this adds one to Y
        JMP   MOVEIT    go back and try again
        ...
```

There are two things about example 6-11 which must be noted before going on. First, if more than one of the bit positions had changed only the rightmost would have been detected. Second, the JMP is a three word instruction. While it works, the thoughtful programmer will take advantage of an economy possible here. The Y register was originally loaded with zero (LDY #0). Incrementing Y with the INY instruction caused it to become nonzero. Like addition and subtraction, incrementation causes the flags to be set in the 6502's processor status word. Since the Y register is now nonzero and the flags reflect this fact, the two word BNZ instruction could have been used instead of the JMP.

The AND instruction has many uses, one of the simplest of which is shown in example 6-12.

EXAMPLE 6-12

A numeric ASCII character is in location CODE. In ASCII code the low four bits of the numeric

digits contain the binary value of the decimal digit that the ASCII character represents. Isolate the low four bits by masking with an AND and leave the result in location BIN.

```
label   inst    operand

        LDA     CODE    get ASCII character
        AND     #$F     mask out high four bits
        STA     BIN     and save result.
        ...
```

Immediate mode addressing was used with the AND to form the logical product of the contents of A with the constant. The constant itself was expressed in hexadecimal here, though any of the other forms could have been used. Note that it was not necessary to specify the constant as $0F, though it would have done no harm to do so. In general it is never necessary to include leading zeros when dealing with the assembly language. Another common use of the AND function depends upon the fact that if any of a string of bits ANDed together is a zero, the result will be a zero. This finds use in situations like that shown in example 6-13.

EXAMPLE 6-13

Signed binary numbers are in memory locations ERSTE, ZWEITE, DRITTE, und VIERTE. Use the AND function to determine whether any of them is positive. If one or more is positive send control to PLOC.

```
label   inst   operand

PTEST   LDA    ERSTE    load first number
        AND    ZWEITE   AND with second
        AND    DRITTE   AND with third
        AND    VIERTE   AND with fourth
        BPL    PLOC     control to PLOC if positive
        ...
```

The instruction INY, meaning INcrement Y, was introduced in example 6-11 as a means of keeping a count of shifts. INY is one of four 6502 instructions which increment and decrement the X and Y registers. While these instructions perform arithmetic functions they differ from addition and subtraction in that they never affect the carry and overflow bits. The register increments and decrements set only the N bit (sign flag) and the Z bit. The four instructions are:

INY - Increment the Y register

DEY - Decrement the Y register

INX - Increment the X register

DEX - Decrement the X register

Remember that it is only possible to test the status word for the zero or negative condition, and only immediately after one of these instructions. Execution of other instructions before the test may alter the Z or N flags.

Much of the information presented in this chapter must of necessity seem fragmentary and unrelated. There is no way to avoid this, since some elementary

material must be presented in pieces before it can be integrated into something meaningful. The time has now come to bring these fragments together, particularly the contents of this chapter and chapter 5. The two examples which follow will be somewhat difficult to follow, but understanding them will serve as a good test of comprehension of what has gone before.

EXAMPLE 6-14

A series of twenty numbers is located in a block of memory whose lowest address has the label BLOCK. Using indexed addressing, find the sum of the twenty numbers. If an overflow occurs send control to location TOOBIG, otherwise store the sum in location BSUM.

This exercise will require that you understand the role of an index register in the determination of the effective indexed address and the use of a counter to control the number of executions of a block of instructions. This last technique is known as looping. Understanding loops is a fundamental part of programming. A block of memory is reserved by the assembly program by the use of the RES pseudo operation. A pseudo operation is written in the instruction field of the source program, but is not a machine instruction. It is a directive to the assembly program. The block of twenty locations required above can be reserved by:

```
BLOCK  RES   20
```

the RES in the instruction field meaning REServe. The size of the block is specified in the operand field. The symbol in the label field, BLOCK in this case, has as its value the lowest memory

address of the block. The twenty word block thus occupies memory from BLOCK to BLOCK+19. The program to find the sum looks like this:

```
label   inst    operand

BTOTAL  LDY     #19       Y is loaded with 19 so that
                          the sum of the contents of Y
                          and the nominal address given
                          in the memory reference inst-
                          ruction will result in an
                          effective address of BLOCK+19,
                          i.e., the address of the 20th
                          member of the block.
        CLA               set A to zero
LOOP    ADD     BLOCK,Y   following the operand BLOCK
                          with Y separated by a comma
                          specifies that the contents of
                          the Y register are added to
                          the address BLOCK. This sum
                          forms the effective address.
        BVS     TOOBIG    check for overflow!
        DEY               decrement Y. This means that
                          the next time the ADD is
                          executed the effective
                          address will be BLOCK+18. Y
                          also serves as a counter to
                          control the number of times
                          the loop is executed. When
                          Y becomes negative the loop
                          has been executed 20 times,
                          i.e., all 20 numbers have
                          been added. Since DEY sets
                          the N flag we can test for
                          this condition with the BPL
                          instruction.
        BPL     LOOP      when the 20th execution of
                          the DEY causes Y to become
                          negative the BPL will not
                          branch, instead sending
```

```
                    control to the STA below.
STA   BSUM          save result
...
```

While example 6-14 used Y as the index, X could have been used in exactly the same way. If indirect addressing is not used X and Y may be used interchangeably.

Indirect addressing was introduced in chapter 5 and the need for pointers held in scratchpad discussed. No means for creating these pointers was specified, however, and this gap must be filled before the next example can be undertaken.

A pointer is a 16 bit quantity held in two adjacent memory words, the low significance half of the pointer in the lower memory address. We speak of a pointer held, for example, in location $68. It must be understood that this pointer actually occupies both locations $68 and $69. The 6502 contains no facility for handling any kind of 16 bit quantity except in two 8 bit halves. The assembler provides a facility to allow this. Prefixing the operand of an immediate mode instruction with "<" or ">" allows a number or symbol to be treated in halves. The number -587 is too large to be held in 8 bits, but such a constant can be created and stored into memory by the use of these operators. Four instructions are required to do this, i.e.:

```
LDA   #<-587      get low half of value
STA   DBLCON      save in low constant
LDA   #>-587      get high half of value
STA   DBLCON+1    save in high constant
```

The use of this method is not limited to constants. A symbol may be used as the operand to create double length pointers, i.e.:

```
LDA   #<SYMVAL
STA   SPDLOC
LDA   #>SYMVAL
STA   SPDLOC+1
```

in which SYMVAL is the address of any memory location and SPDLOC is any memory location. In doing this we have created a pointer at SPDLOC and SPDLOC+1 which points to location SYMVAL.

In application programs creation of pointers is one of the most frequently performed operations. Requiring, as it does, four instructions, the repeated creation of pointers becomes very tedious because of the amount of writing involved. The assembly program, therefore, provides a shorthand means for doing this. As with all of the compound instructions, this one does nothing for you that you could not do for yourself - it just saves writing. A constant or pointer can be created by the CDB (Create Double) pseudo operation, with two operands separated by commas. To create the constant -587 at location DBLCON, write:

```
CDB   DBLCON,-587
```

To create a pointer at SPDLOC pointing to SYMVAL, write:

```
CDB   SPDLOC,SYMVAL
```

CDB assembles to exactly the sequence shown above. Pointers and constants may be created anywhere in memory, not just scratchpad. The only exception to this is that no pointer or constant may be created across the scratchpad boundary, i.e., with the low half in $FF and the high half in $100. Attempting to do this will result in a B (Boundary) error flag from the assembler. Note here that since CDB uses the A register to load and store the created constant or

pointer, the contents of A are destroyed. If the A contents are required they may be saved with an STA and reloaded with an LDA after the CDB. There is a much better way to do this, as we shall see when we study the stack pointer. With this information in hand we can go on to example 6-15.

EXAMPLE 6-15

Solve the problem of example 6-14 using indirect addressing through a scratchpad pointer. Index Y is to be used, and perform the addition from lowest to highest address instead of highest to lowest.

```
label   inst    operand
BTALLY  CDB     PTR,BLOCK   this creates a pointer to
                            BLOCK at scratchpad location
                            PTR, as just described

        CLA                 zero A, as before
        TAY                 Transfer A to Y. This is a
                            one word instruction which
                            copies the contents of A into
                            Y. LDY #0 will also work, but
                            requires two words. TYA
                            performs the inverse
                            operation.
LOOP    ADD     *PTR,Y      the asterisk specifies indir-
                            ect addressing. The effective
                            address equals the contents
                            of pointer PTR plus those of
                            the Y register.
        BVS     TOOBIG      check for overflow
        INY                 increment the index
        CPY     #20         index 20 yet?
        BNE     LOOP        if not do it again
```

```
        STA   BSUM     save result
        ...
```

Unlike the simple indexed program in example 6-14, X and Y are not interchangeable when indirect addressing is involved. Use of X in this case would not have worked, since X preindexes rather than post-indexes as Y does. Preindexing is an addressing mode of very limited use. An example of its use will appear later in the text.

The 6502 also allows memory to be incremented and decremented directly, an unusual feature in current microprocessors. This feature opens the possibility of an almost unlimited number of counters, a powerful plus for this computer in some situations. Like their X and Y register counterparts, these instructions set only the N and Z flags of the status word, never carry or overflow. Memory incrementation and decrementation are done by:

INC - Increment the memory location named in the operand. INC is two words long if the operand is in scratchpad, otherwise it is three words long.

DEC - Decrement the memory location named in the operand. DEC occupies two or three words as for INC above.

The 6502 has no native hardware instructions for incrementing and decrementing the A register. A can be incremented by:

```
        CLC
        ADC  #1
```

and decremented by:

```
SEC
SBC  #1
```

The lack of single hardware instructions to increment and decrement A seems to have caused distress in some circles, particularly among trade magazine editors and other priestly persons. To alleviate this distress and to address a number of other complaints from this source, the assembler has been written to include the compound instructions:

INA - Increment the A register

DEA - Decrement the A register

These mnemonics assemble to the sequences shown above. Note that, unlike the X and Y register increments and decrements, INA and DEA will set overflow and carry, as well as N and Z. In response to complaints that the 6502 has no instructions for clearing A or taking the ones or twos complement of A, the assembler also supports the following compound instructions:

CLA - Clears A to zero. Assembles as:

```
LDA  #0
```

CMA- Ones complements A. Assembles as:

```
EOR  #$FF
```

TCA - Twos complements A. Assembles as:

```
EOR  #$FF
CLC
ADC  #1
```

CLA and CMA will affect only the N and Z bits, but TCA involves an addition which sets all of the arithmetic condition bits.

The final topic of this chapter is 6502 shifts. We have already introduced the RORA instruction in examples in this chapter. Shifts may be performed on the A register or memory. Right and left A register shifts are performed by:

```
RORA      right rotate with carry
```

and

```
ROLA      left rotate with carry
```

In both cases the bit rotated out enters the carry bit and the former contents of the carry bit enter the position vacated by the shift. Shifts of data held in memory are done by:

```
ROR  LOC  rotate memory right with carry
```

and

```
ROL  LOC  rotate memory left with carry
```

where LOC is a memory address. Logical shifts are performed in A by:

```
ASLA  logical left shift A
```

and

```
LSRA  logical right shift A
```

and in memory by:

```
ASL  LOC   logical left shift memory
```

and

```
LSR  LOC   logical right shift memory
```

where LOC is an operand address. Note that the shift instructions alter the contents of carry, as well as the N and Z flags. The 6502 has no arithmetic shift instructions.

CHAPTER 7

DOUBLE AND MULTIPLE PRECISION METHODS

Like most current microprocessors, the 6502 has an eight bit word length, capable of representing data to a precision of about .4 percent. This precision is unusable for any but the most primitive computational tasks and will certainly not serve general application needs. Ways must therefore be found to extend the range and precision of 6502 arithmetic.

This is done by representing numbers in two or more words. Numbers so represented are called **multiple precision numbers**. When the number requires exactly two words it is said to be **double precision**. The purpose of this chapter is to examine methods by which arithmetic may be performed on multiple precision numbers.

Multiple precision numbers are represented in the 6502 as a series of words, the least significant occupying the lowest memory address. The double precision constant 1025 in binary is:

00000100 00000001

it being the usual practice to write the high significance half on the left. In the computer, however, we usually represent a string of words from lowest to highest address causing this representation to be reversed to:

00000001 00000100

The instruction JMP 1025 thus assembles to the three words:

01001100 00000001 00000100

or 4C 01 04 in hexadecimal, the 4C being the JMP instruction. This rule holds for all multiple

precision quantities, whether constants or addresses.

A signed multiple precision number uses only the high bit of the highest order word for the sign bit. The seven low bits of the high order word and all eight bits of all lower order words represent the magnitude of the number. In double precision this means a sign and fifteen magnitude bits, giving a range of +32767 to -32768.

Double precision constants can be created in the source program with the DBL pseudo-op, e.g.:

```
DELTA  DBL  1023
```

creates the 16 bit number 00000011 11111111, the low order 11111111 being stored at DELTA and the high order 00000011 at DELTA+1. DBL will accept all of the data forms noted in example 6-9 with prefixes $ for hexadecimal, @ for octal and % for binary. DBL will also accept a symbol as an operand, allowing it to be used to create pointers, e.g.:

```
JPNTR  DBL  JTABLE
```

will create a pointer at locations JPNTR and JPNTR+1 which points to JTABLE. DBL will also accept multiple operands separated by commas, e.g.:

```
DSTRNG  DBL  -1,$43E7,@7734,RLOC-1
```

creates four double precision quantities. The label attached to the DBL, if any, is associated with the low order half of the first double precision quantity in the string of operands.

In performing multiple precision arithmetic it is useful to think of each word in the multiple precision number as a kind of digit in base 256. Just as we generate a carry into the next higher order column when the sum of two decimal digits exceeds 9, we generate a carry into the next higher order "digit" of the multiple precision number when the sum exceeds 255.

This carry must be included when the higher orders are added. There is no carry into the lowest order column, there having been no previous addition to create one. The two low order words of the sum are thus generated by simple addition, the second and all higher orders being added to include carry from the orders previously added. A simple double precision addition is shown in example 7-1.

EXAMPLE 7-1

Two double precision numbers are located at TINKER and EVERS. Find their double precision sum and leave it in CHANCE.

```
label   inst    operand

DBLSUM  LDA     TINKER    get low order TINKER
        ADD     EVERS     form low order sum
        STA     CHANCE    save low order result. At
                          this point the carry bit is a
                          one if the low order sum was
                          greater than 255.
        LDA     TINKER+1  get high order TINKER
        ADC     EVERS+1   form high order sum
                          plus carry from low sum
        STA     CHANCE+1  save high order result
        ...
```

The low order addition was done with ADD because we didn't want the accidental contents of carry to be included. After the low order sum had been formed, however, carry contained the actual value of carry out of the low order. Since this must be included in the high order sum ADC was used to form the high order sum.

The double word CHANCE in example 7-1 was left undefined in the example, presuming it to have been defined elsewhere in the program. CHANCE is a program location which is altered during the course of program execution. Room for CHANCE could be created with the DBL pseudo-op, i.e.:

```
CHANCE DBL 0
```

which will do the job, but leaves an ambiguity. When the program you have written is re-examined six months afterward you may well wonder whether the DBL 0 means a real zero constant or a temporary working location as we have in example 7-1. The assembler furnishes a convenient way around this, the double analog of the *** operation described in example 6-3. Writing:

```
CHANCE  $$$
```

creates the necessary doubleword and flags CHANCE as a temporary location whose contents will change during execution.

Double precision subtraction is equally simple. The low order subtrahend is subtracted from the low order minuend, ignoring the previous state of carry/borrow. The higher orders are then subtracted including any borrow generated by the low order. On the 6502 the borrow state is zero, not one. This is the opposite of the convention for most computers. This is why it is necessary to <u>set</u> carry before a simple subtraction. If you are used to another computer keep this inversion in mind or it will cause trouble. A simple double precision subtraction is shown in example 7-2.

EXAMPLE 7-2

A double precision minuend is located at DARTH and

a double precision subtrahend at VADER. Find the double precision difference and save it at EMPIRE.

```
label   inst    operand

DSUB    LDA     DARTH     load low order minuend
        SUB     VADER     form low order difference
        STA     EMPIRE    save low difference
        LDA     DARTH+1   load high order minuend
        SBC     VADER+1   the high order subtraction
                          is done with SBC to include
                          any borrow from the low order
        STA     EMPIRE+1  save the high order
                          difference
        ...
```

Triple and higher precisions are a simple matter of extension of the methods used for double precision. The higher orders are all added and subtracted to include the carry or borrow from the lower orders. A triple precision subtraction is shown in example 7-3.

EXAMPLE 7-3

Triple precision numbers are located at LISA and PAUL, LISA being the minuend and PAUL the subtrahend. Find the triple precision difference and store it at RUTH.

```
label   inst    operand

TDIFF   LDA     LISA      low minuend to A
        SUB     PAUL      form low difference
        STA     RUTH      save low result
        LDA     LISA+1    get 2nd order minuend
        SBC     PAUL+1    form 2nd order difference
```

```
        STA   RUTH+1   save 2nd order result
        LDA   LISA+2   get 3rd order minuend
        SBC   PAUL+2   form 3rd order difference
        STA   RUTH+2   save 3rd order result
        ...
```

The assembler provides no triple precision analog of *** or $$$. The best way to reserve space for a triple precision result like RUTH above is:

```
RUTH    RES   3
```

which reserves the necessary space.

A generalized N-precision addition or subtraction is easily done with a simple program loop. The case of addition is shown in example 7-4.

EXAMPLE 7-4

Two numbers of unknown but equal precision are located in memory beginning at BIGN1 and BIGN2. The number of orders which comprises BIGN1 and BIGN2 is in a memory location named N. Find the N-precision sum of BIGN1 and BIGN2 and leave it at BIGN3.

```
label   inst  operand

BIGADD  LDY   #0        clear Y for use as an index
        CLC             clear carry before add loop
ADLOOP  LDA   BIGN1,Y   load augend order BIGN1+Y
        ADC   BIGN2,Y   add addend order BIGN2+Y
        STA   BIGN3,Y   save sum order BIGN3+Y
        INY             increment Y to point to next
                        order
```

```
        DEC   N         decrement order count. When
                        N is zero all orders have
                        been added.
        BNZ   ADLOOP    test for zero and repeat if
                        necessary
        ...
```

N-precision subtraction will be left to the reader to try as an exercise. It is substantially the same as the N-precision addition above, and trying it will be a good test of your understanding of multiple precision. Remember that carry/borrow must be set to one before the first subtraction.

It is often necessary to add a one word addend to a multiword augend. The addition is simple enough, just add the one word addend to the low word of the augend and propagate any carry to the higher orders. A simple way to do this is shown in example 7-5.

EXAMPLE 7-5

A single precision addend is located at KARL and a double precision augend at ERIC. Find their sum and leave it at ERIC.

```
label   inst    operand

        LDA     KARL    get single addend
        ADD     ERIC    form sum
        STA     ERIC    save low result. The carry
                        produced by this sum must
                        now be added to the high
                        order of the double
                        precision number ERIC. One
                        way to do this is with INC.
        BCC     NXT     skip if carry zero
```

```
        INC   ERIC+1   otherwise increment high
                       order ERIC
NXT     ...            program continues
```

An alternate way to propagate the carry in the example just above is load the high operand into A and execute:

```
ADC  #0
```

This has the effect of adding the immediate operand, a zero, plus the current value of carry to A. The high augend must then be stored back into memory.

The examples up to this point have involved loops in which the count was a small number. This will often not be the case, however, and for loop counts greater than 255 it becomes necessary to use double precision counts. This can be done either by incrementing a negative number toward zero or decrementing a positive number toward zero. Incrementing a negative double precision number toward zero is by far the simpler of the two. The process is shown in example 7-6.

EXAMPLE 7-6

A double precision number is in locations DBLCNT and DBLCNT+1. Increment the number.

```
label   inst   operand

DINC    INC    DBLCNT     this increments the low order
                          half of DBLCNT.
        BNZ    DINC2      explained below
        INC    DBLCNT+1   increment high half if
                            necessary
DINC2   ...
```

After the INC DBLCNT we need some means of telling whether a carry should be propagated into the high half in DBLCNT+1. Since INC does not affect carry we would seem to be at a dead end, but there is a solution. There is one, and only one, circumstance in which incrementing could produce a carry. This is when the number was 11111111 to begin with. If, and only if, this was the case would adding 1 to it produce a carry. This is easy enough to see. If there were a zero anywhere in the word adding one to the word would produce carries only up until the zero. The chain of carries would then be broken and no carry out of the high bit would result. If the previous content of the word was 11111111 incrementing it would produce 00000000. Testing whether the incremented number is zero is therefore the same as testing for carry in this case. The BNZ instruction makes this test and skips over the second increment if the result of the first was not zero.

Further, after this double increment the zero flag of the status word reflects the outcome of the double increment in every case. If, and only if, both halves of the number are zero after the double increment is the Z flag set. It can then be tested by BNZ or BZ.

The operation in example 7-6 is so common that the assembler provides a compound instruction to allow it to be written on one line instead of three. A double precision number or pointer may be incremented by writing:

```
IDB  LOC
```

where LOC is the address of the low order half of the

quantity to be incremented, i.e., the one with the lower memory address. The operand of the IDB may be anywhere in memory except across the scratchpad boundary.

Decrementing a double precision quantity is not nearly so simple. The problem is that the high half must be decremented when the low half is decremented to 11111111, and there is no way to test for this condition in memory. The test must be performed in a register. The process is shown in example 7-7.

EXAMPLE 7-7

A double precision number is located at BIGNUM and BIGNUM+1. Decrement the number.

```
label   inst   operand

DBLDEC  LDY    BIGNUM     get low half
        DEY               decrement it
        CPY    #$FF       test for 11111111
        BNE    DD2        skip if low half = 11111111
        DEC    BIGNUM+1   otherwise decrement high
                          half
DD2     STY    BIGNUM     replace decremented low half
                          in memory
        ...
```

Shifting of multiple precision numbers is required in multiplication by constants, as we will shortly see. To shift right we begin with a logical shift of the leftmost word. This brings a zero into the vacated bit position and puts the bit shifted out of the leftmost word into carry. This bit is in its turn rotated into the next word to the right, the rotation bringing the bit from the leftmost word into the second and rotating

a bit out of the second word. This process continues for as many orders as are required. A double precision right logical shift is shown in example 7-8.

EXAMPLE 7-8

A double precision number is in HARVEY and HARVEY+1. Perform a double logical right shift on the number.

```
label   inst    operand

DRSHF   LSR     HARVEY+1    shift high order right.
                            This puts a zero in the high
                            bit and the former low bit
                            into carry.
        ROR     HARVEY      the content of carry is
                            rotated into the high bit of
                            the low order word. The bit
                            rotated out of the low order
                            word enters carry.
        ...
```

The effect of this is to shift both halves to the right with the former low bit of the high half being transferred to the high bit of the low half.

A double logical left shift is as simple as its right shift counterpart in example 7-8. This shift is shown in example 7-9.

EXAMPLE 7-9

A double precision number is in RALPH and RALPH+1.

Perform a logical left shift on this number.

```
label   inst    operand

        ASL     RALPH     shift low half. This brings
                          in a zero on the right and
                          puts the high bit into carry.
        ROL     RALPH+1   rotate high half. This
                          brings carry into the low bit
                          and puts the former high bit
                          into carry.
        ...               shift completed.
```

Double precision arithmetic shifts are another matter entirely. The 6502 has no arithmetic shift instructions at all. There is the problem, therefore, of propagating the sign into the vacated space for a right shift and preserving the sign in a left shift. The right shift is fairly simple. It is shown in example 7-10.

EXAMPLE 7-10

A signed double precision number is at locations DBLNUM and DBLNUM+1. Perform an arithmetic right shift on this number.

```
label   inst    operand

DARSHF  LDA     DBLNUM+1  get high order
        ASLA              move sign into carry bit
        LDA     DBLNUM+1  get high order again. The
                          sign of A and the carry bit
                          now agree.
        RORA              rotate carry into sign and
                          low bit of high order into
```

```
                    carry.
        ROR   DBLNUM    rotate carry into high bit
                        of low order and low bit of
                        low order into carry.
        STA   DBLNUM+1  restore high order in memory
        ...
```

Note that the bit which was shifted out of the low order enters carry, where it is available for rounding if required.

An arithmetic left shift is more difficult in that the sign must be preserved. This is done by keeping a copy of the original sign and forcing the sign of the result to agree with it. The process is shown in example 7-11.

EXAMPLE 7-11

A signed double precision number is in memory at X and X+1. Perform an arithmetic left shift on the number.

```
label   inst   operand

DALSHF  LDA    X+1      get high order
        AND    #$80     isolate sign. This masks out
                        everything but the sign bit.
        STA    SIGN     save sign
        LDA    X+1      get high order again
        ASL    X        shift low order
        ROLA            rotate carry into high order
        AND    #$7F     mask out high order bit
        ORA    SIGN     and force sign to original
                        value
        ...
```

This will perform the double left shift according to the specification, but something important is missing and must be addressed. If the bit shifted from the high magnitude position of the high order word into the sign bit was different from the original sign the highest significant bit was lost, a numerical catastrophe. Some computers provide hardware to guard against this very problem. The M6800, for example, sets overflow if this occurs. It would be an excellent exercise for the reader to modify this example to include a check for this condition. There are a number of ways to do this. Perhaps the easiest is to form the exclusive OR (¥) of the saved sign with the shifted high order word before the masking operation. If the A register is negative immediately after this exclusive OR, then the bit shifted into the sign was different from the bit shifted out.

Before leaving this example, note the use of the symbol X as the operand of a memory reference instruction. This is prohibited by most 6502 assemblers, since X is the name of a machine register. A, Y, S, and P are also usually forbidden. This assembler contains no such reserved symbols. If you use the name of a machine register as a symbol it will be interpreted correctly. Within the rules for symbol formation you may use any name at all as a label.

Mention was made earlier in this chapter about incrementing a negative double length counter toward zero rather decrementing a positive counter toward zero. This presumes that we know how to negate a double precision number. This is not a difficult process if we remember the definition of twos complementation. The number is first ones complemented and then incremented. The process is shown in example 7-12.

EXAMPLE 7-12

A double precision number is located at JEKYLL. Its twos complement is to be stored at HYDE, but the contents of JEKYLL are not to be disturbed.

```
label   inst    operand

        LDA     JEKYLL      get low order JEKYLL
        CMA                 ones complement it
        STA     HYDE        move to low order HYDE
        LDA     JEKYLL+1    get high order JEKYLL
        CMA                 ones complement it
        STA     HYDE+1      move to high order HYDE
        IDB     HYDE        double increment HYDE
        ...
```

Before leaving this example it is worth noting what the use of the pseudo-ops CMA and IDB saved. If the above had been written with only native 6502 machine instructions it would have looked like this:

```
        LDA     JEKYLL
        EOR     #$FF
        STA     HYDE
        LDA     JEKYLL+1
        EOR     #$FF
        STA     HYDE+1
        INC     HYDE
        BNE     LOC
        INC     HYDE+1
LOC     ...             program continues
```

The pseudo ops allow considerable saving in the writing that must be done.

DOUBLE AND MULTIPLE PRECISION METHODS

At various points in the discussion of memory reference instructions in this and previous chapters, mention has been made of different length requirements of these instruction. The point has been reached where we must be more specific about this. Memory reference instructions which address operands in scratchpad require, in general, only two words. Those which address operands in general memory require three words. The two word scratchpad addressing format is not only shorter but faster than the three word general memory format. The location of the operands can have a gross effect on both the space required by a program and the speed with which it executes. This becomes particularly important in a machine like the 6502 which is used so widely as a controller. It is thus necessary for a programmer to know how the assembler processes operands and determines the required addressing mode.

From the point of view of the assembler the source program is a series of statements, only one of which can be seen by it at any time. The program is scanned beginning with the first line and proceeding until a line containing "END" in its instruction field is encountered. This means that when the assembler encounters a memory reference instruction whose operand has not yet been defined it must make an assumption about the location of the operand, either scratchpad or general memory. The practice in the past has been to assume that the operand is in general memory and use the three word form of the instruction. If the eventual definition of the operand shows it to be in scratchpad the high order address of the instruction is set to zero. This results in an instruction which will work but which wastes one word of memory and is slower.

The problem has been solved in most previous 6502 assembly systems by forcing the programmer to define all of his scratchpad references at the beginning of the program, before any instructions. This allows the assembler to assume that any operand not defined by the time a reference to it occurs is a general memory

reference.

Some unknown philosopher is supposed to have remarked that "There is no problem so big that it can't be run away from", and this seems to have been the case with 6502 assemblers to date. The purpose of a programming system is to spare the programmer the clerical work of keeping track of details like this. The assembler in use here, therefore, has been constructed to do this clerical work automatically.

The ORG pseudo op has already been introduced. It is a means of telling the assembler where the instructions and other data which follow are eventually to be located in memory. Writing:

```
ORG   $2B00
LDA   TIGER
```

insures that the first word of the LDA instruction will be located exactly at $2B00. The ORG furnishes a means of precisely locating instructions or data in memory.

Scratchpad data are similarly located using the SPD (ScratchPad Data) pseudo op. SPD and its companion pseudo op ESP (End ScratchPad) are used to bracket data which are to be located in scratchpad. Data so located will <u>always</u> be addressed by the shorter, faster two word scratchpad instruction format. SPD is written with an operand whose value is less than 256. Following the SPD and before the ESP are the data definitions which are to be considered scratchpad for addressing purposes. The SPD-ESP pair may occur anywhere in the program, and any number of of SPD-ESP pairs may be used. No instructions are allowed between an ESP and an SPD, only data reserving operations. Some of these operations have been introduced earlier. The full set of data reserving pseudo ops is:

RES - Reserves a memory block whose size is specified by the operand. If a label is used with RES the label identifies the lowest memory address of the block.

```
FRENCH RES   $14
```

reserves a twenty word block. The label FRENCH is associated with the lowest address of the block.

DATA - Generates a single data word for each of its operands. DATA may have multiple operands separated by commas, e.g.:

```
PIERRE DATA  1,$E3,@376,%11000001,'A'
```

generates five words in the object program, decimal 1, hexadecimal E3, octal 376, binary 11000001, and the ASCII code for the letter A. If the operand string runs into the comment field the comment field will be moved right to accomodate it automatically.

DBL - Generates a doubleword for each of its operands. DBL may have multiple operands separated by commas as for DATA above. In addition it may have symbols as operands, e.g.:

```
MARIE  DBL   PROG1,PROG2,PROG3,PROG4
```

generates a table of pointers to PROG1, PROG2, etc. The comment field will be moved as for DATA above.

ASC - Generates a string of ASCII character codes for its single operand, a string of characters enclosed in apostrophes, e.g.:

```
BMSG   ASC   'FREE BEER!'
```

generates 10 words, with the label value equal to the location of the F in the string. The string may not contain apostrophes. An apostrophe code can be generated by:

```
DATA  $A7
```

ASCII data generated in this way have the high bits set to one. The comment field is adjusted as for DATA above.

*** - Generates one word of zero in the object program under the name given in the label, if any. The line:

```
AGNES  ***
```

generates one word of zero under the name AGNES.

$$$ - Generates one doubleword of zero under the label given, if any. Writing:

```
PAUL   $$$
```

generates two words of zero under the label PAUL.

EQU - Sets the value of the symbol in the label field to the value of the operand field. Anything in the operand field of an EQU must either be numeric or ASCII, or a symbol previously defined in an SPD-ESP range. The line:

```
RPAREN EQU   ')'
```

sets the value of RPAREN to the ASCII value of the character ")", allowing future references to take the form:

```
        LDA   #RPAREN
```

which loads the ASCII code for ")" into A. EQU is a powerful programming aid and we will be studying more of its many uses in the chapters to come.

REM - A documentation device to allow alignment of comments. If the comment field of an instruction or pseudo op line is not large enough to contain the necessary text, following the line with a REM allows the comment to continue on the next line with its left border aligned with the comment field of the instruction. Using REMs makes a neat legible listing. The REM in the instruction field does not appear.

TITL - If TITL is used anywhere in a program its operand field will appear as a page heading in the assembly listing, appearing just to the right of the page number.

In addition, any line which begins with an asterisk (*) as the first character of the label field will be considered to be all comment. It is printed in the assembly listing but has no effect whatever on the assembly.

The above data defining pseudo ops may appear anywhere in a program, but only these pseudo ops may appear within the range of an SPD, i.e., bracketed by an SPD-ESP pair. Appearance of a label in the range of an SPD insures that references to that label will generate the optimal two word instruction type. This is extremely important in applications which are critical with respect to memory size or speed. The greatest performance can be gotten from the 6502 only by the skillful use of scratchpad memory.

DOUBLE AND MULTIPLE PRECISION METHODS

Obviously the greatest economies can be made by placing the data locations most often referred to in scratchpad. Just which data items are most frequently referred to is often unknown when a large programming project is begun. This is only a temporary nuisance. The reference frequencies can be easily estimated after the program is finished and the data most often referred to moved into scratchpad. One caution needs to be observed when doing this. Moving an item to scratchpad not only shortens the instructions which refer to it, it makes those references faster. It follows then that data which are referred to as part of a timing loop should be moved only with great care. The timing loop may have to be rewritten after the move.

CHAPTER 8

SOFTWARE MULTIPLICATION AND DIVISION

Like most microprocessors, the 6502 has no hardware facility for performance of multiplication and division. The responsibility for these functions falls on the programmer. The purpose of this chapter is to present methods for performing multiplication and division for a number of special cases and the general case.

We begin with what is certainly one of the oldest tricks in the programmer's book, a multiplication by ten performed by shifting and adding. The need for multiplication by ten or some other special constant multiplier is very common. The process is shown in example 8-1.

EXAMPLE 8-1

A binary number is contained in the low four bits of the A register, the high four bits being zero. Multiply the number by ten leaving the result in the A register

```
label   inst    operand

BYTEN   ASLA            multiply by two
        STA     TEMP    save intermediate result
        ASLA            multiply by two again. The
                        original number has now been
                        multiplied by four.
        ASLA            and again. The original
                        number has now been multi-
                        plied by eight.
        ADD     TEMP    now add the intermediate
                        result
        ...
```

This works because for any number N:

$$10N = 8N + 2N$$

The sharp-eyed reader may have noticed an economy that could have been made in this example. Since the upper four bits of A were guaranteed zero by the conditions of the problem, carry is guaranteed to be zero after the third ASLA. It is therefore unnecessary to clear it again as the ADD instruction does. One program word could have been saved by using ADC instead of ADD.

Multiplication by a constant ten arises in the conversion of decimal data to binary as we shall see in chapter 14.

It is possible to construct a sequence of shifts and adds or subtracts to multiply by any constant number. Consider the multiplication by thirteen in example 8-2.

EXAMPLE 8-2

The number in memory location FRIDAY is to be multiplied by thirteen. Return the result to location FRIDAY. Since 13N = 16N - 2N - N, the procedure is straightforward. We find 2N with a single shift and store it. The number is then shifted three more times for a total of four shifts. Four shifts multiplies by sixteen. 2N and the original number N are then subtracted from this to yield 13N, i.e.:

```
label   inst   operand

TIMZ13  LDA    FRIDAY   get number
```

```
        ASLA            multiply by two
        STA     TEMP    save it
        ASLA            times four...
        ASLA            times eight...
        ASLA            times sixteen
        SUB     TEMP    subtract 2N saved above
        SUB     FRIDAY  subtract N, the original
                        number
        STA     FRIDAY  and save result
        ...
```

Note that in this case SBC could not have been used in place of SUB. Since the bits shifted out are zeros and carry must be set before simple subtraction there is no way around using SUB or the SEC-SBC sequence.

This process is easily extended to multiple precision. The exact mechanics are a bit different since the shifting is done in memory, but the principle is the same. A double precision multiply by a constant is shown in example 8-3.

EXAMPLE 8-3

The number in ALPHA and ALPHA+1 is to be multiplied by 35. Perform the multiplication and leave the result at ALPHA and ALPHA+1.

```
label   inst    operand

        LDA     ALPHA   save original number
        STA     TEM1    ...
        LDA     ALPHA+1
        STA     TEM1+1
        LDA     ALPHA   multiply original number
        ASLA            by two and save it
```

```
        STA   TEM2     ...
        LDA   ALPHA+1
        ROLA           roll in low order bit
        STA   TEM2+1
        ASL   ALPHA    the original number is now
        ROL   ALPHA+1  shifted left five times to
        ASL   ALPHA    multiply by 32
        ROL   ALPHA+1  ...
        ASL   ALPHA
        ROL   ALPHA+1
        ASL   ALPHA
        ROL   ALPHA+1
        ASL   ALPHA
        ROL   ALPHA+1
        LDA   ALPHA    now add original number
        ADD   TEM1
        STA   ALPHA
        LDA   ALPHA+1
        ADC   TEM1+1   propagate carry
        STA   ALPHA+1
        LDA   ALPHA    now add original number
        ADD   TEM2     times two
        STA   ALPHA
        LDA   ALPHA+1
        ADC   TEM2+1   propagate carry
        STA   ALPHA+1
        ...

Where TEM1 and TEM2 are defined elsewhere as:

TEM1    $$$
TEM2    $$$
```

While this process is tedious and long, it is faster, if not shorter, than a full multiply. The method of shifting and adding or subtracting can be extended to cover multiplication by numbers which contain fractional parts like pi, 3.141592653... and

the base of natural logarithms e, 2.7182818... To do this you must first decide how much precision you need and then select a fractional approximation to the constant which has a denominator that is an exact power of two. In the case of pi there are a number of choices. For fairly low precision work the fraction:

$$201/64 = 3.140625$$

will work. If more precision is needed the fraction:

$$3217/1024 = 3.1416015$$

is available. The point here is that the indicated division need never be done. By keeping the denominators of the fractions exact powers of two the division is reduced to a shift. For 201/64 the number is multiplied by 201 by the shift and add process and then shifted six places right to divide by 64. For the 3217/1024 approximation the number is multiplied by 3217 and the result shifted right ten places to divide by 1024. For the number e the approximations:

$$87/32 = 2.71875$$

and

$$5567/2048 = 2.718261719$$

will serve. This will give you the idea of how it is done. A few minutes with a calculator will yield a satisfactory approximation to any required constant.

In general multiplication the two numbers to be multiplied are known as the **multiplier** and **multiplicand**. The single precision multiplicand is multiplied by the single precision multiplier to yield a double precision result known as the **product**. The product of two single precision numbers is a double precision number since in general the product of an N bit number and an M bit number is N + M bits in length.

SOFTWARE MULTIPLICATION AND DIVISION

For an understanding of what goes on in machine multiplication it is instructive to multiply a pair of binary numbers on paper by the common rules of arithmetic. Consider the multiplication of 7 times 5 or 0111 times 0101 in binary. Begin by writing one number below the other as in decimal multiplication:

```
 0111   the multiplicand
 0101   the multiplier
```

Just as in decimal multiplication the digits of the multiplier are successively tested and the multiplicand multiplied by the digit value, this partial product being written below, indented so that the lowest digit of the partial product is aligned with the multiplier digit which produced it. This is much simpler in binary than in decimal since there is no multiplication table to memorize. If the multiplier bit is a zero it is ignored. If the multiplier bit is a one the multiplicand is simply copied into the partial product line, suitably indented. The first step of our binary multiplication looks like this:

```
 0111
 0101
 0111
```

the 0111 being copied into the partial product because the rightmost multiplier bit was a one. The next multiplier bit is tested and the result is:

```
 0111
 0101
 0111
0000
```

this line of the partial product being zero because the twos bit of the multiplier was zero. The fours bit of the multiplier is now tested with the result:

```
  0111
  0101
  0111
 0000
0111
```

Finally, the eights bit of the multiplier is now tested and the result is:

```
  0111
  0101
  0111
 0000
0111
0000
0100011
```

Machine multiplication is the exact analog of this process. The multiplier in memory is shifted to place successive multiplier bits into carry where they can be tested. If the multiplier bit is a one the multiplicand is added to the low product. The double precision product is then shifted, this shifting performing the same function as the indentation of the partial product lines in the paper multiplication above.

In the multiplication exercise above we began testing multiplier bits on the right, proceeding from right to left as is conventional. There is no particular reason for this. The process could be carried out from left to right beginning with the high multiplier bit. The partial product lines would then be indented to the right. You can easily verify this on paper by executing the same multiplication as above. A 6502 program to perform multiplication of two eight

bit unsigned numbers is shown in example 8-4.

EXAMPLE 8-4

Two numbers are in locations MPLR and MCND. Find their product and leave the low half of the product in LOPROD and the high half in HIPROD.

```
label   inst    operand
MPY     LDY     #8          number of multiplier bits
        CLA                 clear A to hold low product
MPY1    ASLA                shift double product left
        ROL     HIPROD      ...
        ASL     MPLR        multiplier bit to carry
        BCC     MPY2        skip if multiplier bit zero
        ADD     MCND        add multiplicand to low
                            product
        BCC     MPY2        check carry from low to
                            high product
        INC     HIPROD      increment high product
MPY2    DEY                 decrement multiplier bit
                            count
        BNZ     MPY1        and repeat if not zero
        STA     LOPROD      save low product
        ...
```

There are a couple of things to be noted about this example before passing on to another topic. First, notice that the low product held in A was cleared before the loop beginning at MPY1 was entered but the high product held in memory was not. This is because the eight successive left shifts of the double product clear the former contents of HIPROD. The clearing of HIPROD was therefore unnecessary.

Second, a significant saving in time could have

been made by placing MPLR, MCND and the product in scratchpad. While the precise execution time of the program in example 8-4 depends on the number of one bits in the multiplier, for any given bit configuration the program will execute about 11% faster with the operands in scratchpad than in general memory. This is done by placing the definitions of the four memory operands in the range of an SPD-ESP pseudo op pair, like this:

```
        SPD   $80      set scratchpad location
MPLR    ***
MCND    ***
LOPROD  ***
HIPROD  ***
        ESP
        ...
```

Multiplication of signed numbers is only a little more elaborate than this. The simplest way to multiply signed numbers is to first determine the sign of the product by exclusive ORing the multiplier and multiplicand as described on page 11. This result is saved and both multiplier and multiplicand are then forced positive. The multiplication is performed on these positive operands. When the product is found the saved sign indicator is tested and the double precision product negated if necessary. The multiplication of two signed numbers is shown in example 8-5.

EXAMPLE 8-5

A pair of signed twos complement binary numbers are in locations MPLR and MCND. Find their double precision product and leave it, properly signed, in LOPROD and HIPROD.

```
label   inst   operand

SMPY    LDA    MPLR     get multiplier
        EOR    MCND     logical difference with
                        multiplicand.  The sign bit
                        of this result is the same as
                        the sign bit of the product.
        STA    SIGN     save sign
        LDA    MPLR     get multiplier
        BPL    SM2      skip if already positive
        TCA             otherwise force positive
        STA    MPLR     save positive multiplier
SM2     LDA    MCND     get multiplicand
        BPL    SM3      skip if already positive
        TCA             otherwise force positive
        STA    MCND     save positive multiplicand
SM3     LDY    #8       get multiply step count
        CLA             clear low product
SM4     ASLA            shift low product
        ROL    HIPROD   and high product
        ASL    MPLR     multiplier bit to carry
        BCC    SM5      skip if zero
        ADD    MCND     otherwise add multiplicand
        BCC    SM5      skip if no carry to HIPROD
        INC    HIPROD   otherwise propagate carry
SM5     DEY             decrement loop count
        BNZ    SM4      repeat if not zero
        STA    LOPROD   save low product
        LDA    SIGN     test for negative product
        BPL    SM6      ...
        LDA    LOPROD   must negate.  Ones complement
        CMA             low order
        STA    LOPROD   ...
        LDA    HIPROD    ones complement high order
        CMA             ...
        STA    HIPROD
        IDB    LOPROD   increment the double
                        precision result
SM6     ...             program continues
```

Notice that though the twos complement operation,

TCA, appeared twice in this example, in neither case was any check made for overflow afterward, even though the number being twos complemented could very well have been the maximum negative number. This can do no harm in this case, but it would be instructive for the reader to figure out why this was permissible here. Clue: the multiplication itself is done on unsigned eight bit magnitudes. Also note that the program assumes that HIPROD is in the memory location immediately following LOPROD. This can be assured by defining them as follows:

```
LOPROD ***
HIPROD ***
```

either in scratchpad or general memory. The same speed considerations apply as in example 8-4.

Another way to accomplish the necessary definition is with the EQU pseudo op, like this:

```
LOPROD $$$
HIPROD EQU   LOPROD+1
```

The use of $$$ in the instruction field saves the necessary doubleword. The EQU generates no program words but simply defines the symbol HIPROD to mean the location following LOPROD, wherever LOPROD happens to be located. The only caution in the use of EQU is that anything in its operand field must be either self defining, i.e., numeric or ASCII, or be defined at a point in the program previous to its use as the operand of an EQU. We will see much more of this useful pseudo op as the book proceeds.

Multiplication of double precision requires that the multiplier and multiplicand be considered as the

sum of their respective high and low orders. If the multiplier is A+B and the multipliand C+D, then the product of these is:

$$(A+B)(C+D) = AC + AD + BC + BD$$

Each of these products is, however, a double precision number itself and the result of multiplying 16 bits by 16 bits is a 32 bit product, i.e., a quadruple precision number. If the high and low halves of each of the products is represented by prefixing it with H or L, then the four orders of the quadruple precision product are found by adding the partial products like this:

4th order	3rd order	2nd order	1st order
HAC	LAC		
	HAD	LAD	
	HBC	LBC	
		HBD	LBD

This sum is formed beginning with the first or lowest order, the carries from each order being propagated into the next higher order. If we were to attempt an example of this using only the tools developed so far in the book the example would run for several pages. For this reason an example of double precision multiplication will be deferred until chapter 9 when the required techniques for reducing its size have been developed.

Division is the process by which all possible multiples of a number called the **divisor** are removed from a number called the **dividend**. The number of times the divisor can be removed from the dividend without the result crossing zero is known as the **quotient**. The residual result, i.e., the number left after all possible multiples of the divisor have been removed from the dividend, is called the **remainder**.

In pencil and paper division there is no restriction on the sizes of the numbers involved, e.g.,

if more quotient digits are required one can, if necessary, simply tape an extra piece of paper to the edge of the current one and keep going. In computer division, however, there are restrictions on the sizes of the numbers involved.

The dividend, that number which is to be divided by the divisor, is considered to be a double precision number, and the divisor a single precision number. Further, dividend and divisor are required to be of such relative magnitude that the resulting quotient will be containable in a single computer word. If this is not the case a **divide fault** condition is said to exist. This does not mean that the division cannot be done. It only means that it cannot be done with a single divide operation.

To understand how machine division works, consider a division on an imaginary four bit machine:

```
      ________
0110 )00100011
```

i.e. 35/6. In pencil and paper division we do not automatically select any particular number of digits of the dividend for our first divide attempt, but scan to the right until we have a group of digits at least as large as the divisor. In this case the group would be the leftmost six bits of the dividend which form 001000. In machine division no such such scanning is possible. The number of digits selected for the first trial is fixed, in this case at four bits, the word length of our imaginary demonstration machine. Given that we are forced to do this the outcome is:

```
        0
      ________
0110 )00100011
     0000
       10
```

The zero product of quotient bit and divisor is then subtracted from the high four bits of the dividend. In conventional division a new bit would be "brought down"

from the dividend and the divide attempted again, with a result that looks like this:

```
      00
0110 )00100011
     0000
       100
```

Again a digit is "brought down" and the divide again attempted, this time successfully:

```
      001
0110 )00100011
     0000
       1000
       0110
        101
```

the difference, 1000 - 0110 being 10. "Bringing down" another bit makes the new partial dividend 101. This generates another zero quotient bit and the result, with the final bit "brought down" is:

```
      0010
0110 )00100011
     0000
       1000
       0110
        1011
```

This generates a one bit in the quotient and the result is:

```
      0010l
0110 )00100011
     0000
       1000
       0110
        1011
        0110
         101
```

Since there are no more bits to be "brought down" the division is finished. The resulting quotient above being 5 and the remainder also being 5.

Most of this excess transcription is dispensed with in computer division. Rather than write the product of a quotient digit and the divisor below the partial dividend and then subtract, the subtraction is done directly from the partial dividend. The "bringing down" of digits is eliminated by shifting the dividend left. As the dividend is shifted left vacant bits appear on its right. It is into these vacated dividend bits that the successively developed quotient bits are written so that the developing quotient gradually replaces the low dividend. With the subtraction being done directly from the high dividend the remainder is the high dividend after the final subtraction.

This process is, admittedly, sore tedious. It is much easier to follow on a machine like the Z80 which has 16 bit registers in which the entire dividend can be shifted as a single entity. In our example, the high dividend is held in the A register and the low dividend in memory. This requires that the shift of the dividend occur in halves, which clouds the issue somewhat. The process is presented in example 8-6. If you have the fortitude to get through this example you will have a good understanding of the division process.

Before undertaking this labor there is one more facet of division to be understood, the divide fault problem mentioned before. If the division would generate a quotient larger than one computer word the divide cannot be done. This condition is arithmetically the same as the high half of the dividend being greater than or equal to the divisor. It is the comparison of the high dividend with the divisor in memory that constitutes the divide fault check here. You may wish to verify with a few trial dividends and divisors that this is indeed the case. It is very tempting to gloss over and skip an example like 8-6. Don't do it. If you conquer example 8-6 you have nothing to fear from the remainder of this book.

EXAMPLE 8-6

A sixteen bit dividend less than 32768 is in LODVND and HIDVND. An eight bit divisor is at DVSR. Find the quotient and remainder, leaving the quotient in LODVND and the remainder in REM. If a divide fault condition occurs send control to location DVDFLT.

```
label   inst    operand
DIVIDE  LDA     DVSR      get divisor
        BZ      DVDFLT    cannot divide by zero! Now
        LDA     HIDVND    check that high dividend is
        CMP     DVSR      less than the divisor.
        BGE     DVDFLT    divide fault if DVSR is less
        LDY     #8        divide step count to Y
        CLC               clear carry to start
DVD1    ROL     LODVND    shift dividend
        ROLA              ...
        CMP     DVSR      greater than divisor?
        BLS     DVD2      skip if not
        SBC     DVSR      otherwise subtract divisor
                          from high dividend. SBC was
                          used here because carry is
                          guaranteed to be a one. The
                          BLS instruction, remember,
                          assembles as BCC.
DVD2    DEY               decrement count
        BNZ     DVD1      repeat until zero
        ROL     LODVND    bring in last quotient bit
        STA     REM       save remainder
        ...
```

The quotient need not be moved as it has replaced the low dividend. The high dividend is undisturbed. As with the multiplication examples, a considerable economy of time is possible by locating the various operands in scratchpad. When

assigning operands to scratchpad, it is most important to get the ones most frequently referred to into the scratchpad area. This means the memory references which occur within loops such as the divide loop above.

As with multiplication, the simplest way to divide signed numbers is to determine the sign of the quotient by the exclusive OR process shown in example 8-5, then force both divisor and dividend positive. Keep in mind that the divisor is, by definition, a seven bit magnitude and the dividend a fifteen bit magnitude. The divide fault check must now compare the high eight bits of the dividend to the seven bit divisor to assure that the quotient magnitude will fit into seven bits. When the divide fault test has been passed proceed with the divide, adjusting the sign of the quotient to agree with the result of the earlier exclusive OR. The sign of the remainder must agree with the sign of the original dividend.

Division in double precision is probably best done on the 6502 by simply doubling the dimensions of the single precision divide process. The dividend is a quadruple precision number, and the divisor double precision. The divide fault check and the test for divisibility during division must be made in double precision and the dividend shift in quadruple precision. The result will be long, slow and ponderous and only good to 16 bits.

Multiple precision division is quite often associated with the manipulation of high precision fractions, such as arise in applications in astronomy and crystalography. In these applications only the quotient is required, the remainder being discarded. In these cases an approximation is usually used which gives a result accurate to almost 30 bits in quadruple precision. These applications deal with extremely precise angles, and since 21 bits will resolve an angle

to less than one second of arc, the approximation is adequate. It is based on the expansion of the expression:

$$\frac{(A + B)}{(C + D)}$$

in which A and B are the double precision halves of the quadruple precision dividend, and C and D are the corresponding parts of the divisor. This kind of a division is well beyond the scope of a book like this. If you require further information about it consult more advanced sources, e.g., Assembly Level Programming for Small Computers by W. J. Weller (Lexington Books, 1975).

CHAPTER 9

SUBROUTINES AND THE STACK POINTER

In several of the previous chapters the use of pointers in addressing memory has been discussed and illustrated. This chapter takes up the use of a very special pointer which is implemented in the 6502 hardware. This hardware feature plays a central role in 6502 programming. It is known as the **stack pointer.**

The pointers used in the earlier text, e.g., example 6-15, were 16 bits in length and held in a pair of adjacent scratchpad memory locations. In the case of example 6-15 the pointer was used to access a block of memory locations for the purpose of finding their sum. After each access the effective pointer was incremented by incrementing the Y register which modified the pointer, in this way accessing higher and higher memory addresses.

In contrast to the 16 bit pointers held in memory in the earlier examples, the stack pointer is an eight bit hardware register which holds the low eight bits of a memory address. The high eight bits are set by the hardware to 00000001. Since the hardware stack pointer containing the low eight bits of the address is eight bits long and the upper eight bits of the address are "hardwired" to 00000001, the stack pointer can contain an effective address of $100 to $1FF, i.e., it can point to any of the 256 locations which immediately follow the scratchpad. We will call this 256 word memory block the **stack area** or the **stack.** Data entered into the stack are said to be **pushed** onto the stack. Data extracted from the stack are said to be **pulled** or **popped** from the stack.

Besides having its high eight bits fixed by the 6502 hardware, the stack pointer is different in one important feature from the other pointers we have encountered in that it is self incrementing and decrementing. Pushing a datum onto the stack causes that datum to be stored into the memory location

pointed to by the stack, and then causes the stack pointer to be decremented, i.e., to point to the next lower memory location. A subsequent push stores the datum into this next lower memory location and decrements the stack pointer to point to the one next lower than this, and so forth.

Pulling a datum from the stack first increments the stack pointer to point to the next higher memory location, then accesses that location. The system of pushes and pulls exactly parallels the operation of the serving tray stacks found in cafeterias. Imagine the tray dispensing device empty. Now drop a tray onto the dispenser, then another, then another. The first tray dropped on is on the bottom, and the last one dropped on is on the top. Accessing this stack of trays one at a time, the last tray dropped onto the stack (pushed) is the first tray available from the stack, the first to be pulled. Accessing the stack through the stack pointer works exactly the same way. Data are withdrawn, i.e., pulled, from the stack in exactly the opposite order from that in which they were pushed. Such a structure has a formal name, a LIFO stack, meaning Last In First Out. There are other types of stacks but only the LIFO stack will concern us here.

The stack pointer is loaded by transferring the desired contents from the X register by means of the TXS (Transfer X to S) instruction. Example 9-1 shows this operation.

EXAMPLE 9-1

The stack pointer is to be loaded in such a way that it contains the highest possible stack address.

```
label  inst  operand

STKLD  LDX   #$FF     X is loaded with the largest
```

```
                    number containable in 8 bits.
        TXS         move X contents to stack
                    pointer
        ...
```

The 6502 allows the contents of the A register and the processor status word to be pushed and pulled. This is in sharp contrast to other microprocessors in which most machine registers can be pushed and pulled. The instructions which push and pull A and PSW are:

PHA - Causes the contents of the A register to be pushed onto the stack at the location currently pointed to by the stack pointer (SP). SP is then decremented to point to the next lower address. The contents of A are not changed.

PLA - Causes the stack pointer to be incremented and the contents of the location now pointed to by SP to be loaded into A. The contents of memory are not changed.

PHP - Causes the contents of the processor status word (PSW) to be pushed onto the stack at the location currently pointed to by SP. SP is then decremented to point to the next lower address. The contents of PSW are not changed.

PLP - Causes SP to be incremented and the contents of the location now pointed to by SP to be loaded into the processor status word, PSW. The contents of memory are not changed.

The meanings of the bits of the processor status word were detailed on pages 58 and 59. Their actual physical locations within the PSW is occasionally important, as we will see in example 9-7. The PSW layout is shown below.

PSW Bit Locations

position	7	6	5	4	3	2	1	0
contents	N	V	*	B	D	I	Z	C

the bit 5 position being unused.

The transfer of data from A to the PSW is shown in example 9-2.

EXAMPLE 9-2

The contents of the A register are to be copied into the PSW. Perform this operation using the stack.

```
label   inst    operand

ACOPY   PHA     contents of A to stack, decrement
                stack pointer.
        PLP     contents of stack location to PSW
                then increment stack pointer.
        ...
```

The contents of the A register and the PSW may be swapped by pushing them onto the stack, then pulling them back in the opposite order. This operation is shown in example 9-3.

EXAMPLE 9-3

The contents of the A register and PSW are to be exchanged by pushing and pulling from the stack.

```
label   inst    operand

RSWAP   PHA             A contents to stack,
                        decrement SP
        PHP             PSW contents to stack,
                        decrement SP
        PLA             increment SP, stack
                        contents to A register
        PLP             increment SP, stack
                        contents to PSW
        ...
```

The swap will not work in the opposite order, i.e., pushing the PSW first, then A and pulling PSW and then A. It would be a very good exercise for the reader to read the descriptions of the instructions in the 6502 reference manual and find out why.

The stack can be used as a very efficient means for addressing blocks of memory to be scanned. This works very well since it eliminates incrementing and decrementing, but you must never use this technique in a system in which an interrupt can occur. The reason for this warning will be clear when interrupts have been discussed. Use of the stack to scan a memory block is shown in example 9-4.

EXAMPLE 9-4

A 72 element character string is located beginning at $100. Using the stack to "pop" successive characters into A, count the numbers of left and right parentheses in the string. If the numbers are unequal send control to location TILT. If they are equal control should go to OK.

```
label   inst    operand

SCAN    LDX     #$FF      load stack pointer
        TXS               ...
        LDX     #0        X will count "("
        LDY     #0        Y will count ")"
        LDA     #72       set string length
        STA     COUNT     in memory
SCNLP   PLA               get character from stack
        CMP     #'('      is it "("?
        BNE     TRYRT     skip if not
        INX               if yes, increment "(" count
TRYRT   CMP     #')'      try ")"
        BNE     EOL       skip if not ")"
        INY               otherwise bump ")" count
EOL     DEC     COUNT     decrement character count
        BNZ     SCNLP     and repeat if count not zero
        STX     COUNT     move "(" count to memory
        TYA               and ")" to A register
        SUB     COUNT     form difference
        BNZ     TILT      go to TILT if not equal
        BZ      OK        otherwise OK
        ...
```

There are some important things about this example which should be understood before passing on. First, the stack pointer was loaded with $FF, but the block began at $100. Recall that SP is incremented before pulling. Incrementing $FF yields $00 and the first number to be pulled from the stack will come from this location. It is important to understand this property of the stack, i.e., that it "wraps" around when incremented or decremented through zero. Second, when the scan loop was finished we stored X into COUNT. This saved a memory location since COUNT was no longer needed to hold the loop count.

SUBROUTINES AND THE STACK POINTER

In example 8-4 a single precision multiplication program was given, multiplying an eight bit multiplicand by an eight bit multiplier to give a sixteen bit result. Multiplication is a very commonly required function, likely to be needed at a number of points in a large program. Up to this point in the book, if some function such as multiplication were required at, say, four points, as for the double precision multiplication scheme on page 115, there would be no alternative to writing out the program in example 8-4 four times. This would certainly do the job but would be terribly wasteful of memory space.

What is needed here is some means by which we can have the multiplication or other program in memory only once. At the point in the program where a multiplication was needed the program could load multiplier and multiplicand into A and X and jump to this single program for the product. Jumping to the program is certainly no problem since the JMP instruction with its 16 bit address can reach anywhere in memory. The problem is how to get back to the proper place in the program after the product is found.

This can be done in several ways. The simplest would be for the program that required the product to leave a pointer in scratchpad. The pointer would contain the address of the point in the program to which control was to be transferred after the product was found. The part of the program which required the product would then look something like this:

```
      CDB   RTP,HERE  create return pointer at RTP
      LDA   MPLR      get multiplier
      LDX   MCND      get multiplicand
      JMP   MPY       execute the multiply program
HERE  ...
```

and the multiply program would begin by storing A in MPLR and X in MCND. The program would then proceed exactly as in example 8-4. When the product had been

found control would be returned to location HERE by:

```
JMP  *RTP
```

While the above will certainly work it is clumsy and long winded, particularly when one remembers that CDB requires 8 words of instructions to create the return pointer in scratchpad. There has to be a better way, and there is.

What we require here is a kind of jump instruction that will somehow remember where it jumped from, marking the place so that a return can be made to the instruction which followed the jump. Such an instruction exists on the 6502. It is the JSR, i.e., Jump to SubRoutine instruction. JSR is a three word instruction which specifies a full 16 bit address. It can therefore reach anywhere in memory. It "marks the spot" by pushing both halves of the 16 bit program counter onto the stack. This means that the stack contains the updated program counter. This updated program counter can be retrieved from the stack and used as a jump pointer to return to the proper place. Again, it is possible to access the stack locations which contain the return pointer through the normal instruction set of the 6502, but this is not necessary.

A companion instruction to JSR is RTS. RTS pulls the 16 bit number at the top of the stack and loads it into the program counter, effectively transferring control to the instruction at that location. The whole picture is a little more complicated than this, but this will give you the general idea of how it works. Using this scheme a single copy of the multiply or other program can be kept in memory and at every point where multiplication is required a JSR to the multiplication program is used. The multiplication program, in its turn, always ends with an RTS instruction which transfers control to the instruction following the JSR. Now let's try to apply this to the case of double precision multiplication mentioned on pages 114-115. The multiplication program of example

8-4 requires a little modification. The modified version is shown in example 9-5.

EXAMPLE 9-5

Modify the multiplication program of example 8-4 so that it can be invoked by a JSR instruction and will return control to the instruction following the JSR with the low product in A and the high product in X. Upon entry to the MPY program the multiplier will be in A and the multiplicand in X.

```
label   inst   operand

MPY     STA    MPLR      save multiplier
        STX    MCND      save multiplicand
        LDY    #8        set multiply step count
        CLA              clear low product
MPY1    ASLA             shift double product left
        ROL    HIPROD    ...
        ASL    MPLR      multiplier bit to carry
        BCC    MPY2      skip if multiplier bit zero
        ADD    MCND      otherwise add multiplicand
        BCC    MPY2      check for carry
        INC    HIPROD    propagate carry
MPY2    DEY              decrement step count
        BNZ    MPY1      repeat if necessary
        LDX    HIPROD    get high product
        RTS              return to calling point
MPLR    ***              multiplier storage
MCND    ***              multiplicand storage
HIPROD  ***              high product storage
```

A program written in this form, so that it can be invoked by other programs is known as a subprogram or subroutine. A subroutine can be viewed as an extension

of the machine's instruction set in the sense that a block of instructions can be created to perform any required task and invoked by using a JSR addressed to the location at which the block begins. Creation of a subroutine means that the function which it performs need only be written once and not repeated at each point at which the function is needed. With this in hand we can now address the double precision multiplication requirement. The example below uses the notation detailed on page 115 for the high and low order parts of the various partial products.

EXAMPLE 9-6

A double precision multiplier is in memory locations A and B, high order in A, and a double precision multiplicand is in memory locations C and D, high order in C. Create a subroutine named DMPY to find the quadruple precision product of these numbers using the MPY subroutine of example 9-5. Leave the product at locations BIGPRD through BIGPRD+3, lowest order at BIGPRD.

```
label  inst  operand

DMPY   LDA   A       find product of two
       LDX   C       high orders
       JSR   MPY     A times C
       STA   LAC     save product
       STX   HAC     ...
       LDA   A       find A times D
       LDX   D       ...
       JSR   MPY     ...
       STA   LAD     save product
       STX   HAD     ...
       LDA   B       find B times C
       LDX   C       ...
       JSR   MPY     ...
```

```
        STA   LBC      save product
        STX   HBC      ...
        LDA   B        find B times D
        LDX   D        ...
        JSR   MPY      ...
        STA   BIGPRD   low order BD is low order
                       product.  The remaining three
                       high orders must now be
                       added to form the final
                       quadruple precision product.
        TXA            move HBD to A and
        LDY   #0       clear Y to receive carries.
        ADD   LBC      HBD + LBC
        BCC   DM1      skip if no carry
        INY            bump carry word
DM1     ADD   LAD      HBD + LBC + LAD
        BCC   DM2      skip if no carry
        INY            bump carry word
DM2     STA   BIGPRD+1 save 2nd order product
        TYA            move carries to A
        LDY   #0       and clear carry word again
        ADD   HBC      carries + HBC
        BCC   DM3      skip if no carry
        INY            bump carry word
DM3     ADD   HAD      carries + HBC + HAD
        BCC   DM4      skip if no carry
        INY            bump carry word
DM4     ADD   LAC      carries + HBC + HAD + LAC
        BCC   DM5      skip if no carry
        INY            bump carry word
DM5     STA   BIGPRD+2 save 3rd order
        TYA            move carries to A
        ADD   HAC      form carries + HAC
        STA   BIGPRD+3 save high (4th order)
                       product
        RTS            return control to calling
                       program.
        ...
```

As clumsy as this task is, think of trying to do it without the MPY subroutine. The MPY program of example 8-4 would have had to be written four times in line. The effect of the existence of the MPY subroutine is to increase the 6502's instruction set to include multiplication.

A subroutine may be written to fulfill any special need, e.g., to check whether a character is legitimate decimal or not. Such checks are known as **validity checks**. A validity check is shown in example 9-7.

EXAMPLE 9-7

A subroutine is to be written to check the validity of the ASCII character in A. If the character is a valid numeric digit 0 - 9, return with carry off. If it is not a valid decimal digit return with carry on. In neither case should the A register be disturbed.

```
label   inst   operand

DECHEK  CMP    #'0'     is it > or = to ASCII zero?
        BLS    NOGOOD   not if less
        CMP    #$BA     check for less than ASCII
                        character after nine
        BGE    NOGOOD   skip if > or = to $BA
        CLC             carry off for valid decimal
        RTS             return with carry off
NOGOOD  SEC             carry on for garbage
        RTS             return with carry on
```

A kind of validity check which can save a great deal of space in a large program like an assembler involves testing a character for several conditions and setting the flags of the PSW to reflect those

conditions. This kind of subroutine is usually called from a great many points within a large program and setting the flags in this way saves considerable testing after each call. With the known flag settings the calling program can do its testing briefly with BZ, BMI, etc. This kind of validity check is shown in example 9-8.

EXAMPLE 9-8

A validity check subroutine is to be written which will examine the character in A and return with the flags set as follows:

N flag on if the character is a period
V flag on if the character is a carriage return
Z flag on if the character is a blank
C flag on if the character is a comma

This exercise will be considerably aided by use of the EQU pseudo op. If the symbols N, V, Z, and C are defined:

```
label   inst    operand

N       EQU     $80
V       EQU     $40
Z       EQU     2
C       EQU     1
```

they may be referred to symbolically, without having to specifically memorize the bit position table given earlier in this chapter. Having done this the validity check subroutine looks like this:

```
VCHK    LDY     #N          period code to Y
        CMP     #'.'        was it period?
```

```
        BEQ   VRTN     yes
        LDY   #V       C.R. code to Y
        CMP   #$8D     was it carriage return?
        BEQ   VRTN     yes
        LDY   #Z       blank code to Y
        CMP   #' '     was it blank?
        BEQ   VRTN     yes
        LDY   #C       comma code to Y
        CMP   #','     was it comma?
        BEQ   VRTN     yes
        LDY   #4       character was not period,
                       carriage return, blank or
                       comma. Clear Y except for
                       bit 2. Bit 2 is explained
                       in chapter 18.
VTRN    STA   T        save character in A
        TYA            move pseudo PSW to A
        PHA            pseudo PSW to stack
        LDA   T        retrieve character
        PLP            pseudo PSW to real PSW
        RTS            return to calling program.
```

In the discussion a bit earlier in this chapter it was mentioned that the JSR-RTS operation was not quite as simple as it looked, and now we must face the complication before taking up the next topic.

JSR is an exception to the rule that the program counter always points to the instruction following the one being executed. JSR is a three word instruction which should, it would be supposed, increment PC by three to point to the instruction following the JSR. This is not quite true. JSR increments by two instead of three and the result is that the address pushed onto the stack by JSR is the address of the last word of the JSR instruction itself rather than that of the first word of the following instruction.

This causes no problem when JSR is used in conjunction with RTS, since RTS compensates for the

strange behavior of JSR by incrementing the address pulled into PC before the actual transfer of control takes place. Control then actually passes to the proper point, the first word of the instruction which follows the JSR and no damage is done. If JSR or RTS is to be used out of this context however, this peculiarity must be dealt with.

A frequent requirement in 6502 application programs is a computed jump address. This arises where some piece of data is evaluated with the next program action being determined by the result of the evaluation. A table of addresses of the point to be jumped to is kept and, on the basis of the data evaluation, an address is selected from the table. There are several ways to do this. The simplest is to assemble a dummy jump instruction like this:

```
CJMP  JMP  0
```

When the address has been selected from the table it is simply filled into the second and third words of this dummy jump instruction. When control passes to the JMP a jump to the address which has been filled in will occur. This is simple and very easy to do but it has some disadvantages, the most serious of which is that the technique will not work in ROM because the address cannot be written into memory. This is an instance of a group of techniques which go by the name of **self modifying code**. Besides not working in ROM, the use of self modifying code offends some programming purists.

Another way to acomplish the same thing is to fill in a scratchpad pointer and then jump indirectly to this pointer. This still modifies memory but in a data area rather than a program area. A better way is to push the desired address onto the stack and then execute an RTS, which pulls the address just pushed onto the stack into PC. The address pushed onto the stack, however, must be one less than the actual desired target point because RTS will increment it before the actual transfer of control, thinking to

compensate for the oddity of the JSR instruction. The entries in the jump table must thus have the form:

```
DBL  TGT-1
```

A computed jump of this kind is shown in example 9-9.

EXAMPLE 9-9

An ASCII character is in A. It is to be examined for one of the five values Q, W, E, R, or T. If it is a Q control is to be transferred to QPR, if a W control should go to WPR, etc. If the character is none of the above control should be transferred to NG.

This problem could be attacked by the brute force method of comparing the character in A successively to all of the possible legal values and branching to a special block of instructions to handle that case. We will use a more elegant solution, a table lookup. The tables are defined as:

```
label  inst  operand

CTABLE ASC   'QWERT'
JTABLE DBL   QPR-1,WPR-1,EPR-1,RPR-1,TPR-1
```

and the program to search CTABLE and look up the corresponding address in JTABLE is:

```
       LDY  #JTABLE-CTABLE this loads a value into
                           Y equivalent to the differ-
                           ence of the symbols JTABLE
                           and CTABLE, i.e., a 5.
ST1    CMP  CTABLE-1,Y remember that the actual
                       table runs from CTABLE to
                       CTABLE+4 so that the effect-
```

```
                                ive address of this instruc-
                                tion is CTABLE-1+5 or
                                CTABLE+4 on the first pass.
        BEQ     ST2             branch out of loop if equal
        DEY                     otherwise bump index
        BNZ     ST1             and do it again
        BZ      NG              a jump would do as well here
                                but since we know the Z flag
                                is set we can take advantage
                                of the shorter branch.
ST2     TYA                     Y contains the address of
                                the character relative to the
                                beginning of CTABLE, i.e., it
                                is a displacement. Since the
                                entries in JTABLE are each
                                two words long we must double
                                the displacement.
        ASLA                    AR = AR times 2
        TAY                     move back to Y. Y now
                                contains a number from two
                                to ten.
        LDA     JTABLE-1,Y      if the character was a Q
                                Y contains a two. The effec-
                                tive address of this LDA is
                                therefore JTABLE-1+2 or
                                JTABLE+1, the address of the
                                high order QPR-1.
        PHA                     push onto stack high order
                                first
        LDA     JTABLE-2,Y      fetch low order jump
                                address
        PHA                     and push it onto the stack
        RTS                     now pull the top two stack
                                words into PC and increment.
                                This takes you where you
                                want to go.
```

This business of table lookup is very important

and will be covered in greater detail in chapter 11.

Subroutine operations are not limited to a single depth, i.e., a subroutine can call other subroutines and these call still lower level subroutines. The case of a subroutine calling another subroutine is shown in example 9-10.

EXAMPLE 9-10

A subroutine is required which will convert the binary contents of the A register to two ASCII-hexadecimal digits, the high order to be returned in X and the low order in A.

The conversion itself is fairly straightforward. Each four bit group is isolated and tested. If the value of the group is less than ten the high bits 1011 are attached to it and the conversion is complete. Since there is a gap of 7 characters between "9" and "A" in the ASCII sequence, a 7 must be added to the four bit group if its value is ten or greater. To avoid duplicating the actual conversion part of the program we relegate it to another subroutine.

```
label   inst   operand

CNV2HX  PHA            save binary A register
        LSRA           shift right four bits
        LSRA           very,
        LSRA           very
        LSRA           slowly
        JSR    CNVN    convert high four bits
        TAX            save high character in X
        PLA            retrieve original number
        JSR    CNVN    convert low four bits
        RTS            return to calling program
CNVN    AND    #$F     clear high four bits
```

```
        CMP   #$A      less than 10?
        BLS   CNVN2    skip if less than 10
        ADD   #7       otherwise bias up by 7
CNVN2   ADD   #$B0     if the four bit value was
                       less than 10 this ADD has
                       the same effect as an OR. If
                       the value was 10 or more the
                       upper four bits of the char-
                       acter become 1100, resulting
                       in an alphabetic character
                       A through F.
        RTS            this RTS returns control to
                       a point in the CNV2HX sub-
                       routine, not to the point
                       which called CNV2HX.
```

The actual mechanics of the conversion are not important here. What is important is that you understand the techniques used. First the stack was used to save the binary A register. A STA would have done as well, followed by an LDA when the number was to be retrieved, but the PHA-PLA sequence is shorter. Second, the subroutine CNV2HX, itself called by some other program, in its turn called a yet lower level subroutine, CNVN. This may go on to any reasonable depth within the limits of stack capacity.

In the examples given so far the numbers upon which the subroutine operated and the results it produced were transmitted between the calling program and the subroutine in the 6502's registers. The number or numbers supplied to the subroutine by the calling program are known as the arguments. The results returned from the subroutine to the calling program are called functions. If the subroutine is simple and requires three or fewer arguments or functions to be transmitted, then the 6502's registers will suffice. If there are more than three some other means of transmitting the arguments and functions must be found.

The simplest way to do this is through scratchpad. The calling program leaves its arguments in a specified scratchpad area, and the subroutine expects to find them there. The subroutine then operates on the numbers in these locations and places its results in the scratchpad, where the calling program expects to find them. This kind of argument transmission is shown in example 9-11.

EXAMPLE 9-11

A subroutine is required which will find the sum of a variable number, less than 21, of single precision unsigned magnitudes. These numbers are in a block of memory labeled VECTOR. The number of numbers is in a scratchpad location named N. The calling program places the address of VECTOR into scratchpad location PTR. The double precision sum of the numbers is to be found and left in scratchpad location VSUM. The calling program has defined the required scratchpad locations by:

label	inst	operand	
	SPD	$5A	
PTR	$$$		filled in by program. This is an address pointer to the block to be added.
N	***		also filled in by program. This is the number of numbers to be added.
VSUM	$$$		the double precision sum is to be left here.

The calling program has filled in the pointer at PTR with:

```
        CDB   PTR,VECTOR
```

The subroutine is:

```
DBLSUM  LDY   0         clear index
        TYA             clear A for low sum
        TAX             clear X for carries
ADLOOP  ADD   *PTR,Y    add number from VECTOR
        BCC   ADL2      skip if no carry
        INX             propagate carry to X
ADL2    INY             bump index
        DEC   N         and decrement count
        BNZ   ADLOOP    repeat until N = 0
        STA   VSUM      save low sum
        STX   VSUM+1    save high sum
        RTS             return to calling program
```

Before leaving example 9-11, notice that a pointer to the block to be added was kept in scratchpad, not the block itself. Scratchpad space is too precious to waste on large blocks of data.

The final method for argument and function transmission which will be considered here is that in which a block of addresses is held in scratchpad. Each address so held functions as a pointer to an argument held elsewhere in memory. This method of transmission illustrates one of the very few uses of preindexed indirect addressing. It is shown in example 9-12.

EXAMPLE 9-12

A calling program has left the addresses of three numbers whose sum is to be found at scratchpad locations NPTR1, NPTR1+2, and NPTR1+4. Write a subroutine to find the required sum and return it in the A register. Overflow may be ignored.

```
label   inst   operand

SUBSUM  LDX    #0        clear index
        LDA    *NPTR1,X  the effective address of
                         this LDA is formed by adding
                         the contents of X to those of
                         NPTR1 and NPTR1+1. The
                         contents of this address is
                         then loaded into A.
        INX              bump index to point to next
        INX              address pointer.
        ADD    *NPTR1,X  now add second element
        INX              bump index to point to
        INX              third address pointer
        ADD    *NPTR1,X  add the third element
        RTS              return to calling program.
```

Finally, it should be noted that the usual method for transmitting arguments in microprocessors and minicomputers is to have the addresses of the arguments follow the calling instruction. On the 6502, such a subroutine call would look like this:

```
JSR   SUBR
DBL   ARG1
DBL   ARG2
DBL   ARG3
...
```

and so forth for as many arguments as exist. This can be done on the 6502 and it would be an excellent programming exercise to attempt it. As a practical matter, however, the 6502's facilities for handling argument transmission in this form are such that the programming overhead involved becomes very burdensome.

CHAPTER 10

ADDRESSING ARRAYS AND TABLES

Several references have been made in the earlier chapters to the addressing of blocks of memory. The purpose of this chapter is to detail more formal methods for addressing such memory units and to develop these methods for the 6502.

A series of memory locations with sequential addresses which contain data which are related in some way is known as an **array**. Such a structure is under some circumstances known as a **vector**. The way in which the items in the array are related is specific to the problem to be solved. There might, for example, exist a series of doublewords which contained the addresses of subroutines to be called, as in the assembler listing at the back of the book. In this circumstance the structure is referred to as a **transfer vector**. The array might contain successive values of some mathematical function like logarithms or trigonometric functions to be looked up by some indexing argument. An array like this is known as a **table**.

In the assembly language space is reserved for an array by means of the RES pseudo-op. To reserve 57 words of memory under the label FLIGHT, we write:

```
FLIGHT RES  57
```

The label FLIGHT applies to the member of the array with the lowest memory address. The use of RES in the source program causes a special record to be generated in the object program which, when encountered by the loader program, will skip 57 words in memory. Note that the 57 words are not set to zero or any other specific value, only skipped. If the array is supposed to have some initial value, then either the program must actively store this value into it or the array should be created by a different means, e.g., a DATA or ASC pseudo-op. NEVER assume that an uninitialized

memory location or group of locations contains a particular value or values when program execution begins. An assumption like this can lead to some mysterious errors. If the program initializes the memory word or words at the end of its execution, i.e., a re-initialization, the program will produce correct results the second and succeeding times but not the first. This kind of problem can be awkward to find.

Using the DATA or ASC pseudo-ops to fill a block is simple. To get a block which contains the first ten integers under the label TRIXIE write:

```
TRIXIE DATA  1,2,3,4,5,6,7,8,9,10
```

The label TRIXIE refers to the location containing the 1. The operand field of this pseudo-op is longer than normal and runs into what is normally the comment field. Don't let this bother you as the assembler has enough sense to push the comment field over to the right to compensate. To use the ASC pseudo-op to generate a block containing text under the label KENOBI write:

```
KENOBI ASC  'MAY THE FORCE BE WITH YOU'
```

Again, the label KENOBI refers to the first word of the block, in this case the one containing the ASCII code for the letter M. As with DATA above, the comment field of an ASC will be moved to the right to accomodate a long operand field like the one above.

Short data arrays are usually handled in the 6502 by means of the index registers. In simple indexed addressing the effective address of the memory reference instruction is formed by adding the contents of X or Y to the address given in the instruction. Since the index registers are limited to eight bits this method will not work for arrays which are more than 256 words long. This limitation is one of the principal reasons for the awkwardness of array addressing on the 6502. A use of this simple method of

array addressing is shown in example 10-1.

EXAMPLE 10-1

A 50 word block begins at location RECORD. This block contains data which are to be written onto an external medium. It is the common practice in such writing to include with the record being written some kind of error detection code. The simplest of such codes is known as a **checksum**. The checksum of a block of data is simply the raw sum of all of its words, ignoring carry and overflow. This checksum is included with the record being written. When the external record is read back into the computer, the program which reads it, known as a **loader**, computes an independent checksum and, when the entire record has been read, compares its computed checksum to the one included with the record read. If the two are not identical an error has occurred and the loader program notifies the operator that the loaded data are not to be relied upon. Write a program to compute the simple checksum of the 50 words beginning at RECORD, storing the result at CKSM.

```
label   inst   operand
        LDY    #50          number of words in block to Y
        CLA                 clear A to receive sum
CHKSUM  ADD    RECORD-1,Y   the block runs from RECORD
                            to RECORD+49. The effective
                            address of this instruction
                            is therefore RECORD-1+50 or
                            RECORD+49, the highest
                            address of the block.
        DEY                 decrement index. The next
                            time through the add loop the
```

```
                        effective address of the ADD
                        will be RECORD+48.
BNZ     CHKSUM          the last time through the
                        loop Y will contain a 1, so
                        that the effective address
                        of the ADD is RECORD-1+1 or
                        just plain RECORD. When Y
                        becomes zero the loop is
                        finished.
STA     CKSM            save computed checksum
...                     program continues
```

The coincidence of the two computed checksums in example 10-1 does not insure that no error has taken place, it only makes it less probable. This kind of error check is the minimum which must be performed. No data should ever be transmitted outside the computer and reread without at least a checksum error check.

It is often necessary to handle two or more arrays at once. The case of two short arrays is fairly simple on the 6502, since two index registers are available. This case is shown in example 10-2.

EXAMPLE 10-2

A bidirectional printer is to be driven using a 6502 as the "intelligent" element. To achieve correct printing in the reverse direction the computer must, among other tasks, reverse the image of the line to be printed, i.e., the first character is swapped with the last and so forth until the entire line has been reversed. The line length is 80 characters. Such applications as this require great economy in the use of memory, since not just one printer is involved. The program will typically be used in thousands of

printers, and a waste of memory of only a few words becomes very expensive when multiplied by the number of units involved. The problem, therefore, must be solved by reversing the array "in place", i.e., without using an auxiliary array for workspace. This is accomplished by dividing the array into halves and treating it as two arrays. The array begins at location LINE.

```
label   inst   operand

        LDA    #40        the number of swaps which
                          must be made is half of the
                          array size.
        STA    SCNT       save count in memory
        LDY    #0         index to low half of array
        LDX    #79        index to high half of array
REVERS  LDA    LINE,Y     get low array member
        PHA               save on stack
        LDA    LINE,X     get high array member
        STA    LINE,Y     save in low array position
        PLA               retrieve low array member
        STA    LINE,X     save in high array position
        INY               increment low index
        DEX               and decrement high index
        DEC    SCNT       decrement swap count
        BNZ    REVERS     swap again if not zero
        ...               program continues
```

Perhaps the most common of all array processing tasks is sorting, i.e., the rearrangement of the members of an array in ascending or descending order. Since this task is so common, particularly in commercial data processing environments, a very great deal of effort has been expended in finding methods for doing it efficiently. A full treatment of this subject is well beyond the scope of a book like this. If the reader has an interest in this subject he would be

advised to consult the specialized literature, e.g., Computer Sorting by Flores (Prentice-Hall, 1969).

The simplest of all sorting methods is known as a **bubble sort**. Suppose an array is to be sorted so that the numerically smallest member is to occupy the lowest memory address. The bubble sort does this by comparing each member of the array to the one in the next higher memory address, i.e., member N is compared to member N+1. If member N is greater than member N+1 the two are swapped. This process is continued until an entire comparison pass over the array can be made with no swaps. On each pass through the array a larger number moves one space closer to the top. The number of necessary passes is obviously indefinite, only one pass being necessary if the array is already in order, the number of passes required increasing as the disorder of the array increases, the worst case being that in which the array is in exactly the opposite of the required order.

In an array which contains N elements N-1 comparisons are necessary on the first pass. Further, since the largest element is "floated" to the top on the first pass it need not be recompared, and the second pass need only make N-2 comparisons beginning at the bottom. On this second pass the second largest number is "floated" to the second highest position and need not be recompared on subsequent passes. The number of required comparisons on the third pass, therefore, becomes N-3. This continues for as many passes as are necessary, the number of required comparisons for each pass becoming one less. This process continues until the number of required comparisons becomes zero, or a pass is made which requires no swapping of array elements.

The case of two equal array elements requires special attention. If two array elements are identical they must not be swapped. This would result in the number of swaps per pass being nonzero under all conditions. In this situation the sort can never end since the number of swaps cannot be zero.

Sorting an array with multiple length members is a good deal messier mechanically but the principle is the same. The comparison must be carried out in multiple precision. If the multiple precision numbers are signed remember that all but the highest orders are compared as simple magnitudes. A simple single precision bubble sort is shown in example 10-3.

EXAMPLE 10-3

An array begins at location VCTR. The number of array elements is in location VSIZE. Sort the array into ascending order using the bubble sort method.

```
label   inst    operand

        LDX     VSIZE     get array size
        BZ      EOSORT    skip out if array empty
SRTA    DEX               number of compares is one
                          less than array size
        BZ      EOSORT    finished if only one element
        STX     PCNT      number of compares/pass
        STX     DCNT      number of compares this pass
        LDX     #0        index to point to element N
        LDY     #1        and to element N+1
        STX     NSWAPS    clear swap counter for pass
SRTC    LDA     VCTR,X    get element N
        CMP     VCTR,Y    compare to element N+1
        BEQ     SRTD      don't swap if equal
        BLS     SRTD      or less
        PHA               otherwise save element N
        LDA     VCTR,Y    get element N+1
        STA     VCTR,X    place in element N
        PLA               retrieve old element N
        STA     VCTR,Y    place in element N+1
        INC     NSWAPS    increment swap counter
SRTD    INX               increment index
```

```
        INY             ...
        DEC   DCNT      finished with this pass?
        BNZ   SRTC      not if still nonzero
        LDX   PCNT      compares/pass to X
        LDA   NSWAPS    any swaps this pass?
        BZ    EOSORT    not if NSWAPS = zero
        BNZ   SRTA      otherwise set up next pass
EOSORT  ...             program continues
```

The bubble sort method of example 10-3 will work for an array of any length if modification is made to eliminate the use of the short eight bit index registers. The reader should be aware, however, that the execution time for the sort increases very sharply with the length of the list.

Table lookups are divided into two broad types, the first being that in which some incoming argument can be used directly as an index to locate the required item in the table. This technique is often used even when the the table item could be computed because of the great speed of the technique. A simple example of a fast table lookup is shown in example 10-4.

EXAMPLE 10-4

A subroutine is required which will return the square of the unsigned integer in A on entry. This integer will normally be in the range 0 - 13 inclusive. If the incoming integer is in this range return its square with the carry flag off. If the integer is out of the 0 - 13 range return with A unchanged and the carry flag on.

```
label   inst    operand
SQUARE  CMP     #14       check argument for range
        BGE     SQRTN     abort if illegal argument
```

```
        TAY             move argument to index
        LDA   SQTBL,Y   get square from table
SQRTN   RTS             return to calling program
SQTBL   DATA  0,1,4,9,16,25,36,49     table of
        DATA  64,81,100,121,144,169   squares
```

This program requires a bit of explanation. Even though the program specification above required that the carry bit be set off if the argument was legal and on if it was illegal the program contains no reference whatever to the carry bit, yet it does work. Advantage has been taken here of one of those time, labor and space saving coincidences which sometimes occurs. The BGE which follows the CMP assembles as BCS, i.e., branch if carry is set. Thus there is no need for an SEC instruction since the compare set carry. The RTS is thus executed with the carry bit on, as specified. If the BGE was not taken the carry bit was off. Since neither TAY nor LDA changes the carry bit it is still set properly when control reaches the RTS. The result is thus proper in both cases.

If you use a trick like this in a program, be sure to note the fact in the documentation. The poor devil who has to read the program a year from now after you've gone will appreciate it.

Example 10-3 showed a simple table lookup in which both argument and function were single length. Very often this is not the case, however. If, for example, the required range were not 0 - 13 but 0 - 255, the table entries would need to be of double length to accomodate the squares of the single length arguments. This requires that the indexing argument be doubled to form the displacement which is then added in double precision to the address of the origin of the table.

This case is shown in example 10-5.

EXAMPLE 10-5

A subroutine is required which will return the square of the magnitude in A. The number in A may be any value from 0 to 255 inclusive. The double length square of the argument is to be returned with the high half in X and the low half in A.

```
label   inst   operand

BIGSQR  LDY    #0          clear Y to receive bit
                           shifted out of A
        ASLA               multiply A by 2
        BCC    BSQ2        skip if nothing shifted out
        INY                high bit of A to Y
BSQ2    ADD    #<SQRTBL    form low table address
        STA    INDPTR      save in scratchpad pointer
        TYA                move high bit to A
        ADC    #>SQRTBL    form high table address
        STA    INDPTR+1    save high table pointer
        LDY    #0          clear Y for indirect
                           postindexed table access
        LDA    *INDPTR,Y   get low half of square
        PHA                save it on the stack
        INY                Y = Y+1 to access high half
        LDA    *INDPTR,Y   get high half of square
        TAX                move high half to X
        PLA                retrieve low half from stack
        RTS                and return
```

The double length table is generated by means of the DBL pseudo-op, i.e.:

```
SQRTBL DBL   0,1,4,9,16,25,36,49,64,81,100
       DBL   121,144,169,196,225,256,289
       DBL   324,361,400,441,484,529,576
```

and so forth up to:

```
DBL   65025
```

When selecting a problem solution like this, keep in mind that a very large table is required, in this case 512 words long. If space is not at a premium, all well and good, but if there are constraints on the use of memory take a long look to see whether you really need the speed that this method offers. Very often a trade of speed for space can be made. The need for speed is quite often the least critically analyzed factor in program design.

Example 10-5 also illustrates the use of the "half symbol" handlers, < and > to perform the double precision addition necessary to form the correct address in SQRTBL.

The second type of table lookup arises when it is not possible to use the argument to generate an index with which to compute the table address of the required function. In this case two tables are required, one for the argument and one for the function. The entries in the two tables may be of equal or unequal length. In either case the argument table is searched for the required value. When the argument value is found the address of the beginning of the argument table is subtracted from the table address at which the argument was found, yielding a **displacement**. This displacement is then applied to the beginning address of the function table in such a way as to generate the function table address of the required entry. The function table entry is then picked out of the table. A simple case of such a table lookup is shown in example 10-6.

EXAMPLE 10-6

An automatic telephone dialing system requires that a subroutine accept a three character identifier and from this identifier look up the seven digit phone number corresponding to it. If the identifier is found the seven digit phone number is to be copied into locations NUMBER through NUMBER+6. If the identifier is not in the table of legal identifiers the subroutine is to call a subroutine which will display an error message. This display subroutine expects to find the memory address of the ASCII string to be displayed in A and X, low half in A. The message must end with an ASCII carriage return (8D). The identifier is in locations CHAR1, CHAR2, and CHAR3.

```
label   inst   operand

PHONBK  LDY    #0          clear Y for index
        LDX    #0          and X for counter
MABELL  LDA    NAMTBL,Y    get 1st name character
        CMP    CHAR1       compare to 1st ID character
        BNE    INY1        failed if not equal
        INY                1st character OK, try next
        LDA    NAMTBL,Y    get 2nd name character
        CMP    CHAR2       compare to 2nd ID character
        BNE    INY2        failed if not equal
        INY                1st and 2nd equal, try third
        LDA    NAMTBL,Y    get 3rd name character
        CMP    CHAR3       compare to 3rd ID character
        BNE    INY3        if control gets past this BNE
                           the 3 character identifier
                           exactly matched a table
                           entry. The number of this
                           entry, beginning with 0, is
                           in X.
        TXA                entry number to A. Since the
```

```
                                entries in the phone number
                                table are seven words long
                                the displacement is computed
                                by multiplying the entry
                                number by seven. Methods for
                                multiplication by constants
                                were developed in chapter 8.
        STA     TEMP            first save the number
        ASLA                    then multiply by 8 by
        ASLA                    shifting left three times
        ASLA                    ...
        SUB     TEMP            then subtract the original
                                number, i.e., 7N = 8N - N
        TAY                     move back to Y for use as
        LDX     #7              index, move count to X
XMT     LDA     NUMTBL,Y        transmit digit
        STA     NUMBER,Y        to NUMBER area
        INY                     bump index
        DEY                     and count
        BNZ     XMT             and do it again if not zero
        RTS                     finished, return
INY1    INY                     increment index three times
INY2    INY                     or twice
INY3    INY                     or once
        INX                     increment count
        CPX     #10             end of table yet?
        BNE     MABELL          if not do it again
        LDA     #<MSG           get low message address
        LDX     #>MSG           get high message address
        JSR     DISP            send to message display
        RTS                     subroutine and return
```

The tables and the message look like this:

```
NAMTBL  ASC     'FBI'
        ASC     'CIA'
        ASC     'ONR'
        ASC     'HEW'
        ASC     'NSC'
        ASC     'GPU'
```

```
        ASC   'HUD'
        ASC   'AFL'
        ASC   'CIO'
        ASC   'FDR'
NUMTBL  ASC   '1375960'
        ASC   '5548765'
        ASC   '3387556'
        ASC   '9671714'
        ASC   '4462521'
        ASC   '7291396'
        ASC   '3923250'
        ASC   '4924669'
        ASC   '4809240'
        ASC   '8694008'
MSG     ASC   'IF YOU CAN'
        DATA  $A7
        ASC   'T SPELL DON'
        DATA  $A7
        ASC   'T BLAME MA BELL'
        DATA  $8D                    carriage return
```

The message IF YOU CAN'T SPELL DON'T BLAME MA BELL was chosen to illustrate how apostrophes are inserted into an ASCII string. Since the delimiter for an ASCII string is an apostrophe the apostrophe may not be used as a character within a string. If an apostrophe is required in a character string the string must be terminated and the numerical code for apostrophe, $A7 or its equivalent created with a DATA pseudo-op. The string is then resumed with another ASC as shown above.

The examples thus far have involved arrays short enough that the index registers could be used to address any member. Arrays are often, however, much longer, and techniques for handling arrays longer than 256 words must be developed. There are three basic methods for doing this. The first involves creating a pointer in the scratchpad and incrementing Y to reach successive array members via indirect postindexed addressing. When Y reaches zero the upper half of the

scratchpad pointer is incremented, bringing the next 256 words within addressing range. The program then proceeds to address higher memory by incrementing Y alone until Y reaches zero again, at which time the high half of the scratchpad pointer is again incremented. This process continues until the entire array has been covered. This method has the advantage of speed since a memory word is only incremented once for each 256 accesses. Memory incrementation is much slower than register incrementation. It is in general much faster to perform any operation in a register than in memory. An array search which uses this type of incrementation is shown in example 10-7.

EXAMPLE 10-7

An array named LETERZ is located in upper memory. It is to be searched and the number of appearances of the letter E is to be counted, the count being kept in locations ECOUNT and ECOUNT+1. LETERZ is 3987 words long.

```
label   inst   operand

        CDB    ECOUNT,0        clear count cells
        CDB    SRCNT,-3987     negative of array size
                               for loop count
        CDB    PTR,LETERZ      create array pointer
SEARCH  LDA    *PTR,Y      get array character
        CMP    #'E'        check for E
        BNE    S2          skip if not E
        IDB    ECOUNT      increment count of Es
S2      INY                increment index
        BNZ    S3          skip if Y not zero
        INC    PTR+1       increment high pointer
S3      IDB    SRCNT       increment count
        BNZ    SEARCH      repeat if not zero
        ...
```

The use of CDB and IDB in example 10-7 highlights the usefulness of these pseudo-ops. CDB allows four lines to be replaced by one and IDB replaces three lines with one. Had the instruction sequences which CDB and IDB represent been written out the example would have been eleven lines longer. It must be reiterated here that these pseudo or compound instructions do nothing that the programmer could not do for himself. They only save writing.

A second method for dealing with the addressing of larger arrays is to dispense with the use of the index register entirely. Y is set to zero and the scratchpad pointer incremented after each access. This process is slower than the one shown in example 10-7 but is a little easier to understand.

EXAMPLE 10-8

The problem posed in example 10-7 is to be solved but without the use of the index register. The scratchpad pointer is to be incremented directly.

```
label   inst   operand
        CDB    ECOUNT,0        clear count cells
        CDB    SRCNT,-3987     set loop count
        CDB    PTR,LETERZ      create scratchpad
                               pointer
        LDY    #0          clear index
SEARCH  LDA    *PTR,Y      get character
        CMP    #'E'        check for E
        BNE    S2          skip if not E
        IDB    ECOUNT      increment E count
S2      IDB    PTR         increment double address
                           pointer in scratchpad
        IDB    SRCNT       increment loop count
        BNZ    SEARCH      repeat if not zero
```

```
...                 program continues
```

Remember that this only looks shorter because the IDB performed on the address in scratchpad is written as a single operation. While it is simpler to understand than the operation in example 10-7, this method is slower because memory is being incremented after every access instead of every 256th access.

A third method for addressing large arrays involves the use of a technique called **self modification**. This is normally the least desirable method since it cannot be employed in programs which are to be located in ROM. It is employed only as a method of last resort to satisfy special conditions, e.g., a constraint that the program may not use scratchpad to hold a pointer. In this case the self modification involves the direct incrementation of the address of the memory reference instruction. Memory reference instructions which do not address scratchpad are three words long, the first word being the binary instruction, complete with addressing mode, and the last two words the direct address expressed as a double precision binary number. An LDA addressed to location $C37A, for example, consists of the following three words:

```
AD  7A  C3
```

the AD being the LDA instruction for a non scratchpad load. The 7A is the low half of the address and the C3 is the high half in the normal double precision order with the low half occupying the lower memory address. Performing a double increment on the address occupied by the 7A, as with an IDB pseudo-op, yields the following result:

```
AD  7B  C3
```

i.e., the address of the LDA has been changed to $C37B. This technique will yield correct results, of course, but should be used only when there is no other way out. It will not be illustrated further here. It is used in the debug program at the back of the book for the reason that the debug uses no scratchpad memory whatever to avoid interaction with the program being debugged.

CHAPTER 11

PROGRAMMED INPUT/OUTPUT

If a computer program is to perform any meaningful task it must at some point present its results to the outside world. This presentation may take the form of a CRT screen display or a printed page for human inspection, or it may take the form of a controlling signal for some external machine being operated by the computer. Except for theoretical computations which are to be executed only once with fixed data a program also requires some starting data. The problem in both cases is communication between the computer and the outside world.

Most computers transfer data to and from the outside world by means of special instructions known as **input/output** instructions, a set of special instructions which transfer data between machine and operating environment, usually via the machine's A register. Each of the devices with which the computer is to communicate has a unique number known as a **device address** associated with it, and data are routed to and from the proper device by means of this number.

The 6502 is one of a small number of computers which uses a different method for data transfer. It has no instructions specifically devoted to input/output nor is any device address scheme used. Communication with external devices, or **peripherals**, is accomplished through computer memory. Each peripheral device has one or more memory addresses associated with it. Data are transmitted to a peripheral device from the 6502 by storing into the appropriate address. Transmission of data from a peripheral device to the 6502 is accomplished by loading a register from the appropriate memory location. We will take up the problem of input first.

In the case of the keyboard of the Apple II getting the input is no problem. The ASCII code associated with the key appears in memory location

$C000, i.e., when the "A" key is struck on the Apple keyboard the contents of location $C000 become $C1, the ASCII code for the letter A. The contents of $C000 may then be loaded into A with an LDA and examined. The problem is how the program knows that a key has been struck at all.

In the case of the Apple keyboard only ASCII codes will be transmitted. A table of these codes is given at the end of this chapter. Each of these eight bit codes has its most significant bit set to one, meaning that this bit contains no information about the code at all. In fact these codes would function as well if all of the high bits were set to zero. Since the high bit is irrelevant the Apple people have decided to use it for something else, namely as a flag to tell the program when a key has been struck. If no key has been struck the high bit of location $C000 is a zero. When a key is struck the high bit of $C000 is set to one and the low seven bits of this memory word are set to the ASCII code for the key. Before the keystroke location $C000 looks like this:

0xxxxxx

the x's meaning irrelevant contents. After the "A" key is struck the contents of $C000 look like this:

11000001

or $C1, the ASCII code for the letter A. The high bit is thus used as a kind of flag indicating keyboard activity. The flag can be tested by the program to determine when data are to be transmitted from the keyboard. There are several ways to do this. Since the flag is the high bit of $C000 this word can be loaded into a register and tested for negative. If a key has been struck, thus turning on the high bit, the register will test negative. If the register is positive no key has been struck. Since it is commonly the situation that the program must wait for input, the

usual practice is to simply keep loading the contents of $C000 into a register and testing until a key is struck. After the code is read from $C000 it must be "reprimed" for the next keystroke by setting the high bit back to the zero or "not ready" condition. One might think that this would be done by clearing $C000 but this is not the case. In the Apple the ready flag is cleared by storing into location $C010. The operation is shown in example 11-1.

EXAMPLE 11-1

Using the data given above, write a subroutine which will wait until a key is struck on the Apple keyboard, then return with the ASCII code for the key in the A register. The subroutine should have entry point name RKB.

```
label   inst    operand

RKB     LDA     $C000   get contents of $C000
        BPL     RKB     test high bit. If the high
                        bit is zero no key has been
                        struck. In this case branch
                        back and try again until a
                        a keystroke makes $C000
                        negative.
        STA     $C010   clear this keystroke
        RTS             and return.
```

The scheme shown in example 11-1 will work, of course, but the the use of absolute memory addresses is a nuisance and in many cases a source of error. Consider what might happen if a keyboard read sequence like the above were written at a number of different points in the program. References to the data location

$C000 and the "clearing" location $C010 would be scattered over the program. If the locations were ever to change for some reason, e.g., the installation of an auxiliary keyboard, every one of the references would have to be changed. This would not be difficult for one short sequence like example 11-1, but if there were 10 or 20, it would be very easy to miss one and chaos would ensue.

This is a case where good programming practice would dictate that the programmer force the programming system to do the housekeeping work for him. To do this we have the EQU pseudo-op. EQU allows the symbol in its label field to be given the value in its operand field. To give the symbol HANS the value $B300 and FRITZ the value $B302, we write:

```
HANS   EQU  $B300
FRITZ  EQU  $B302
```

EQU may have symbols or sums of symbols and numbers in its operand field but no symbol may appear in the operand of an EQU which has not appeared in a label field at a point previous in the program, i.e.:

```
HANS   EQU  $B300
FRITZ  EQU  HANS+2
```

is legal, but:

```
FRITZ  EQU  HANS+2
HANS   EQU  $B300
```

is not and will result in an error message and the assembly will be aborted. EQUs which define scratchpad locations must appear within the range of an SPD pseudo-op, and those which define general memory locations must be outside the range of an SPD. Using EQUs to define the keyboard data location and "clearing" location (known as the keyboard strobe) the Apple keyboard input subroutine looks like this:

EXAMPLE 11-2

The problem of example 11-1 is to be solved without reference to absolute memory addresses. The keyboard data location KB and the keyboard strobe location KBS are to be defined using EQUs.

```
label   inst   operand

RKB     LDA    KB        get flag and data
        BPL    RKB       test ready
        STA    KBS       clear strobe
        RTS              return to calling program
KB      EQU    $C000     define KB
KBS     EQU    KB+$10    define KBS
```

The Apple keyboard arrangement in which the ready flag and the data share the same word is an unusual one. Most commonly there are two words associated with each peripheral device, one being the data word and the other containing the ready flag as well as other indicators of the status of the peripheral. This second word is known at the **status word**. We will see this usual arrangement when we examine the parallel printer.

Output of data to be displayed on a screen or printed on a page takes two general forms. In the simplest form the successive characters are simply put out in a string with the peripheral device automatically advancing the position for the appearance of the next character. This is the case with such devices as teletypewriters. Once a character has been sent out it cannot be recalled or changed since the print carriage has already physically moved in anticipation of the next character. In simple devices

like teletypewriters the characters are printed as they are received, i.e., no facility exists for assembling an entire line before print is initiated.

Printers most often contain a buffer in which an entire line is assembled, with the actual printing being initiated by the appearance of a special character, usually carriage return ($8D). The Apple parallel print interface presents a fairly typical situation in this respect. We will study a specific case here, that of the parallel printer located in slot 3 of the Apple chassis. The data location in this arrangement is $C0B0, i.e., data sent to the printer are stored into $C0B0. In most situations the ready flag location is in the next higher or lower memory address, but the Apple people have not done this for some reason which is not apparent to this writer. The ready flag is located at $C3C1. Further, the interface is so arranged that the memory reference instruction which stores data into $C0B0 or loads the flag from $C3B1 must be indexed by X. This seems an odd bit of design, but we will pass on without further comment.

Data to be sent to the printer are stored into location $C0B0 by an instruction of the form:

```
STA  PRTR,X
```

X having been previously set to zero and the symbol PRTR having been defined elsewhere by:

```
PRTR   EQU  $C0B0
```

The printer status word is defined:

```
PRFLAG EQU  $C3C1
```

As with the keyboard, the ready flag for the printer is the high bit of $C3C1, so it can be tested in A by testing the sign bit. A string of characters is sent to the printer ending with carriage return. A parallel printer driver subroutine is shown in example 11-3.

EXAMPLE 11-3

A subroutine is required which will print a line on the Apple parallel printer. The address of the first character of the string to be printed will be in A and X on entry, low half in A. The string is to terminate when a carriage return is encountered. When the carriage return is sent out the subroutine is to send out a line feed ($8A) and return control to the calling program. A previously defined scratchpad pointer named IOPTR is to be used to hold the string address.

```
label   inst   operand

PLPRT   STA    IOPTR      save string pointer
        STX    IOPTR+1    ...
        LDX    #0         clear X to address
                          printer location
        LDY    #0         clear index for access
PLPRT2  LDA    *IOPTR,Y   get string character
        STA    PRTR,X     send to printer
        PHA               save character on stack
PLPRT3  LDA    PRFLAG,X   get status word
        BMI    PLPRT3     branch back if not ready
        PLA               retrieve character
        INY               increment index
        CMP    #$8D       was it a carriage return?
        BNE    PLPRT2     no, get next character
        LDA    #$8A       yes, generate line feed
        STA    PRTR,X     send out line feed
PLPRT4  LDA    PRFLAG,X   get print ready flag
        BMI    PLPRT4     wait for line feed to clear
        RTS               and return
```

In brief resume, a character is sent to the printer and the ready flag is tested until the printer is ready to accept another character. This effectively "hangs" the computer while waiting for the output operation to be completed, i.e., the computer can do nothing else but wait for the ready flag. This is the reverse of the order used in reading the keyboard, i.e., the ready flag is tested until it indicates that a key has been struck and the data are then transmitted.

One might wonder in passing here what consequences might occur in some program situations where "hanging" the computer like this might not be tolerable. Suppose that your 6502 were monitoring the core temperature of a reactor, reading this temperature once per second or so and attending to certain auxiliary tasks like keeping a record of the name of the current reactor operator. Now imagine that a shift change at the reactor is taking place, and the new operator is typing in his name at the keyboard. Suppose further that the new operator, like this writer, were a two finger typist, and that he had a name like Lambros Alexios Papatriantaphilopolous. About the end of the Lambros part the reactor core begins to overheat. The computer, however, notices nothing since it is completely tied up waiting for the keyboard ready flag. Poor Lambros is pecking away at the keyboard, doing his noble best, but the reactor core is beginning to melt. About the time Lambros gets to the first letter of Papatriantaphilopolous the entire scene, computer, reactor, building, Harrisburg, and, of course, poor Lambros depart to meet the Great Neutron Moderator in the sky. Get the idea? There are times when you just can't wait for the ready flag.

Since the reader already knows that computers are used for such tasks, he may well be beginning to get an uncomfortable feeling about being only five miles downwind of the Bigblast Nuclear Power Station. Be consoled, therefore, by the thought that when you gotta glow, you gotta glow.

After the Three Mile Island incident such a hypothetical sketch might be viewed by some readers as being in poor taste, but it was necessary to bring this limitation of programmed input/output methods into sharp focus. Obviously, there must be some way around this difficulty and just as obviously, the reactor people have thought of the problem and solved it.

What is required here is some means of freeing the computer from the chore of waiting for the ready flag. The entire problem could be solved if there were some hardware means for monitoring it automatically which did not require the attention of the computer. If some kind of automatic ready flag monitor could be devised which was capable of watching the keyboard ready flag and notifying the computer in some way that a key had been struck, then the computer could go about its temperature monitoring task continuously, taking only the few microseconds necessary to actually read the data from the keyboard when a key was struck. If this were the case we could look forward to actually being notified in time to see the message:

THAT'S ALL FOLKS

appear on the CRT before we shuffle off Mr. Shakespeare's mortal coil.

Such a facility exists in modern computers. It is known as the **interrupt system**. Using the interrupt system of the computer any number of data sources and/or destinations can be automatically monitored while allowing the computer to perform useful work without waiting for ready flags. The interrupt system will be discussed in detail in chapter 18. While the 6502 has interrupt capability, the Apple II allows the programmer no access to it without hardware modification.

Yet another form of output involves the direct projection of the contents of a block of memory onto a CRT. To make something appear on the screen it is simply placed into the appropriate area of memory.

Such an arrangement is known as **memory mapped I/O** and the Apple II screen works this way. Lest this begin to sound too simple, the Apple designers, for reasons known only to them, have not made this task so easy that the fun and challenge are all missing. The Apple has two areas of memory from which text can be projected onto the CRT. One of these begins at $400 and the other at $800. The mechanics are identical for both screens. We'll choose the higher one. Which screen is displayed at any one time is determined by a sequence of instructions detailed in the Apple manual.

The text contents of the top line of the screen are determined by the contents of memory locations $800 through $827. The screen width is 40 characters. The leftmost displayed character is the one in memory location $800. Suppose we filled the $800 - $827 block with hexadecimal $AA, the ASCII code for an asterisk. The top line of the screen would look like this:

****

Thinking to begin the second line we put $AA into location $828, as a reasonable person might suppose that the second line would begin where the first line left off. Surprise! The asterisk appears in the first position of line 9! Something is not obvious about this process.

After some experimenting we compile the following table of line numbers and the hexadecimal and binary addresses of their leftmost characters.

Line #	Hex Address	Binary Address
0	800	1000 0000 0000
1	880	1000 1000 0000
2	900	1001 0000 0000
3	980	1001 1000 0000
4	A00	1010 0000 0000
5	A80	1010 1000 0000
6	B00	1011 0000 0000
7	B80	1011 1000 0000
8	828	1000 0010 1000
9	8A8	1000 1010 1000
10	928	1001 0010 1000
11	9A8	1001 1010 1000
12	A28	1010 0010 1000
13	AA8	1010 1010 1000
14	B28	1011 0010 1000
15	BA8	1011 1010 1000
16	850	1000 0101 0000
17	8D0	1000 1101 0000
18	950	1001 0101 0000
19	9D0	1001 1101 0000
20	A50	1010 0101 0000
21	AD0	1010 1101 0000
22	B50	1011 0101 0000
23	BD0	1011 1101 0000

the lines being numbered 0 through 23. At first glance this might look hopeless but there is a some regularity. In all but two cases the address of the next line can be gotten by adding $80 to the address of the one before it. In the binary column a couple of things become obvious almost at once. First, the leftmost two bits and the rightmost three bits are always the same. All of the variation occurs in the bits in between. If we rewrite the binary address table spaced a little differently the addressing scheme becomes a little clearer.

Line #	Binary Address				
0	10	000	00	00	000
1	10	001	00	00	000
2	10	010	00	00	000
3	10	011	00	00	000
4	10	100	00	00	000
5	10	101	00	00	000
6	10	110	00	00	000
7	10	111	00	00	000
8	10	000	01	01	000
9	10	001	01	01	000
10	10	010	01	01	000
11	10	011	01	01	000
12	10	100	01	01	000
13	10	101	01	01	000
14	10	110	01	01	000
15	10	111	01	01	000
16	10	000	10	10	000
17	10	001	10	10	000
18	10	010	10	10	000
19	10	011	10	10	000
20	10	100	10	10	000
21	10	101	10	10	000
22	10	110	10	10	000
23	10	111	10	10	000

Now the order begins to emerge. The second and third columns from the right, each two bits wide contain the identical information in every case. Further, in the topmost group these bits are 00, in the second group they are 01 and in the third group they are 10, i.e., 0, 1, and 2. The Apple screen appears to be split into three subscreens with these bit groups identifying the subscreen.

Looking now at the second column from the left, in each group this three bit column contains 000, 001,

010, 011, 100, 101, 110, and 111, the integers from 0 through 7. This three bit group apparently identifies the line number within the subscreen. With this information it will be possible to concoct a screen addressing scheme that will allow the Apple screen to be addressed in a somewhat orderly way.

The first task is to create a means by which we can generate the address of the first character of a line from the line number. The binary line numbers run from 00000 to 10111. If we split the binary line number like this into two groups of bits the course to be followed becomes obvious. Write the line number like this:

10 111

and it is immediately apparent that the left two bits are the subscreen number and the right three are the line number within the subscreen. Using this information we can write a subroutine to generate the address of the first character of each line from its line number. Given this address the address of the following characters of the line are found by simply adding the character number 0 through $27 to the address of the leftmost character. The subroutine is shown in example 11-4.

EXAMPLE 11-4

A subroutine is required to compute the address of the leftmost character of the line whose number is in A on entry. The number will be in the 0 - 23 range. The address is to be returned in A and X. low order in A.

```
label   inst   operand

LNADR   TAX            save line number in X
```

```
        AND   #7        isolate line number
                        within subscreen
        ORA   #$10      this creates the leftmost
                        five bits of the address
                        but displaced left one bit
        STA   T+1       save high five bits
        TXA             retrieve line number
        AND   #$18      isolate subscreen number
        STA   T         save it
        ASLA            shift left to make copy
        ASLA            ...
        ORA   T         form low seven bits
        ASLA            displace one bit left
        STA   T         at this point the full
                        address of the line has been
                        computed and is in T and T+1
                        but displaced left one bit.
                        This is remedied by a
                        double shift.
        LSR   T+1       shift high half, low bit to
        ROR   T         carry, then shift low half
                        and roll carry in.
        LDA   T         low address to A
        LDX   T+1       high address to X
        RTS             return with address in A & X
T       $$$             temporary doubleword
```

The address of any character on the screen can now be generated by adding the horizontal character position, beginning at zero, to the address generated by the LNADR subroutine of example 11-4.

In most circumstances not involving graphics the screen is most conveniently treated as if it were a piece of rolled paper which scrolls upward automatically each time a new line is to be written. The new line is then written in the vacated space at the bottom of the screen. The problem is how to make the screen scroll up one line. Obviously, the second

line from the top must be copied into the top line, the third line into the second and so forth, with the bottom line being set to blank to receive the new text. A method for doing this is shown in example 11-5.

EXAMPLE 11-5

A subroutine is to be created which will scroll the Apple screen up one line and blank the bottom line. Use the method of direct modification of addresses detailed in chapter 10.

```
label   inst   operand

SCRADV  LDA    #0        set line number to zero
        PHA              save on stack
        JSR    LNADR     generate address of line
        STA    FRADR     save in "from" address
        STX    FRADR+1   ...
SCR2    LDA    FRADR     move "from" address
        STA    TOADR     to "to" address
        LDA    FRADR+1   ...
        STA    TOADR+1   ...
        PLA              retrieve line number
        INA              increment it
        CMP    #24       last line moved?
        BEQ    SCRTN     yes if equal
        PHA              save line number again
        JSR    LNADR     compute line address
        STA    FRADR     save new "from" address
        STX    FRADR+1   ...
        LDY    #23       load index for move
SCR4    LDA    SCR4,Y    get character and move
SCR5    STA    SCR5,Y    to new line. The actual
                         addresses of these last two
                         instructions are not SCR4
                         and SCR5. The addresses
                         have been overwritten by the
```

```
                              results returned by the
                              LNADR subroutine.
FRADR   EQU     SCR4+1        this EQU allows direct
                              modification of the address
                              of SCR4 above.
TOADR   EQU     SCR5+1        and this one does the same
                              for SCR5
        DEY                   decrement index
        BPL     SCR4          and repeat if still plus
        BMI     SCR2          otherwise see if any more
                              lines are to be moved
SCRTN   LDY     #23           blank out bottom line
        LDA     #' '          blank to A
SCR6    STA     LAST,Y        blank out character
        DEY                   decrement index
        BPL     SCR6          repeat if still plus
        RTS                   and return
LAST    EQU     $BD0          address of first character
                              of last screen line
```

It is to be reiterated here that this addressing technique is the one of last resort. It will not run in ROM and should only be used if no space is available for a scratchpad pointer to allow indirect addressing. Understanding how it works does, however, make an excellent test of your understanding of the programming concepts involved.

In summary, programmed input/output is the transfer of data to or from the external world under control of a ready flag, this ready flag becoming "true" when the peripheral device is prepared to transmit data. Note here that in the case of input the ready flag only becomes true when data are ready to be transmitted, i.e., the normal state of the flag when the device is inactive is false. With output, however, the flag is true when the device is inactive, meaning that it is ready to accept data whether or not the computer is ready to transmit. This distinction has

important ramifications for input/output done under interrupt control, as we shall see.

Finally, there is the matter of defining the ASCII codes. An ASCII character is an eight bit unit which represents a letter of the alphabet, a decimal digit, or other special character. The assembler generates ASCII codes with the high bit set to one.

The two highest bits of ASCII characters have a special significance in the Apple computer. If they are set as the assembler generates them, then the ASCII characters they represent will appear normal on the Apple screen, i.e., white on black background. If these two bits are set to zeros the character will appear inverse, i.e., black on a white rectangular background. If the two high bits are set to 01 the character will appear as inverse, but flashing. The ASCII characters and their hexadecimal representations are tabulated on the next page.

Symbol	Hex	Symbol	Hex
@	C0	blank	A0
A	C1	!	A1
B	C2	"	A2
C	C3	#	A3
D	C4	$	A4
E	C5	%	A5
F	C6	&	A8
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

carriage return	8D
line feed	8A
bell	87
rubout	FF
null	00

CHAPTER 12

6502 DECIMAL ARITHMETIC

Brief mention has been made earlier of BCD numbers, i.e., four bit quantities whose value is less than ten which represent decimal digits. BCD means **binary coded decimal**. Since BCD digits are only four bits long, two of them can be packed into a single 6502 word. The 6502 has some limited ability to operate on such decimal numbers and it is the purpose of this chapter to describe this ability in detail and the programming methods necessary to exploit it.

Most microprocessors have some limited decimal arithmetic capability, confined to addition or to addition and subtraction at the most. BCD numbers are operated upon by the same instructions as binary numbers. In most microprocessors a special "adjust" instruction is provided to correct the binary result to represent the BCD sum or difference.

The 6502 is unique among current microprocessors in possessing the ability to perform BCD addition and subtraction directly, with no special action required of the programmer to correct the sum or difference. It does this by entering a **decimal mode** in which all addition and subtraction is treated as decimal. The decimal mode is entered on the 6502 by executing the SED (SEt Decimal) instruction. The decimal mode is cleared, i.e., normal binary mode is re-entered by executing the CLD (CLear Decimal) instruction.

Decimal arithmetic is performed on unsigned BCD numbers, two digits at a time. The carry bit functions in a manner analogous to the way in which it functions for normal binary arithmetic. If a decimal sum greater than 99 is formed the carry bit is set to one. This carry may thus be propagated to higher orders allowing multiple precision decimal addition to be done. The ADD and ADC instructions may be used just as they are in binary.

BCD subtraction follows the same rules as binary

subtraction. If a situation arises in which the subtrahend is greater than the minuend the carry bit is set just as it is for binary subtraction. Note again that zero in carry means a borrow and one means no borrow. This convention is different from that of every other current microprocessor. The addition of two BCD numbers is shown in example 12-1.

EXAMPLE 12-1

Two BCD numbers are located at D1 and D2. Find their sum and save it in location DSUM. If the sum exceeds 99 transfer control to location TOOBIG after saving the sum.

```
label   inst    operand
        SED             set decimal mode
        LDA     D1      get addend
        ADD     D2      form BCD sum
        STA     DSUM    save result
        CLD             restore binary mode
        BCS     TOOBIG  check for overflow
        ...             program continues
```

The process is really very simple. The presence of a one in carry in example 12-1 simply means that the sum was greater than decimal 99, i.e., another word is required to hold it. Performance of BCD addition in double precision is equally simple. Just as in multiple precision binary addition, the low order sum is formed first, with the second and higher orders including carries out of the lower orders. This can be carried out to any required precision whatever with the general case being very similar to the N precision binary addition of example 7-4 on pages 88-89. A

double precision BCD addition is shown in example 12-2.

EXAMPLE 12-2

Two four digits BCD numbers are located at N1 and N1+1, and at N2 and N2+1. Find their BCD sum and leave it at DSUM and DSUM+1. If the sum exceeds 9999 transfer control to location TOOBIG.

```
label   inst    operand

        SED             set decimal mode
        LDA     N1      get low augend
        ADD     N2      add low addend
        STA     DSUM    save low sum
        LDA     N1+1    get high augend
        ADC     N2+1    add high addend. Note that
                        the ADC propagates the
                        carry from the low order
        STA     DSUM+1  save high sum
        CLD             restore binary mode
        BCS     TOOBIG  check for overflow
        ...             program continues
```

It is important to understand what decimal mode will not do as well as what it will do. Setting the 6502 into decimal mode does not convert the contents of the A register to decimal. This seems to be widely misunderstood. Further, the operation of decimal mode is confined to the A register. Incrementing a memory location which contains a BCD number will not, in general, produce a correct BCD result.

BCD arithmetic is not strictly limited to decimal manipulation. The sharp-eyed programmer can sometimes find a use for it elsewhere. Consider the problem of converting a four bit number 0 - 15 to an ASCII digit

which represents its value. Values 0 - 9 present no problem since the ASCII codes for 0 - 9 are $B0 - $B9. If the digit is greater than 9, however, we must come to grips with the fact that there is a gap in the ASCII sequence between 9 and A. A trick using decimal arithmetic solves this in example 12-3.

EXAMPLE 12-3

A four bit binary number 0000 through 1111 is in the low four bits of A, the upper four bits being zeros. Convert this number to its ASCII-hexadecimal equivalent.

```
label   inst    operand
HEX     SED             set decimal mode
        ADD     #0      this ADD seems to do nothing
                        since its operand is zero.
                        If the low four bits of A are
                        a number greater than 9,
                        however, it changes the
                        number to its BCD equivalent,
                        i.e., 00001100 becomes
                        00010010. If the low four
                        bits are 9 or less nothing
                        happens.
        CLD             re-enter binary mode
        CMP     #$A     this compare has the effect
                        of setting carry to 1 if A
                        is greater than 00001001,
                        i.e., if the decimal ADD
                        caused a carry into the upper
                        four bits of A. If A is 9 or
                        less carry is cleared.
        ADC     #0      this ADC simply adds the
                        carry bit to A. If the orig-
                        inal number in A was 00001010
```

```
                        A now contains 00010001.
ADD    #$B0             if A contains 00001001 or
                        less this simply generates
                        10110000 through 10111001,
                        the ASCII digits for 0 - 9.
                        If A originally contained a
                        number greater than 00001001
                        it now contains an alphabetic
                        character in the range A - F.
...                     program continues.
```

Decimal subtraction is performed in much the same way as binary subtraction. A simple decimal difference is computed in example 12-4.

EXAMPLE 12-4

Two BCD numbers are in locations S and T. Perform the BCD subtraction S - T and leave the difference in U.

```
label  inst   operand

       SED             set decimal mode
       LDA    S        get minuend
       SUB    T        subtract subtrahend
       STA    U        save BCD difference
       CLD             restore binary mode
       ...             program continues
```

It must be understood here that BCD arithmetic is performed on unsigned BCD magnitudes. This leads to an unexpected result if the subtrahend is larger than the minuend. Subtracting 2 from 1 gives a difference of

99. Since decimal arithmetic cannot form a negative result it does the next best thing and "wraps around". Readers familiar with old fashioned mechanical calculators may recognize this phenomenon. BCD arithmetic uses what is called a **tens complement** scheme for subtraction. The tens complement result of subtraction when the subtrahend is larger than the minuend is a string of nines filling the nonsignificant leading digits. This is the exact decimal analog of the ones which fill leading nonsignificant bits in a negative binary result.

Double precision binary subtraction is equally straightforward if one keeps in mind the inverted sense of carry during subtraction. The process is shown in example 12-5.

EXAMPLE 12-5

A double precision BCD minuend is located at X1 and X1+1, and a double precision BCD subtrahend is at Y1 and Y1+1. Find their difference and leave it at DIFF and DIFF+1.

```
label  inst   operand
       SED              set decimal mode
       LDA    X1        get low minuend
       SUB    Y1        subtract low subtrahend
       STA    DIFF      save low difference
       LDA    X1+1      get high minuend
       SBC    Y1+1      subtract high subtrahend
       STA    DIFF+1    save high difference
       CLD              restore binary mode
       ...              program continues
```

A common programming error when using BCD

arithmetic is forgetting to return to binary mode when the decimal computation is finished. The program goes on attempting to compute binary results and getting nonsense. This can be quite mysterious since it can cause a previously checked out and debugged section of the program to behave strangely. Remember to turn decimal mode off when you have finished.

It is possible to perform decimal multiplication and division on the 6502, but the question of whether it is worthwhile must be raised. Multiplication is performed by the same shift and add process as in binary, but the digit shifted out cannot be held in carry, since it is four bits long. Further, the test of the shifted out multiplier digit is not a simple zero-one test, but a test for zero and nonzero. If the digit is nonzero the multiplicand must be added to the partial product a number of times equal to the shifted out digit.

BCD division is similarly grim. The decimal dividend must be compared to the decimal divisor for the divide fault condition. The division proceeds as in binary with the dividend being shifted left, the high half compared to the divisor and, if subtraction is possible, the divisor is subtracted from the high dividend until zero is crossed. The number of such successful subtractions becomes the BCD quotient digit.

As to the question of whether this is worth the trouble, the answer must be a resounding no. If multiplication or division must be done it will be much faster and shorter to convert the BCD numbers to binary, perform the required operations via the methods described in chapter 8, and then convert the binary results back to decimal. This will be faster and cleaner in all cases.

While there is no doubt whatever that BCD multiplication and division can be programmed on a 6502, the execution of such a task must be an academic exercise. The result will be of marginal usefulness because of glacially slow execution times and the size of the program required to accomplish the task.

What would be required to speed the process up to the point of feasibility would be decimal shifts, i.e., shifts which moved four bits at a time between memory and the A register. The Z80 is unique among current microprocessors in having such shifts. Full four function decimal arithmetic is developed in the writer's earlier **PRACTICAL MICROCOMPUTER PROGRAMMING: THE Z80** (Northern Technology Books, 1978). Even with this most powerful of eight bit microprocessors the result is of marginal usefulness because of lengthy execution times.

CHAPTER 13

FLOATING POINT ARITHMETIC

The discussion of arithmetic operations thus far has centered around situations in which the ranges of the numbers to be handled were strictly limited and of a known precision, being confined in most cases to integers. Such arithmetic is known as **fixed point** arithmetic. Fixed point arithmetic remains the method of choice for all situations in which the the range of intermediate and final results can be known exactly.

Situations do arise, however, particularly in scientific and engineering environments, in which it is difficult or impossible to predict the ranges of intermediate and final results, these ranges being dependent upon the range of data input to the problem.

In such situations the programmer must resort to an alternate means of arithmetic, one which will keep automatic track of the magnitudes of results. This scheme of computation is known as **floating point** arithmetic. The purpose of this chapter is to acquaint the reader with the methods by which such arithmetic is done to enable him to apply floating point methods intelligently and be aware of the limitations of these methods. The examples here will be given in decimal for clarity until the basic principles have been worked out.

The representation of floating point numbers is very similar to that used in scientific notation. In scientific notation a number is represented as a magnitude with a single integer digit accompanied by an exponent, i.e., a power of ten by which the magnitude is to be multiplied to yield the true value of the number. The number 384.7 is thus represented in scientific notation as:

$$+3.847 \times 10^{+2}$$

This number has four basic parts: the algebraic sign,

the magnitude, the sign of the exponent and the exponent itself. A floating point number is similarly composed, though there is a difference in the way the exponent and its sign are represented. We will come to this difference shortly. Some floating point systems represent numbers with a single integer digit to the left of the point as in the example above, e.g., the system which Intel Corporation has proposed as an industry standard uses this format. Most, however, represent numbers using a magnitude which is pure fraction. In this system the number above would be written:

$$+.3847 \times 10^{+3}$$

This amounts only to shifting the decimal fraction right one digit and incrementing the exponent to compensate for the shift. Further, since the number base does not change, it is unnecessary to write the exponent in explicit form. We represent the exponent as a pair of digits separated from the fraction, like this:

+3847 03

the decimal point being implicitly always at the extreme left end of the magnitude.

The addition and subtraction of floating point numbers exactly parallel those operations on numbers in scientific notation. In either case numbers with different exponents cannot be added or subtracted directly. Before addition or subtraction of the magnitudes can take place the exponents must be equalized. Thus the pair of floating point numbers:

+3847 03 or 384.7
+8275 02 or 82.75

cannot be added directly, as the exponents are unequal. We have two ways to equalize the exponents here.

Either the magnitude of the larger number can be shifted left and its exponent decremented, like this:

+3.847 02

or we can shift the magnitude of the smaller number to the right, introducing a nonsignificant leading zero to its fraction, and increment its exponent, like this:

+0.0827 03

In pencil and paper operations it makes no difference which we do, as the result of the addition or subtraction will be the same. One magnitude or the other must be shifted a number of places equal to the difference of the exponents. Unlike pencil and paper operation, however, performing this operation in a computer runs into some hardware constraints. If the magnitude of the larger number were to be shifted left the floating point processor would have to leave room for a shift of the maximum possible exponent difference. To leave less room than this would risk a high significance digit or digits being lost in the shift. The choice, therefore, must be to shift the fraction of the smaller number to the right a number of places equal to the difference of the exponents. Since the length of a floating point magnitude is fixed and limited this involves the risk of losing significant digits on the right end. Indeed the sharp-eyed reader may have noticed that this is exactly what happened when +8275 02 was shifted to +0827 03 above. The magnitude being confined in this case to four digits, the 5 was lost in the shift.

Such losses are in the nature of floating point arithmetic. The obvious answer to this problem is to carry more significant digits, and this solution has been applied. Another solution is to build a "guard digit" into the system which "catches" the digit shifted out and restores it if possible when the operation is finished. This will be discussed in

somewhat greater detail later in the chapter. A case of simple addition of floating point numbers is shown in example 13-1.

EXAMPLE 13-1

Convert the numbers 445.1 and 16.74 into floating point format and perform the floating point addition.

It is necessary to shift 445.1 three places to the right to make it a pure fraction, .4451. This three place shift means that an exponent of 3 is attached to the floating point number. 445.1 in floating point is therefore:

+4451 03

16.74 must be shifted two places right to make it a fraction. The exponent is therefore 2 and the floating point form of 16.74 is:

+1674 02

To add these numbers write one below the other as for ordinary addition, i.e.:

```
+4451  03
+1674  02
```

Since the exponents are not equal the fraction of the smaller number, +1674 02, must be shifted right a number of places equal to the difference of the exponents, in this case one. The addition now becomes:

```
+4451  03
+0167  03
```

Note that the low order digit of the shifted number was lost in this process. Since the exponents are now equal the magnitudes can be added:

```
+4451  03
+0167  03
+4618  03
```

yielding a sum of 461.8, different by .04 from the true sum of 461.84.

We will get back to this "erosion" of precision a little later in the chapter. Yet another case of floating point addition is shown in example 13-2.

EXAMPLE 13-2

Convert the numbers 9.945 and 1.341 to floating point and find their sum.

9.945 must be shifted one place to the right to become a fraction. Its exponent is therefore 1 and the floating point number is:

```
+9945  01
```

A similar operation is performed on 1.341 to yield:

```
+1341  01
```

Since the exponents are identical the addition can proceed without shifting either magnitude, i.e.:

```
 +9945  01
+1341   01
+11286  01
```

The sum is correct, but there is something wrong here. Adding the two fractions gave a result which is not a fraction. In effect we have generated 1.1286 in the fraction. Since the magnitude of a floating point number is pure fraction this is not a proper result. We must force this result into fractional form by shifting the magnitude right and incrementing the exponent, which yields:

```
+1128  02
```

which is a proper floating point number, but notice that the rightmost digit of the result, the 6, has been lost in the shift. More "erosion".

Yet a third case arises when the numbers to be added have different signs, or when a floating point subtraction is to be done. In this case the subtraction may generate a result which has a zero in the most significant fraction position. This is said to be an **unnormalized** floating point number. A normalized number must have a significant digit in its highest fraction position. This case is shown in example 13-3.

EXAMPLE 13-3

Convert the numbers 11.77 and 2.24 to floating point and perform the subtration 11.77 - 2.24.

11.77 is shifted two places right and the floating

point result is:

+1177 02

2.24 needs only a single shift, with the result:

+2240 01

The subtraction is thus:

```
 +1177  02
-2240  01
```

Since the exponents are unequal the fraction of the smaller number must be shifted right by the difference, in this case 1. This gives:

```
 +1177  02
-0224  02
 +0953  02
```

Since this result is not normalized it must be made normal by shifting the fraction left and decrementing the exponent. The result of this is:

+9530 01

While this difference is exact, the exactness is due only to the fact that the low digit of the smaller number, the subtrahend, was a zero to begin with, and no significance was lost when the zero was shifted out.

You may be beginning by this time to notice that a heavy emphasis is being laid on the loss of precision inherent in floating point methods. This is deliberate. A friend of the writer's once made the remark that if he were a novice and had listened to the

writer discourse on floating point in this vein he would be afraid to use floating point at all. It is not the intention here to scare the reader away from these methods. On the contrary, floating point serves a valid need and is a valuable tool when properly used. The intention here is to convey a healthy respect for the limitations of the method and cause the reader to examine results gotten by floating point methods more critically than might otherwise be the case.

The extreme precision loss in the examples above is due, of course, to the small number of digits in the fraction. Increasing the size of the fraction cuts the loss greatly but the loss still exists, and even more important, it compounds as the number of operations increases. If you are tired of all of this talk of precision loss, there is only one more trial to endure and the subject will be dropped. This is the case of the addition of three numbers. Consider the three numbers:

```
+5135  01  or  5.135
+4074  01  or  4.074
+6381  01  or  6.381
```

We will add these numbers in two different orders and observe the results. Begin by adding the top pair, i.e.:

```
+5135  01
+4074  01
+9209  01
```

Since the exponents were equal the two fractions could be added directly. Now take this result and add the third number to it, i.e.:

```
+9209  01
+6381  01
+15590 01
```

but this result is not proper since an integer digit has been generated. Shifting the magnitude right and incrementing the exponent to compensate yields:

```
+1559  02
```

or 15.59. Now let's add the numbers another way. First add the bottom pair:

```
 +4074  01
+6381  01
+10455  01
```

Since the result contains an integer digit it must be adjusted to:

```
+1045  02
```

Now take the first number and add it to this sum.

```
+1045  02
+5135  01
```

Since the exponents are unequal the magnitude of the smaller number must be shifted right by the difference. The addition now looks like this:

```
+1045  02
+0513  02
+1558  02
```

or 15.58. In other words, adding in different orders produces different sums!

When this material is being taught in a classroom it is always amusing to look around the room as the above result sinks in. It is easy to identify the students who handle money with computers by the horrified expressions on their faces. There is one very obvious moral to the above example. DON'T, DON'T USE FLOATING POINT ON NUMBERS THAT REPRESENT MONEY!

What has happened here is that, in the second case, the first addition resulted in an exponent of 2. When the third number was added a pre-addition shift was necessary, which lost enough precision to produce the difference. In the first case the addition produced a sum which had an exponent of 1. When the third number was added no pre-addition shift was necessary and the sum was, in this case, precise. It was the generation of the larger exponent on the addition of the first pair of numbers in the second case which caused the trouble.

While there is no way around some precision loss in floating point addition and subtraction, the loss can be minimized by intelligent planning. The idea is to have the exponent of the current augend as nearly as possible equal to the exponent of the next addend. This will minimize the amount of pre-addition shifting necessary to perform the addition. This simply means that if there is a way to do it, perform the addition of an array of numbers from smallest to largest rather than largest to smallest or in a random order. In extreme cases this may require the sorting of the numbers to be added before the addition takes place.

The floating point representation disussed so far has allowed numbers from +1000 00 or 1/10 to +9999 99 to be represented. The problem of the negative exponent must now be addressed. Recall that the scientific notation number consisted of four parts, one of which was the sign of the exponent. Nowhere in this discussion has an exponent sign appeared and this problem must now be remedied.

The problem of the exponent sign is solved by dividing the range of the exponent in half, using the lower half for negative exponents and the higher half for positive exponents. This amounts to placing a "bias" on the exponent. In a real decimal floating point system the number pi (3.141...) looks like this:

+3141 51

The true value of the exponent is found by subtracting the bias, 50 in this case, from the given value. The number .7819 looks like this in the biased exponent system:

+7819 50

the real exponent value being the given one minus 50, or zero. The number .01547 looks like this:

+1547 49

Again, the real value of the exponent is the given one minus 50, in this case -1.

Real floating point is, of course, almost always done in binary, and binary floating point is the exact analog of the decimal floating point which has been discussed here. The most common arrangement is to allow 32 bits for the floating point number, four words in the case of eight bit machines like the 6502. One bit of the 32 represents the algebraic sign, eight are usually used for the exponent and 23 are allotted to the fraction. Since the high bit of the fraction is always a one it is usually omitted in the packed representation of the number and appended to the fraction when the number is to be operated upon. This artifice allows one more bit of precision to be squeezed from the scheme. It is usually referred to as the "hidden" bit or the "invisible" bit.

The eight bit binary exponent is divided into two ranges 10000000 to 11111111 representing positive exponents and 00000001 to 01111111 representing negative ranges. A number with exponent 00000000 is considered to be identically zero.

A binary floating point addition proceeds along exactly the same lines as the decimal examples shown before. The difference of the exponents is found and the fraction of the number with the smaller exponent is shifted right a number of bit positions equal to the difference of the exponents. The two fractions are

then added and renormalized if a carry occurs out of the high fraction position. A simple software floating point addition procedure for the 6502 is shown in example 13-4. It should be understood that this example is written for clarity of principle and understanding rather than efficiency. It does not represent methods that would be used in creating a real floating point addition.

EXAMPLE 13-4

Two positive floating point numbers are at EXP1, FR1 through FR1+2 and at EXP2, FR2 through FR2+2. The exponents are at the EXP prefixed locations and the fractions are at the FR prefixed location, lowest order at the lowest memory address. These locations have been defined as follows:

```
label   inst    operand
EXP1    ***             first exponent
FR1     RES     3       first fraction
EXP2    ***             second exponent
FR2     RES     3       second fraction
EXP     ***             result exponent
FR      RES     3       result fraction
```

Perform the floating point addition of these two numbers and leave the result at EXP and FR through FR+2.

```
FAD     LDA     EXP1    get first exponent
        SUB     EXP2    form exponent difference
        BZ      FRADD   if the exponents are the same
                        no shifting is necessary.
        BCS     SHF     if carry is a 1 here EXP1 was
                        larger and the fraction of
                        the smaller number at EXP2,
```

```
                                  FR2 through FR2+2 must be
                                  shifted right by the amount
                                  of the difference.
        LDY     #4                otherwise the two numbers
                                  must be swapped to put the
                                  larger one in EXP1, FR1
                                  through FR1+2
SWAP    LDA     EXP1-1,Y swap the four words
        PHA                beginning at EXP1 with the
        LDA     EXP2-1,Y four beginning at EXP2.
        STA     EXP1-1,Y ...
        PLA                ...
        STA     EXP2-1,Y ...
        DEY                ...
        BNZ     SWAP       ...
        LDA     EXP1       compute shift count
        SUB     EXP2       ...
SHF     TAY                move shift count to Y
SHF2    LSR     FR2+2      shift high order fraction
        ROR     FR2+1      then middle order and carry
        ROR     FR2        then low order and carry
        DEY                last shift?
        BNZ     SHF2       repeat if necessary
FRADD   LDA     EXP1       larger exponent becomes
        STA     EXP        exponent of result
        LDA     FR1        now add the fractions
        ADD     FR2        first the low orders
        STA     FR         save result
        LDA     FR1+1      then the middle orders
        ADC     FR2+1      using ADC to propagate carry
        STA     FR+1       save result
        LDA     FR1+2      then the high orders
        ADC     FR2+2      again using ADC for carry
        STA     FR+2       propagation, save result
        BCC     EOADD      was there a carry out of the
                           high fraction bit?  If so the
                           result fraction must be
                           shifted one bit right and the
                           result exponent incremented.
        ROR     FR+2       ROR is used to bring the
```

```
                                carry back into the high
                                order word.
        ROR     FR+1            shift middle order
        ROR     FR              and low order
        INC     EXP             increment result exponent
EOADD   ...                     program continues
```

Computer designers have, of course, been aware of the problem of loss of precision in floating point operations and have taken various steps to solve the problem. The simplest remedy is to use a longer fraction, thus beginning with more precision and carrying a relatively greater precision throughout the computation. This works very well, but greatly increases the amount of memory required to hold floating point arrays. This development reached its extreme a number of years ago with the introduction of the IBM 7030, the legendary STRETCH machine. STRETCH had a 64 bit wordlength which accomodated both a very large exponent and fraction. It also incorporated a very interesting technique for estimating how much precision was being lost in a given computational situation. This innovation was called "noisy mode" arithmetic.

When a floating point operation results in a fraction which has one or more zeros in the high significance bits the fraction must be shifted left to normalize the number, incrementing the result exponent one for each shift. In this process, however, zeros are introduced into the lower bit positions of the fraction. STRETCH's "noisy" mode was a software switch which introduced ones instead. The problem could then be run both ways and an estimate of the precision loss made by comparing the two results.

Floating point multiplication is a reasonably straightforward process which does not involve the precision loss difficulties encountered in addition and subtraction. The exponent of the result is simply the

sum of the true exponents of multiplier and multiplicand with the exponent bias added. The actual computation involves only the multiplication of the two normalized fractions, the product being another fraction. This fraction may have one zero bit in its high position. If this happens the result fraction is shifted left once and the result exponent decremented.

Floating point division is equally simple in theory, but somewhat more troublesome when it must be implemented in software on a short word machine like the 6502. The exponent of the result is found by subtracting the true exponent of the divisor from that of the dividend and adding the exponent bias. The dividend fraction is then divided by the divisor fraction. A divide fault condition is possible here, but it can be avoided by shifting the dividend fraction right one bit and incrementing the result exponent.

The range of floating point numbers which can be held in a system like the one described here with an 8 bit exponent and a 23 bit fraction is approximately

$$\pm 10^{\pm 38.4}$$

In an effort to increase this range without increasing word length IBM introduced a form of floating point arithmetic on its 360 series machines in April of 1964 which is different from that described here in a fundamental way. To understand what this change meant consider the following binary floating point number:

exponent	fraction
10000001	10000000000000000000111

The true exponent value of this number is 1, obtained when the bias 10000000 is subtracted from the given exponent. The actual binary value of the number is therefore:

$$1.00000000000000000000111_2$$

or about 1.0000008345 in decimal.

In an attempt to increase the range of floating point numbers the exponent was cut from eight bits to seven and given a new interpretation. Instead of the true exponent being a power of two by which the fraction was to be multiplied it was considered to be a power of sixteen. This did in fact increase the range of floating point numbers to about $\pm 10^{\pm 76}$.

There is another effect on floating point numbers, however. In this system a number must be represented by a fraction times a power of sixteen. The number 1 is thus written:

exponent	fraction
1000001	00010000000000000000000

i.e., the true value of the exponent is 1 but this now represents a power of sixteen, and the floating point number 1 must be written as:

$$1/16 \times 16^1$$

The consequence is that floating point numbers cannot be normalized and that there may be as many as three nonsignificant leading zeros. The real precision of this representation is not 24 bits but as little as 21 in the worst case. The number 1.0000008345 written in pure binary floating point above cannot be represented in this system. The inability to normalize numbers crowds the rightmost three bits out of existence.

Floating point systems for microprocessors are implemented in software. No existing machine has floating point hardware capability except via the addition of one of the special chips made for this purpose. These functions must therefore be implemented in software and the quality of the product is severely dependent upon the skill and experience of the

programmer putting it together. It has been this writer's experience with some of these packaged floating point systems that they leave much to be desired and in some cases produce results that are simply wrong. In a number of other cases insufficent attention was devoted to conservation of precision in the design, the result being a much greater erosion of precision than is necessary. If you intend to use one of these systems for important work you would be well advised to test it thoroughly before relying on the results. Caveat Emptor, as the Romans said.

CHAPTER 14

NUMBER BASE CONVERSION

While the fundamental internal processes of the 6502 and other computers are binary, the operating programs written for the machine must interface with the human world, a world whose inhabitants' mental processes are captive to a ten fingered anatomy. It is necessary, therefore, to be able to convert numbers between the binary and decimal systems. The purpose of this chapter is to study the methods by which this is commonly done.

Consider the problem of conversion of BCD to binary. Suppose we have the three BCD digits:

00000010 00000101 00000101

in memory, their values being 2, 5, and 5. The leftmost is the hundreds digit, the next the tens digit, and the rightmost the units digit, so that the value of the decimal number is 255. Our problem is to take these three BCD digits and handle them in such a way that the final result is binary 11111111. A very simple way to do this is to multiply the hundreds digit by 100 and the tens digit by 10. The sum of these two products plus the units digit will give the desired result.

The multiplication is no problem. As we saw in chapter 8 it is possible to construct a shift-add sequence to multiply by any number. The decimal number 100 is in binary:

01100100

The multiplication is done by shifting left twice, saving the result, shifting left three times more, saving this result, shifting left once more and then adding both of the previously saved results. A multiplication by ten was shown in example 8-1. A

program to accomplish this conversion is shown in example 14-1.

EXAMPLE 14-1

Three BCD digits are in memory locations HUNDIG, TENDIG, and UNDIG. They represent the hundreds, tens and units digits of a decimal number. Create the program to convert these digits to a binary number.

```
label  inst   operand
SCNV   LDA    HUNDIG    get hundreds digit
       ASLA             times 2
       ASLA             times 4
       STA    T          save result
       ASLA             shift left three
       ASLA             more times
       ASLA             ...
       STA    T+1        save this result
       ASLA             one final shift gives the
                        original number times 64
       ADD    T+1        add original number times 32
       ADD    T          and original number times 4
                        to get 64N + 32N + 4N = 100N
       STA    T+2        save 100N
       LDA    TENDIG     get tens digit
       ASLA             times 2
       STA    T          save this
       ASLA             times 4
       ASLA             times 8
       ADD    T          10N = 8N + 2N
       ADD    T+2        add the hundreds portion
       ADD    UNDIG      add units digit
       ...              program continues
```

This method will convert the number properly as long as no attempt is made to convert a number greater than 255. Attempting to convert 257 would yield a result of 00000001 and no warning that something had gone amiss. The problem with example 14-1 is that there is no check for overflow.

Further, a special multiplication is required for each decimal digit position. This would become very tedious if large decimal numbers were to be converted. Rather than multiply by a special constant at each stage it would be more efficient to simply multiply by ten and add the next lower significance digit. This amounts to computing:

10(TENDIG+10(HUNDIG))+UNDIG

A subroutine can be created to multiply the A register by ten and check to see whether overflow occurred. This is a simple modification of the program in example 8-1. It is shown in example 14-2.

EXAMPLE 14-2

A subroutine with entry point name TMZTEN is to be written. This subroutine is to multiply the contents of A on entry by ten, returning this result in A with the carry bit off. If the result is too large to be held in A return with carry on.

```
label   inst    operand

TMZTEN  ASLA            multiply by 2
        BCS     TTRTN   if a one bit enters carry at
                        any time during this process
                        the result cannot be held
                        in the A register.
        STA     TT      save intermediate result
        ASLA            multiply by 4
```

```
        BCS   TTRTN   and check for overflow
        ASLA          one last shift
        BCS   TTRTN   check for overflow
        ADD   TT      add saved result
TTRTN   RTS           and return
TT      ***           temporary cell
```

With this subroutine in hand the conversion of example 14-1 becomes much shorter and simpler. The new conversion is shown in example 14-3.

EXAMPLE 14-3

Using the subroutine in example 14-2 write a program to convert the BCD digits of example 14-1. If an overflow occurs send control to location TILT.

```
label   inst   operand

        LDA    HUNDIG   get hundreds digit
        JSR    TMZTEN   multiply it by ten
        BCS    TILT     check for overflow
        ADD    TENDIG   now add tens digit
        JSR    TMZTEN   and multiply by ten again
        BCS    TILT     check for overflow
        ADD    UNDIG    and add units digit
        BCS    TILT     final check for overflow
        ...             program continues
```

The examples above show the simple conversion process, but there is a little more to it than this. The data input to a conversion come, presumably, from the outside world. More than likely they are being

typed by an operator. Even worse, they might be typed by somebody passing by on coffee break when the operator is absent. This might be serious if the typed in data were being used for setting the operating temperature of the reactor core mentioned in chapter 11. Poor Lambros steps out for just a minute because of a hydraulic urgency and the cleaning person decides it might be fun to peck at a few keys on that fancy keyboard! Yet another way to get to meet the Great Fuel Rod Maker in the sky!

The answer, of course, is that ALL input must be checked for errors and plausibility. There are no exceptions, nor can there be any exceptions to this rule. NEVER, NEVER assume that incoming data are correct, and after checking that they are at least correct in form, make a further check for plausibility. 20,000 degrees is a correct, all numeric data entry but hardly a plausible operating temperature for the reactor core.

What is plausible will vary, of course, from situation to situation. An automotive control computer might ask the driver if he really wants to set cruise control at 85 m.p.h. What we can show here is only the check for correctness of form. The data input to a decimal-binary conversion must all be numeric. A check of this kind is known as a **validity check**. The process is shown in example 14-4.

EXAMPLE 14-4

A subroutine is to be written which will check the ASCII character in A on input. If the character is numeric, i.e., in the range $B0 - $B9, the high four bits are to be masked off and the result returned with carry off. If the character is not in this range it is to be returned unchanged with the carry bit on.

```
label   inst   operand

DCMLCK  CMP    #$B0      less than ASCII zero?
        BLS    NDEC      branch if so
        CMP    #$BA      greater than or equal to
                         character after ASCII nine?
        BGE    NDEC      branch if so. If control
                         gets past this BGE the
                         character was valid numeric.
        AND    #$F       strip off high four bits
        CLC              set carry to zero
        RTS              and return
NDEC    SEC              not numeric, set carry to 1
        RTS              and return
```

Note, by the way, that the operand of the AND instruction was written as $F, not $0F. It is never necessary to write nonsignificant leading zeros with this assembler.

The conversions thus far have been for numbers less than 255, not a very useful range. With the short word of the 6502 any useful conversion must be at least double precision. The principle of such a conversion is the same as that for single precision. The mechanical details are a little longer, however. A double precision conversion with validity check is shown in example 14-5.

EXAMPLE 14-5

A program is required to convert a series of ASCII digits from the system keyboard to a number between 0 and 65535, the maximum capacity of 16 bits. A subroutine which reads the keyboard and returns with the ASCII code for the key struck is

available under the entry point name RKB (see example 11-1, page 164). The conversion is to stop if any non-numeric character is received from the keyboard. If an overflow occurs send control to location OFLOW. The result is to be left in normal double precision format in locations N and N+1.

```
label   inst    operand

DCNVT   CLA             clear result cells
        STA     N       ...
        STA     N+1     ...
CNV     JSR     RKB     get character from keyboard
        JSR     DCMLCK  validity check using sub-
                        routine of example 14-4.
        BCS     EOCNV   skip out if not numeric
        STA     NEW     save new digit
        JSR     DBTEN   subroutine below multiplies
                        N, N+1 by ten
        BCS     OFLOW   DBTEN turns carry on if
                        product > 65535
        BCC     CNV     since we know that carry is
                        off we can use the two word
                        BCC instead of the three
                        word JMP.
EOCNV   ...             program continues
```

And the subroutine for multiplication of N, N+1 by ten is:

```
DBTEN   ASL     N       multiply by two
        ROL     N+1     ...
        BCS     EOMP    check for overflow
        LDA     N       save partial result
        LDX     N+1     ...
        ASL     N       multiply by two again
        ROL     N+1     ...
        BCS     EOMP    check for overflow
        ASL     N       multiply by two again
```

```
        ROL   N+1      ...
        BCS   EOMP     check for overflow
        ADD   N        form double sum of 2N and 8N
        STA   N        save in N, N+1
        TXA            high order 2N to A
        ADC   N+1      form high order sum
        BCS   EOMP     check for overflow
        TAX            save high sum in X
        STA   N+1      and in result
        LDA   N        get low order 10N
        ADD   NEW      and add new digit
        STA   N        save low final result
        TXA            get high order 10N
        ADC   #0       propagate carry from low
        STA   N+1      order, save high result
EOMP    RTS            program continues
```

The conversion method shown in example 14-5 can be extended to any necessary multiple precision.

Conversion of decimal fractions to binary can be treated in several ways. The simplest is to convert the successive digits as an integer, all the while keeping count of the number of digits, including leading zeros, which have been processed since the decimal point was encountered. The number of converted digits is the power of ten by which the integer must be divided, or the number of times it must be divided by ten. This division is done by using the highest order of the converted integer as the low order dividend to begin, the high order dividend being zero. This is guaranteed to be divisible, i.e., no divide fault condition can arise. When this division has been done the quotient is stored into the high order result. The second highest integer order is then used as the low order dividend with the remainder from the previous division being used as the high order dividend. This continues through all orders of the integer, the quotient each time replacing the integer order

involved. This process is repeated as many times as required to divide by the proper power of ten. The result is the binary fraction equivalent of the decimal fraction.

Conversions of binary integers to decimal can be done in one of two ways. If the integer is divided by ten the resulting remainder will be the low BCD digit of the decimal result. Successive divisions of the resulting quotient give the higher BCD digits. This process, while simple looking, produces the digits of the result from lowest to highest, an inconvenient order when the digits are being printed as they are converted. One way around this problem is to push the successive BCD digits onto the stack as they are produced. When the conversion is complete the digits can be pulled from the stack in the reverse order, i.e., from highest to lowest order.

A simpler method for converting binary integers to decimal is by subtraction of successive powers of ten beginning with the highest possible power. The number of subtractions possible before zero is crossed, as indicated by the carry bit, is the BCD result for that order. This conversion is shown in example 14-6.

EXAMPLE 14-6

A program is required which will convert the unsigned integer magnitude in A to three ASCII digits to be left in HUNDIG, TENDIG, and UNDIG, whose meanings are as given in example 14-1.

```
label   inst    operand
        LDY     #'0'    ASCII code for zero to Y
S1      SUB     #100    subtract 100 from number
        BCC     S2      did it cross zero?
        INY             no, increment ASCII result
        BNZ     S1      and do it again
```

```
S2      ADC   #100      yes, restore number. Note
                        that we can use ADC instead
                        of ADD here. Carry is known
                        to be zero because control
                        reached here via the BCC
                        instruction above.
        STY   HUNDIG    save ASCII hundreds digit
        LDY   #'0'      get ASCII zero for tens digit
S3      SUB   #10       subtract 10
        BCC   S4        did it cross zero?
        INY             no, increment tens digit
        BNZ   S3        and do it again
S4      ADC   #10       and restore number
        STY   TENDIG    save ASCII tens digit
        ORA   #'0'      form ASCII units digit
        STA   UNDIG     save units digit
        ...             program continues
```

The sharp-eyed reader may have noticed that an economy can be made here. Both of the instructions:

```
S4      ADC   #10
```

and

```
        ORA   #'0'
```

could be replaced by the single instruction:

```
S4      ADC   #$BA
```

If you didn't notice this and don't understand it it would be worth your while to stop and examine it for a moment.

Conversion of signed numbers is best done by forcing the number positive and converting the

magnitude by the methods already developed here. A signed single precision conversion is shown in example 14-7.

EXAMPLE 14-7

A signed number is in the A register. It is to be converted to signed ASCII-decimal, with the ASCII code for the sign, + or -, left in location SIGN and the decimal magnitude left in HUNDIG, TENDIG and UNDIG as before.

```
label   inst    operand
CNVS    LDX     #'+'    ASCII plus sign to X
        PHA             this PHA-PLA sequence sets
        PLA             the flags to reflect the
                        contents of A. The flags were
                        destroyed by the LDX.
        BPL     CNVS2   skip if number positive
        LDX     #'-'    get ASCII minus sign
        TCA             and force number positive.
                        No check is made here for
                        overflow, yet the number
                        could be 10000000, the
                        maximum negative number. Why?
CNVS2   STX     SIGN    save ASCII sign character
        LDX     #'0'    ASCII zero to X
CNVS3   SUB     #100    subtract 100
        BMI     CNVS4   why BMI instead of BCC?
        INX             increment ASCII hundreds
        BNZ     CNVS3   digit and repeat. We can use
                        a two word BNZ instead of a
                        three word JMP here because
                        we know that the INX set the
                        zero flag to show nonzero.
CNVS4   ADD     100     restore number
        STX     HUNDIG  save hundreds digit
```

```
        LDX   #'0'     get ASCII zero again for tens
CNVS5   SUB   #10      subtract ten
        BMI   CNVS6    skip if zero crossed
        INX            increment tens digit
        BNZ   CNVS5    and repeat
CNVS6   ADD   #$BA     if you didn't figure this one
                       out at the end of example
                       14-6, you'd better do it now!
        STX   TENDIG   save tens digit
        STA   UNDIG    save units digit
        ...            program continues
```

Since single precision results are not often useful on an eight bit computer the need for double precision conversions arises quite often. Such a conversion works, of course, on the same principles as the single precision conversions already shown. The mechanics of the conversion, however, become quite messy. Example 14-8 shows such a conversion. You may find that working through this example is fairly heavy going, but don't abandon it without working it all the way through. Quite aside from what it can teach you about the conversion process itself, it draws together a number of skills which you should have mastered by now. It involves indexed addressing, indirect addressing, double precision arithmetic and table access. Working through example 14-8 will be an excellent test of your mastery of these topics.

EXAMPLE 14-8

A double precision unsigned magnitude is in locations N and N+1. Convert this magnitude to ASCII-decimal, leaving the result in DNUMBR through DNUMBR+4, highest order digit in DNUMBR. A scratchpad pointer is available for use at RPTR.

```
label   inst    operand
DCON    LDA     #5          generate digit count. The
        STA     CNT         largest number which can be
                            held in 16 bits is 65535,
                            i.e., 5 digits.
        CDB     RPTR,DNUMBR create scratchpad pointer
                            to result area at DNUMBR
        LDY     #0          clear index for decimal power
                            table access
DC1     LDA     #'0'        set initial digit value
        STA     DIGIT       to zero
DC2     LDA     N           get low order number
        SUB     PT,Y        subtract low order power
                            of ten from table
        TAX                 this is saved in X but not
                            stored in memory, in case
                            this subtraction caused the
                            result to cross zero
        LDA     N+1         get high order number
        SBC     PT+1,Y      and subtract high order power
                            of ten from table
        BCC     DC3         was zero crossed?
        STA     N+1         no, store the result of
        STX     N           the double subtraction
        INC     DIGIT       and increment result digit
        BNZ     DC2         and repeat
DC3     LDX     #0          clear X for indirect access
        STA     *RPTR,X     store result digit in string
        IDB     RPTR        increment store pointer
        INY                 increment index to point to
        INY                 next power of ten in table
        DEC     CNT         last digit?
        BNZ     DC1         not if CNT still nonzero
        ...                 program continues
```

The table of powers of ten is defined elsewhere by the DBL pseudo-op, i.e.:

```
PT      DBL     10000,1000,100,10,1
```

This conversion could be made faster by finding only the first four digits by the double precision subtraction method, since after all of the tens have been subtracted out the residual binary number is less than ten, i.e., it is the BCD low order digit. This can be converted to ASCII by simply ORing $B0 with it. This makes a special case out the low order digit, however, and lengthens the program. Since conversion usually takes place as part of output, and the process is bound by the speed of the output device rather than computational speed, we choose to make the program shorter rather than faster.

The conversion of a binary fraction to a decimal fraction is probably the simplest conversion of all. If a fraction is multiplied by ten the result must be an integer less than ten and possibly a nonzero fraction. Imagine a fixed point number with the point in the middle of the 6502 word, like this:

0000.1100

This is the binary fraction 3/4, a one in the halves bit and a one in the quarters bit. This can be multiplied by ten by the simple procedure shown in example 8-1 to give:

0111.1000

The integer portion of this number, 0111, is the high order BCD result, a 7. Saving the 7 and clearing the high four bits we have:

0000.1000

Multiplying this result by ten gives:

0101.0000

the integer portion, 0101, being the second BCD digit, a 5. We have thus generated the digits 7 and 5 of the decimal fraction .75. Since the remaining fraction is zero there is nothing more to convert.

The process is shown in the above form to avoid grappling with double precision, since multiplying a single precision fraction with the point all the way at the left generates a double precision product. Now that you understand the principle involved you can work through the real single precision conversion. It is shown in example 14-9.

EXAMPLE 14-9

A single precision fraction is in memory at location FRACT. Convert it to an ASCII-decimal character string which represents the decimal fraction beginning at location STRING. Continue the conversion as long as the remaining fraction is nonzero.

```
label   inst   operand
CNVFR   LDY    #0        clear index to store results
CNVFR1  CLA              clear integer space
        STA    T         ...
        ASL    FRACT     fraction times 2
        ROL    T         with bit shifted out going
                         to integer word
        LDA    FRACT     save doubled result
        LDX    T         in A and X
        ASL    FRACT     fraction times 2 again
        ROL    T         ...
        ASL    FRACT     and again
        ROL    T         ...
        ADD    FRACT     now form double precision sum
```

```
STA   FRACT     save for next pass
TXA             get high order back
ADC   T         form high order sum
ORA   #'0'      make ASCII from BCD
STA   STRING,Y  save character in string
INY             increment index for next pass
LDA   FRACT     now test the remaining
                fraction. If nonzero
BNZ   CNVFR1    generate another digit
```

Conversion between decimal and binary floating point is another matter entirely and programs to perform this task are well beyond the scope of a book like this. The general principle involved will be illustrated here but no program examples given.

Consider the binary floating point number:

exponent	fraction
00001000	11111111000000000000000

The value of the fraction is 255/256. The true exponent is shown here, the bias being omitted for clarity. The value of the exponent is 8. The value of the floating point number is therefore:

$$255/256 \times 2^8 \text{ or } 255/256 \times 256$$

or simply 255. Let's rewrite this in decimal and create a new quantity which we'll call DX. BX means binary exponent.

BX	Fraction	DX
8	255/256	0

We begin by subtracting 3 from the binary exponent, BX and multiplying the binary fraction by 4/5. Each time we are able to do this with the exponent value crossing

zero we increment the decimal exponent DX. After one pass the result looks like this:

BX	Fraction	DX
5	255/256 x 4/5	1

The effect of subtracting 3 from the binary exponent is to divide the number by 8. If we concurrently multiply the fraction by 4/5, the net effect on the binary floating point number is to divide it by ten. i.e.,:

$$1/8 \times 4/5 = 4/40 = 1/10$$

We therefore increment the decimal exponent. One more pass yields:

BX	Fraction	DX
2	255/256 x 4/5 x 4/5	2

At this point we can no longer subtract three from the binary exponent BX. When the fraction is consolidated we have:

BX	Fraction	DX
2	51/80	2

The remaining binary exponent can be reduced to zero if we multiply the fraction by four to compensate. We then have:

BX	Fraction	DX
0	51/20	2

The value of 51/20 is 2.55. This result can be gotten from the integer and fraction conversion methods illustrated in this chapter. The decimal value of the binary floating point number is, therefore:

$$2.55 \times 10^2$$

and the conversion is complete. Binary floating point numbers with effective negative exponents are converted by adding 3 to the binary exponent and multiplying the fraction by 5/4, decrementing the decimal exponent each time this is done.

This method is known by the name **radix deflation** and while it is simple in principle it is a veritable minefield of precision loss dangers for the unwary. The multiplication by 4/5, for example, is inexact, since 4/5 cannot be represented exactly in binary. This is for the same reason that 1/3 cannot be represented in a finite number of decimal digits. The purpose of this exercise has been to acquaint you with the principle involved in floating point conversion, not to encourage you to try it. This is a task best left to the professionals. The conversions are easily the most difficult part of the entire floating point process.

CHAPTER 15

GENERATION AND USE OF RANDOM NUMBERS

It is sometimes difficult for a beginning programmer to conceive of any use for randomness. In the well ordered world of a computer program the very notion of randomness would seem to be contradictory to the purpose of programming. Random numbers are, nevertheless, extremely useful and situations exist in which no other approach will yield a result.

It must be understood from the beginning that what are called random numbers in a computer are not random at all, but pseudo random. A computer is an automaton with a finite number of states. The numbers which it generates are of some finite length and, therefore, of finite range. If, as in the case of so-called random number generation, the output from a previous execution of the generator is used as the determining input to a later execution, the program will eventually duplicate one of its previous results and the cycle will start anew. What we mean by a random number in a computer is a member of a series of numbers of fixed range, each of which bears no obvious relation to its predecessor or successor and which, if a sufficiently large number are generated, will be uniformly distributed along the permissible range. Suppose random numbers were to be generated on the interval 0 - 1. In your imagination you can divide this range into some number of subranges, say ten. The first interval runs from 0 to 1/10, the second from 1/10 to 2/10, etc. Rule the subranges onto a piece of graph paper and generate a number. If the number is between 0 and .1 fill in the lowest box of the leftmost column representing the 0 to 1/10 range. If the number is between .5 and .6 fill in the lowest box of the appropriate column and so forth. If you generate enough numbers in this way, filling in the appropriate column for each, the columns should be approximately the same height for all the subranges. Numbers which behave in this way are said to be

uniformly distributed.

To understand how such numbers may be useful, consider the following experiment in your imagination. Imagine that the value of pi, 3.14159..., were unknown and that it was impossible in consequence to compute the area of a circle. Now cut a square from plywood which is exactly ten feet on each side. The area of this square is 100 square feet. Within the plywood square inscribe a circle centered at the exact center of the square with radius 5 feet so that the circle is tangent to the sides of the square at four points. Now we must find a means of hurling a baseball at the square in such a way that the impact point cannot be previously determined and that it is as likely to hit any given point in the ten foot square as any other on any given pitch. We might use a blindfolded individual to throw the ball. Any of the starters on the current Chicago Cubs pitching staff would do nicely, and we could dispense with the blindfold. In this case, however, we would have to add the rule that any pitch which missed the ten foot square altogether wouldn't count and that as a safety measure spectators would be confined to the area behind second base. We now get a supply of baseballs and India ink. Each baseball is dipped in the India ink and hurled at the ten foot square. Each pitch would leave an ink mark on the plywood. After some large number of pitches we stop this nonsense and count all of the marks on the plywood. A second count is done of all of the marks which fell in or on the circle. The ratio of the total number of marks on the plywood to the number which fell in or on the circle is the approximate ratio of the area of the square to that of the circle, and we have determined the value of pi. A further social benefit of such an experiment would be to bolster the self esteem of the pitching staff by proving that even our local heroes are capable of making a contribution. Some good can be found in anything, even if it takes another 35 years to find it. It would seem the humane thing to do in any case. We return now to serious

discussion.

Random numbers are useful in a great variety of situations in which it is impossible to predict the behavior of a system by analytical means, either because no analytical solution exists or because the system is too complex for practical analytical solution. The behavior of such structures as expressways is in general impossible to predict exactly, particularly under full load conditions. It is important to know something of this behavior before committing hundreds of millions of dollars to a plan which may cause a rush hour traffic gridlock unexpectedly. A mathematical model of such a system is therefore built within the computer and the inputs and outputs to the expressway system, i.e., cars entering and exiting, are simulated by routing traffic to on and off ramps randomly but in keeping with known demand. A very good idea can be gotten by this means of how the eventual system will work. An entire group of programming techniques has sprung up around the use of random numbers for such purposes. These techniques are known by the name of **Monte Carlo** methods.

The generation of numbers which exhibit the required pseudorandom traits is surprisingly difficult. Some whimsical programmers are fond of referring to their less than successful programming efforts as "random number generators" but this is really far from the case, indeed it is the nonrandom character of these results which allows the error producing the misbehavior to be traced. The theory of pseudorandom number generation is far beyond the scope of this book. What we will attempt to show here is a fixed procedure for generating pseudorandom numbers by a technique known as the **power residue method**. Those readers who have an interest in the method itself can find it described in numerical analysis texts. R. W. Hamming's Numerical Methods for Scientists and Engineers (McGraw-Hill, 1962) is a good place to start. Since any pseudorandom number generator will eventually repeat the sequence of numbers it produces, the idea is to

make this cycle of repetition, known as the **period of the generator**, as long as possible. For the power residue method the period of the generator is:

$$2^{N-2}$$

where N is the length in bits of the numbers being generated. For the 6502 we will work in double precision, since generating numbers with an eight bit word would yield only 64 numbers before the sequence began to repeat. The method of generation is as follows:

1. A constant multiplier is chosen, as close to $2^{N/2}$ as possible, of the form $8i\pm3$, in which i is an integer. For 16 bit numbers we want a number of this form as close to 2^8 or 256 as possible. The value of i is in this case 16. The seed number may therefore be 253 or 259. We will use 253 here.

2. A "seed" number is chosen. This number may be any odd integer, but it must be odd. This seed number is the starting value for the generator.

3. The 32 bit product of the 16 bit constant multiplier and the 16 bit seed is found. The high order 16 bits of this product are discarded. The low order 16 bits of the product form the generated random number

4. The seed number or starting value is replaced by the generated random number. This replacement occurs on each pass through the generator, so that each pass uses a different "seed" or starting value from the one which preceded it.

This procedure will produce 16,384 numbers before a repeat. A random number generator which uses the power residue method as outlined above is shown in example 15-1.

EXAMPLE 15-1

A subroutine is required to generate random 16 bit unsigned integers by the power residue method just described. The multiplication may be done by the DMPY subroutine of example 9-6, page 131-132. The result is to be left at RNMBR and RNMBR+1.

```
label   inst   operand

RANDOM  LDA    CMPLR     move constant multiplier
        STA    B         to DMPY multiplier area
        LDA    CMPLR+1   ...
        STA    A         ...
        LDA    SEED      move SEED/previous result
        STA    D         to DMPY multiplicand area
        LDA    SEED+1    ...
        STA    C         ...
        JSR    DMPY      execute the multiplication
        LDA    BIGPRD    save low order half of
        STA    SEED      product as new SEED and
        STA    RNMBR     random number result.
        LDA    BIGPRD+1  ...
        STA    SEED+1    ...
        STA    RNMBR+1   ...
        RTS              return to calling program
CMPLR   DBL    253       value of constant
                         multiplier is 253 or 259
SEED    DBL    3         starting value of SEED is
                         any odd integer
RNMBR   $$$              result appears here
```

When using a new random number generator it is good practice to find out something about the uniformity of distribution of the generated results. This need not be an elaborate check, but it should be

done. It is a fairly easy task and makes a good example, since it pulls together a number of different programming concepts from the previous chapters. To make this check a large number of random numbers is generated. As each number comes from the generator its high half is used as in index to increment a count in a frequency distribution table, this table having been previously set to all zeros. The resulting frequency table may then be examined. A good generator should produce no cells containing counts which deviate from the mean by more than 10% to 15%. The creation of a frequency table for the generator in example 15-1 is shown in example 15-2.

EXAMPLE 15-2

The frequency distribution of the numbers generated by the subroutine RANDOM of example 15-1 is to be constructed. Clear an array area of memory to receive the distribution and construct it.

```
label   inst    operand
        CDB     COUNT,-15000 the number of random
                             numbers to be generated for
                             this test is 15,000
        LDY     #0           clear index to zero array
        TYA                  set A to zero. This is
                             shorter than CLA or LDA #0
CLEAR   STA     ARRAY,Y      clear array member
        INY                  bump index
        BNZ     CLEAR        and do it again until done
GEN     JSR     RANDOM       generate a number
        LDX     RNMBR+1      use high half as 8 bit index
        INC     ARRAY,X      this must be done using X,
                             since INC cannot be indexed
                             by Y.
```

```
        IDB     COUNT   increment count
        BNZ     GEN     and do it again until all
                        15,000 have been done.
        ...
```

If 15,000 numbers are generated the average count in each cell of the array should be 15,000/256 or 58.59. When the above program is run and the resulting frequency distribution examined in memory there are more cells with value 58 than any other value. 2/3 of all cells contain values between 55 and 60 inclusive, 4/5 have values between 54 and 61. In no case did any cell contain a value less than 52 or greater than 64. The generator thus passes the test.

The 6502, of course, is indifferent to the imaginary radix point implied by our specification of integer results. We could as well consider that the radix point is all the way at the left end of the word, thus making our results pure fraction, distributed on the interval 0 - 1. This raises a couple of interesting possibilities. The first is the generation of random decimal or BCD digits.

Random decimal or BCD digits are often required in game applications and in programs which generate educational drills, e.g., programs which generate arithmetic problems for school children or choose a word from a list for foreign language vocabulary drill. Random BCD digits are generated by first generating a random binary number and treating it as a pure fraction. This fraction may have a value approaching zero on the low end or approaching one on the high end. Since the binary or radix point is all the way at the left there are no integer bits and the generated number can only approach the value one, not achieve it. If the number so generated is multiplied by ten the result, by definition, is a number composed of one integer BCD digit plus a fraction. This integer part of this number has a value, by definition, less than

ten and greater than or equal to zero. This is our random BCD digit. Once it has been generated in this way it can be converted to ASCII easily by ORing on the numeric ASCII bits, 10110000. The generation of random BCD digits is shown in example 15-3.

EXAMPLE 15-3

A subroutine is required to generate random ASCII numeric digits for use in an elementary school arithmetic drill. The RANDOM subroutine of example 15-1 is to be used and the generated ASCII result is to be returned in the A register.

```
label  inst  operand

RDGIT  JSR   RANDOM   generate binary random
                      fraction
       CLA            and clear result cell
       STA   BCDGT    ...
       ASL   RNMBR+1  multiply the random
       ROL   BCDGT    binary fraction by ten
       LDX   BCDGT    ...
       LDA   RNMBR+1  ...
       ASL   RNMBR+1
       ROL   BCDGT
       ASL   RNMBR+1
       ROL   BCDGT
       ADD   RNMBR+1  form low sum to get carry
       TXA
       ADC   BCDGT    form high sum plus carry
                      from low sum
       ORA   #'0'     change BCD to ASCII
       RTS            return to calling program
BCDGT  ***            temporary storage
```

One parenthetical note should be made about example 15-3 before leaving it. The generator shown will produce a reasonably uniform distribution of random decimal digits, about one digit in ten being a zero. For the arithmetic drill application quoted in the example it turns out that this is an undesirable situation. One zero in each ten digits is much too frequent for this purpose. We are, after all, attempting to teach children arithmetic skills and a zero operand in an arithmetic problem is the trivial case. This writer has found that in implementing such systems zero should be treated as a special case. One zero in forty decimal digits is about right for this application. This requires that a sort of "filter" be built into the generator which discards three zeros out of each four generated. Discarding a zero simply means going back and generating another digit. Writing the program to perform this "filtering" would be an instructive exercise for the reader.

At the beginning of this chapter a random number experiment was proposed which involved ink dipped baseballs and a pitcher who could be relied upon to exhibit perfect randomness of direction. Candidates for the position were even suggested. As an afterthought, the writer realizes that he may have proposed an experiment which would compromise the safety of the observing public. Further, getting a reasonably accurate result would involve several thousand pitches, a feat well beyond the capability of even a winning pitching staff, not to mention the proposed candidates. It has been decided, therefore, to conduct the experiment within the 6502, using random binary fractions rather than baseballs. The Cartesian equation of a circle of unit radius centered at the origin is:

$$x^2 + y^2 = 1$$

The condition that a point (x,y) fall within the circle is:

$$x^2 + y^2 \leq 1$$

Rather than throwing baseballs with random impact coordinates on a wooden board we can generate random coordinates within the computer. A pair of generated random fractions may be viewed as the coordinates of a point within the square whose corners are at (0,0), (1,0), (1,1) and (0,1). This square represents the first quadrant of the square board proposed in the physical experiment. If we generate only unsigned, and therefore positive, pairs of random numbers, these pairs will represent points within this first quadrant. The inequality test shown above will detect points which are within the quarter circle contained in this first quadrant. The ratio of the area of the full square to the full circle in the physical experiment will be the same as the ratio of the small square to the quarter circle contained in the first quadrant. If we determine this ratio we need only multiply it by 4 to find our statistically determined value of pi. The determination of pi by this method is shown in example 15-4.

EXAMPLE 15-4

The value of the constant pi is to be determined by the statistical method outlined above using the pseudorandom number generator RANDOM and the double length multiply routine DMPY. Create the program to perform the necessary computation.

```
label   inst    operand

PI      CDB     TOTAL,-8192 This is the total
                            number of coordinate pairs
                            to be generated. The period
                            of the generator is 16,384
                            but we are accessing it twice
```

```
                                 for each coordinate pair. To
                                 go beyond this number would
                                 only duplicate the result of
                                 the first 8192 pairs.
        CDB     COUNT,0          this is to be the count of
                                 points which fall inside the
                                 circle.
PI2     JSR     RANDOM           generate X coordinate
        LDA     RNMBR            move X coordinate from
        STA     B                generator to multiplier and
        STA     D                multiplicand fields of
        LDA     RNMBR+1          double multiply routine
        STA     A                ...
        STA     C                ...
        JSR     DMPY             now square the X coordinate
        LDA     BIGPRD+2         and save the square
        STA     XSQ              ...
        LDA     BIGPRD+3         ...
        STA     XSQ+1            ...
        JSR     RANDOM           generate Y coordinate
        LDA     RNMBR            move Y coordinate from
        STA     B                generator to multiplier and
        STA     D                multiplicand fields of
        LDA     RNMBR+1          double multiply routine
        STA     A                ...
        STA     C                ...
        JSR     DMPY             square the Y coordinate
        LDA     BIGPRD+2         now add the squares of the
        ADD     XSQ              X and Y coordinates.  If the
        LDA     XSQ+1            sum is 1 or greater, i.e.,
        ADC     XSQ+1            not a fraction, a carry will
        BCS     NEXT             occur out of the high sum.
        IDB     COUNT            this point was within the
                                 circle so we increment COUNT
        IDB     TOTAL            increment loop count
        BNZ     PI2              and do it again if necessary
        ...                      program continues
```

This program takes a number of seconds to execute, since a considerable amount of computation is

involved. When it is executed and locations COUNT and COUNT+1 are examined, they are found to contain $1927, or 6439 decimal. This is the number of points out of 8192 that fell within the circle. The ratio of the area of the quarter circle to that of the square is therefore:

$$\frac{6439}{8192}$$

or .78601074. Multiplying this by four to find the value of pi we get:

pi = 3.14404297

a value quite close to the true one, 3.141592653.

The above example was intended to illustrate a method rather than to present a serious attempt to find the value of pi. As in all statistical determinations a given increase of precision requires that larger and larger numbers of sample points be generated. Nevertheless, example 15-4 nicely illustrates a family of methods which depend upon the generation of pseudorandom numbers for their operation.

The last exercise of this chapter will be concerned with yet another application of random decimal digits. Most people have seen a species of humor which propagates by means of anonymously authored single sheets which mysteriously appear in offices and laboratories. These sheets of paper, looking usually like the 15th Xerox copy, present a variety of humor topics, most of which are too outrageous to appear in normal print. One of these has recently come into the writer's posession and its topic is unusual in being fit to print, funny and germane to the topic of this chapter. The sheet bears no marks of authorship or copyright and was probably generated by one of the army

of amateur humorists who supply city transit stations with the abundant graffiti which are a mark of our time. The sheet is titled:

POCKET FOG GENERATOR

and contains three columns of computer "buzzwords", to wit:

integrated	management	options
total	organizational	flexibility
systematized	monitored	capability
parallel	reciprocal	mobility
functional	digital	programming
responsive	logic	concept
optical	transitional	time-phase
synchronized	incremental	projection
compatible	third-generation	hardware
balanced	policy	contingency

The idea is that the reader of the sheet pick one of the buzzwords from each column to form a sort of "superbuzzword". The intended humor of the piece is resident in the fact that no matter which words are picked, or in what order, the result is a plausible sounding term, of the sort which is heard in the aisles of any technology oriented trade show.

Now, it seems to this writer that some reader could perform a service by applying the random BCD digit generation methods described in this chapter to the automation of the choice of buzzwords from this list. It would be an excellent programming exercise which would hammer home a number of the techniques developed in this and earlier chapters. The simplest approach would be to assign a constant length to each of the buzzwords equal to that of the longest one in any of the three lists. This would make the address computation for the table lookup as simple as possible.

Terms shorter than this maximum length would be padded with blanks to fill the space.

A word would be chosen from each of the lists by generating a random BCD digit, multiplying it by the number of characters in the longest term and using the resulting number as an index for the table lookup. The three words chosen in this way could then be displayed on a single line of the CRT or print device. All possible permutations could be generated and printed.

To carry this idea one step further, the existence of the new electronic speech devices would make it unnecessary to print the generated buzzwords. They could be directly vocalized for all to enjoy. The result might be recorded on cassette for distribution. To take the idea to its conclusion, the sounds of breaking glass and female shrieks might be merged with the buzzword soundtrack to produce an extremely convincing simulation of one of the working sessions of a data processing executive's convention.

The writer sincerely hopes that some reader will find the time and energy to fill in this glaring gap in our technology and culture.

CHAPTER 16

SIMPLE GRAPHIC OUTPUT

The display of data in literal numeric form has the advantage of precision for small amounts of data, i.e., the exact values are known by examination. For large amounts of data, however, the meaning of the data tends to be obscured by its very mass. The effect is literally being "unable to see the forest for the trees". The display of data as a graph lacks the same precision, but allows the meaning of the data, if any, to be grasped by the visual sense rather than the analytical. This greatly enhances the intuitive understanding of some kinds of data.

Preparation of data for graphic display requires that each datum be used to compute a pair of X-Y or other coordinates which fix the location in the display at which the point appropriate to this datum is to appear. This involves setting aside a block of memory to receive the digital "image" of the graph to be displayed. When the image has been built in this memory area, called the **image region**, it is copied line by line to some external device to produce hard copy. Alternately, if the machine is equipped with a hardware assist to enable the image region to be displayed directly on a CRT or other display device, the image will appear point by point as it is built. The Apple II has such a hardware feature which allows four areas of memory to be displayed. Two of these are of very low resolution, 24 by 40 positions. The other two are of much higher resolution, being capable of displaying an image 280 by 192 points. It is of these high resolution displays that we will speak in this chapter. The examples given here will be specific to the Apple II but with sufficent generality to allow them to be modified easily for other types of displays.

In chapter 11 the screen addressing scheme for the Apple was described and methods for computing the screen address of any point given in the examples, in

particular, example 11-4 on page 174. The addressing scheme for the high resolution screen is very similar. The screen is 280 points wide and 192 points deep. Like the text screen desribed in chapter 11, it is divided into three major pages. Each of these major pages displays 64 lines. Each major page is further divided into eight minor pages, each minor page consisting of eight lines. If the memory address of the leftmost word of a line is considered as a 16 bit number with the bits being numbered 0 - 15 beginning at the right, the addressing scheme is as follows:

bits 0 - 2 Always zero

bits 3 - 4 Major page number, 00, 01, or 10

bits 5 - 6 Duplicate major page number

bits 7 - 9 Subpage number within major page, 000-111

bits 10-12 Line number within subpage, 000-111

bits 13-15 Constant, 001 for first screen and 010 for second screen.

As with the text screen in chapter 11, it is possible to compute the address of the leftmost word of a screen line directly from the line number, and by a very similar process. The binary line number, 0 - 191, is picked apart and the pieces used to fill in the bits of the screen line address. This computation is shown in example 16-1.

EXAMPLE 16-1

Create a subroutine which will accept a line number in A and return the screen address of the leftmost word of the line in A and X, high order

in X. If carry is off on entry the address should be in the number 1 high resolution screen which begins at $2000. If carry is on the address should be computed for screen number 2, which begins at $4000.

```
label   inst   operand

HLNADR  PHP            save carry
        PHA            save line number
        PHA            twice
        AND    #$C0    isolate major page number
        STA    LT      and save it
        LSRA           move it over two bits
        LSRA           for duplicate
        ORA    LT      merge with first copy
        STA    LT      and save partial result
        PLA            retrieve line number
        AND    #@70    isolate minor page number
        LSRA           move to position
        LSRA           very,
        LSRA           very,
        LSRA           slowly
        ROR    LT      roll low bit to high word
        STA    LT+1    save partial high result
        PLA            retrieve line number again
        AND    #7      get line # within minor page
        ORA    LT+1    merge partial high result
        ORA    #$20    set address for #1 screen
        PLP            get original carry back
        BCC    HLN2    skip if #1 screen
        ADD    #$20    increment for #2 screen
HLN2    LDA    LT      load result into A and X
        LDX    LT+1    ...
        RTS            return to calling program
LT      $$$            temporary storage
```

The line whose address is returned by subroutine

HLNADR in example 16-1 runs from this address to this address plus 39, each line being 40 words wide. Only the low seven of the eight bits in each word are used in the black and white graphics we will do here. The eighth bit is concerned with color control and will not enter this discussion.

A point is plotted on the Apple II screen by setting the bit which represents that point to one. The easiest way to do this is to store the address returned by HLNADR into a scratchpad pointer, then access the required word in the line using indirect addressing indexed by Y. Don't try to use X for this; it won't work. Y must contain the necessary displacement from the left side of the plot to access the word which contains the desired point. Since there are seven points per word the actual X coordinate must be divided by seven to find this displacement. The remainder which results from this division represents the bit displacement within the selected word. The bits in the display word are numbered left to right rather than right to left as is done with numeric data. The bit plotting scheme within the word looks like this:

Memory Contents	Appears as 1234567
1	*
2	*
4	*
8	*
16	*
32	*
64	*

If more than one bit in a word is set to 1 the corresponding points will appear as bright points on the screen. Setting a word to $7F will cause all seven points represented by that word to appear bright.

We must now face the problem of how to compute the word displacement on the line and the bit displacement within the word. The word displacement is computed easily enough by simple division, i.e., divide the X coordinate by seven. The quotient is the displacement and the remainder is the bit displacement within the word. This is simple enough but very, very slow and speed does become a major consideration in some types of graphics problems. Some means must therefore be found for dividing rapidly by seven. The problem is complicated by the fact that the dividend can be double precision, since the X coordinate may be greater than 255. This is taken care of as a special case. The division is accomplished by successive subtraction, but a special kind of successive subtraction. Since it is equally likely that the point to be plotted will be on the left half of the screen as on the right we begin by subtracting 140. If this subtraction does not cause a zero crossing we may increment our quotient by 20, since we now know that the point lies on the right half of the screen. If the subtraction causes a zero crossing we know that the point is on the left half of the screen and add the 140 back, leaving the provisional quotient zero. Having established which half of the screen the point lies in, we subtract 70 in the same manner to narrow the area to the nearest quarter of the screen. This process continues until we have located the word within four words. Seven is then subtracted successively until zero is crossed, the quotient being incremented by one for each successful subtraction. After zero is crossed the last seven is added back. This results in what would be the remainder in a normal division. This remainder represents the bit position of the point within the word to be indexed by the quotient. This remainder may then be used as a shift count to move a bit into the proper position, but this is slower than we would like, so the proper bit is looked up directly in a table using the remainder as an index. The process is shown in example 16-2. This example is a tedious and complex

exercise, but working it through will help hone your 6502 arithmetic skills.

EXAMPLE 16-2

Create a subroutine which will accept a binary X coordinate 0 - 279 in A and X, high order in X and return the word displacement of the point to be plotted in Y. A should return containing a single one bit in the position to be plotted.

```
label   inst    operand
XC      LDY     #0          clear Y to receive quotient
        DEX                 if the X coordinate is
                            greater than 255 the X
                            register will contain a 1.
                            The special case can thus be
                            tested for by decrementing X.
                            If the result is zero the X
                            coordinate was greater than
                            255.
        BNZ     X2          test for special case
        SUB     #252        this subtracts out the
                            largest multiple of 7 in 255,
                            i.e., 36 times 7.
        LDY     #36         set provisional quotient
        BNZ     XC8         use BNZ instead of JMP
                            because the LDY above cleared
                            the zero flag
XC2     SUB     #140        left or right half?
        BCC     XC3         check result
        LDY     #20         right half, set quotient
        BNZ     XC4         skip restoring ADD
XC3     ADC     #140        ADC can be used because
                            carry is guaranteed to be 0.
XC4     SUB     #70         now identify quarter
        BCS     XC5         test result
```

```
        ADC   #70       restore dividend
        JMP   XC6       skip to 8ths stage
XC5     PHA             save accumulator
        TYA             get quotient
        ADD   #10       increment for quarter
        TAY             restore quotient to Y
        PLA             get accumulator back
XC6     SUB   #35       identify 8th of screen
        BCS   XC7       test result
        ADC   #35       restore dividend
        JMP   XC8       and go to final subtract
XC7     PHA             save accumulator
        TYA             get quotient
        ADD   #5        increment for eighth
        TAY             restore quotient
        PLA             get accumulator back
XC8     SUB   #7        now keep subtracting 7
        BCC   XC9       until zero is crossed
        INY             incrementing the quotient
        BNZ   XC8       each time
XC9     ADC   #7        restore to get remainder
        TAX             remainder to index
        LDA   BTTBL,X   get bit from table
        RTS             return to calling program
BTTBLE  DATA  1,2,4,8,16,32,64 bit position table
```

By using the HLNADR routine of example 16-1 and the XC routine of example 16-2 we can now address any point on the Apple screen.

The line numbers on the Apple screen begin at the top, forcing the user to think of the screen as representing the third quadrant of a four quadrant system. This is awkward but very easy to fix. We would like the logical first line, i.e., line 0, to be at the bottom so that the screen represents the first quadrant rather than the fourth. This is a very simple operation, shown in example 16-3. It consists of subtracting 191, the highest possible line number from

the number of the line to be operated upon. This result is then simply twos complemented to achieve the required result.

EXAMPLE 16-3

Create a subroutine named INVERT which will invert the line numbers on the LOWER Apple screen which begins at $2000, making the bottom line zero and the top line 191. Return with the address of the leftmost character of the line in A and X, high order in X.

```
label   inst    operand

INVERT  SUB     #191        subtract largest possible
                            line number. We now have a
                            number between 0 and -191.
        TCA                 twos complement
        CLC                 clear carry for lower screen
        JSR HLNADR          and get the address
        RTS                 return to calling program
```

We can now draw all of this together into a scheme to plot a point anywhere on the screen. Having laid all of the groundwork, this is an easy task. It is shown in example 16-4.

EXAMPLE 16-4

Using the subroutines developed in the first three examples of this chapter, write a subroutine which will take a Y coordinate from a memory location named Y and an X coordinate from a double memory location X, X+1 and plot the point on the screen.

```
label   inst    operand

PLOT    LDA     Y         get Y coordinate
        CMP     #192      is it on the screen at all?
        BGE     PLRTN     test
        JSR     INVERT    Y OK, get the address
        STA     SPTR      save in scratchpad pointer
        STX     SPTR+1    ...
        LDA     X         get X coordinate
        LDX     X+1       if X+1 is zero the point is
                          on the screen.  If X+1 is not
                          zero the low half must be
        BZ      PL2       tested
        CMP     #24       is the X coordinate greater
                          than or equal to 256+24?  If
                          so it is not on the screen.
        BGE     PLRTN     test
PL2     JSR     XC        compute word and bit location
        ORA     *SPTR,Y   OR with existing contents of
                          this word.  To simply store
                          the bit into this word would
                          clear any bits that had been
                          previously plotted in it.
        STA     *SPTR,Y   store the merged result
PLRTN   RTS               return to calling program
        ...

The scratchpad pointer is defined by:

        SPD     $80       set scratchpad location
SPTR    $$$               reserve pointer space
        ...
```

Just as important as the plot function itself is the need to erase points. This need arises when the plotted display is to change with time in such a way as to show a progression of states, the old image then being erased before the new one is displayed. The

erase function involves the same address computation as the plot function. It is shown in example 16-5.

EXAMPLE 16-5

Construct an ERASE subroutine to clear a screen point whose location is in memory at X, X+1 and Y as in example 16-4.

```
label   inst    operand

ERASE   LDA     Y         get coordinate as before
        CMP     #192      check range
        BGE     ERTN      skip if not in range
        JSR     INVERT    compute line address
        STA     SPTR      save in scratchpad pointer
        STX     SPTR+1    ...
        LDA     X         get X coordinate and check
        LDX     X+1       for range as before
        BZ      ER2       ...
        CMP     #24       ...
        BGE     ERTN      ...
ER2     JSR     XC        compute word and bit position
        CMA               invert the contents of A for
                          use as a mask to "AND" out
                          the bit to be erased.
        AND     *SPTR,Y   AND memory word with mask.
                          this preserves all bits in
                          the word except the one to
                          be obliterated.
        STA     *SPTR,Y   now put the word back
                          into memory
        RTS               and return
```

SPTR is defined as in example 16-4.

The consideration of speed in plotting was raised during the discussion of the computation of the word and bit displacements for the X coordinate. In some kinds of graphic applications which display motion such great speed is required that even more extreme measures are necessary than those shown in the X coordinate computation. The simulation of the appearance of stars flashing by is one of these situations, since a considerable amount of trigonometric computation is required to determine the moment to moment position of each star on the screen. The time overhead involved in the computation of the line addresses by HLNADR or INVERT can thus become the limiting factor in such a display. Think for a moment about the complications involved in such a display.

Suppose you were attempting to simulate motion toward the constellation of Orion. The center of the screen would be occupied by the four stars in the shoulders and knees of Orion, plus the three in his belt and perhaps a couple of stars to represent the sword and scabbard hanging from his belt. Coping with this situation might not be too difficult but just expanding Orion outward toward the edges of the screen would not be a very realistic display. The brightest object in the entire sky, Sirius, is just to Orion's left and could hardy be omitted. Just above Sirius is Canis Minor, also containing a first magnitude star, Procyon, and just to Orion's right is the constellation of Taurus containing Aldebaran, another first magnitude star, and the two galactic clusters, the Pleiades and Hyades. As the forward motion proceeded the fainter stars in all of these constellations would have to become visible, as well as the Orion nebula and the individual members of the two galactic clusters. All in all, this is a very tall graphics order which requires very fast manipulation of the moving points on the screen. Remember also that for each new position plotted the old one must be erased.

Situations like this require that there be an absolute minimum of computation associated with the

placement of each point, and computing overhead that can be done away with must be avoided. In computing, as elsewhere, one very seldom gets anything for nothing, and this is a typical situation in which a tradeoff must be made. In this case space is traded for speed. What can be eliminated here is the execution of the HLNADR or INVERT routines which compute the beginning address of the line. These addresses can be precomputed in a few milliseconds before the plot is started and looked up in a table as required. This table lookup is much faster than computation, but the table requires space, two words for each of the 192 addresses, or 384 words. The generation of such an address table is shown in example 16-6.

EXAMPLE 16-6

Construct a subroutine to compute the beginning addresses of all 192 lines on the Apple screen, assembling these addresses in two tables. The LOADR table should contain the least significant words and the HIADR table the most significant. The table should allow addressing of the Apple screen as a first quadrant rather than a third quadrant.

```
label   inst   operand
GNTABL  LDY    #0          running line number to Y
GT2     TYA                copy line number to A
        JSR    INVERT      compute address
        STA    LOADR,Y     place low address in table
        TXA                move high address to A. This
                           is necessary because the STX
                           instruction cannot be indexed
                           by Y in general memory, only
                           in scratchpad.
```

```
        STA   HIADR,Y place high address in table
        INY           increment count/index
        CPY   #192    finished yet?
        BNE   GT2     do it again if not
        RTS           return to calling program
```

The two tables LOADR and HIADR are defined:

```
LOADR   RES   192
HIADR   RES   192
        ...
```

As tedious as this might seem, it completely eliminates the need to compute addresses in real time. The required addresses are fetched from the table with only four instructions, as contrasted with the 36 instructions required by INVERT, including the calling JSR. The appearance of the PLOT routine in example 16-4 changes only slightly, but its speed is greatly enhanced. The plot function done by this technique is shown in example 16-7.

EXAMPLE 16-7

Perform the plotting function defined in example 16-4 by the table lookup method just described, using the tables previously generated by GNTABL.

```
label   inst   operand
PLOT    LDY    Y         the Y coordinate now goes
                         directly to the Y register
                         for use as an index rather
                         than as an input argument to
                         INVERT or HLNADR
        LDA    LOADR,Y   get low address
```

```
STA   SPTR     place in scratchpad pointer
LDA   HIADR,Y  get high address
STA   SPTR1    place in scratchpad pointer
LDA   X        get low X coordinate
LDX   X+1      and high X coordinate
JSR   XC       compute word and bit
                positions as before
ORA   *SPTR,Y  OR with previous contents
STA   *SPTR,Y  and plot the new point
RTS            return to calling program
```

The ERASE routine changes in an identical way, so it will not be repeated here. Table lookup techniques can also be applied to the determination of the X coordinate for even greater speed, but this is a messy affair requiring indexing in a double precision table. This would make a good exercise for any interested reader who has really stringent speed requirements.

A CRT display is a volatile thing which vanishes when power is removed or the screen is used to display something else. It can be saved on disk but this requires that the disk operating system be invoked every time the display is to be examined. Clearly some means of hard copy is desirable.

There has emerged in the market recently a species of printer which is capable of printing graphics as collections of dots, much as they are represented on a CRT. One of these is the Victor 5080, manufactured by the Victor Data Products Division of Victor Business Products. This printer behaves as any normal printer would as long as ASCII codes are sent to it. Sending certain non-ASCII codes to it, however, allows it to escape into an operating mode in which every dot on a line can be struck in groups of five lines. Thus a five line dot matrix can be printed all across the page. A special line feed mechanism is provided which allows a partial line feed which positions the paper in such a way that the top row of dots of the following

line is immediately adjacent to the bottom row of dots of the line above it. A plot of any length whatever may thus be made in groups of five lines. The printer has 480 possible dot positions, considerably more horizontal resolution than the Apple screen.

This printer has a number of standard interfaces built in. The one used here is the parallel interface used with the standard Apple parallel interface card in slot number 3. The examples which follow will contain this assumption. Changing slots will amount only to changing the EQU's which define the data and sense locations and reassembling.

The graphics mode is entered on this printer by sending the word $18 to the interface. For the duration of the line in progress all data sent to the printer are interpreted as graphic unless a non graphic word is received. A graphic word sent to the printer has the following format:

bit number	7	6	5	4	3	2	1	0
contents	x	1	x	d	d	d	d	d

in which x means irrelevant and d means a data bit. The number 6 bit must be a one as shown. The 0 bit position corresponds to the top row of dots and the 4 bit position to the bottom. After sending the graphics code $18, as noted above, sending out the five words:

x1x00001
x1x00011
x1x00111
x1x01111
x1x11111

followed by a normal ASCII carriage return, $D or $8D will result in the following dot display in the leftmost five columns of the paper:

```
*****
 ****
  ***
   **
    *
```

while the group:

```
x1x00001
x1x00010
x1x00100
x1x01000
x1x10000
x1x01000
x1x00100
x1x00010
x1x00001
```

followed by carriage return results in:

```
*       *
 *     *
  *   *
   * *
    *
```

Up to 480 frames of data may be sent in this way to form one graphic line. The printing is initiated by receipt of either a carriage return or the 480th frame of data. The remainder of this chapter will be taken up with the development of means of copying the contents of the Apple high resolution graphics screen to this printer. This task is necessarily somewhat complicated, made more so by the nonsequential addressing of the lines of the Apple screen. In many ways this is the most complicated single example in the book, but you are urged not to abandon it. If you can get through this example you will significantly improve your understanding of 6502 assembly level programming. This will also serve as an excellent test of your

mastery of the material from the previous chapters. It draws many skills together and forces you to understand all of the component skills in order to understand the whole. Don't give up now, this is the main part of the show.

In the example which follows we begin copying the screen to the printer from the top. To do this we need to generate the addresses of the first five Apple screen lines in scratchpad pointers. The pointers are used successively to access a single bit from each of the five lines. The five bits so collected form one vertical print frame. The 280 print frames, preceded by the graphics escape character, $18, and followed by the carriage return, $D or $8D, and the special line feed character $1A are assembled into a buffer. The buffer is then sent, frame by frame, to the printer. The process is then repeated for the next five screen lines. This continues until the bottom of the screen is reached. Copying the Apple screen to the Victor 5080 graphics printer is shown in example 16-8.

EXAMPLE 16-8

Create a subroutine RCOPY to print a hard copy of the contents of the Apple High resolution graphics screen number 2 on the Victor 5080 as per the above specification.

```
label   inst   operand

RCOPY   CLA            SCNT is the number of the
                       top line of the group of 5
                       currently being processed
        STA    SCNT    for printing
RC2     LDY    #0      clear index for scratchpad
                       pointer storage
RC3     LDA    SCNT    get top line number
        SEC            HLNADR will return #2 screen
```

```
                              addresses if carry is on
        JSR   HLNADR          get line address
        INY                   this sequence is necessary
        STX   SPTR1,Y         because the 6502 cannot store
        DEY                   direct indexed by Y into a
        TAX                   scratchpad location
        STX   SPTR1,Y         save low pointer
        INY                   move index to point to
        INY                   next scratchpad pointer
        INC   SCNT            increment line number
        CPY   #10             all 5 pointers generated
        BLS   RC3             yet?
RC4     LDY   #0              yes, clear word index to
        STY   WINDEX          point to leftmost word
        CDB   SPTR6,BUFFER    this creates a scratch-
                              pad pointer to the buffer
        CDB   COUNT,-283      this is the total count
                              of frames to be output to the
                              printer; graphics escape
                              code, 280 data frames,
                              carriage return and line feed
        LDA   18              graphics escape code to
        JSR   STORE           buffer
RC5     LDA   #7              this is the bit count within
        STA   BINDEX          the word being processed.
        LDA   *SPTR1,Y        get word from 1st line
        STA   W1              save it
        LDA   *SPTR2,Y        get word from 2nd line
        STA   W2              and save it
        LDA   *SPTR3,Y        and so forth for all
        STA   W3              five words
        LDA   *SPTR4,Y        ...
        STA   W4               ...
        LDA   *SPTR5,Y        ..
        STA   W5              at this point the first word
                              of each of five lines has
                              been fetched. On succeeding
                              passes the second, third, etc.
                              will be fetched until all 40
                              words on the five lines are
```

```
                              exhausted.
RC6     CLA                   clear A to receive data frame
        ROR     W5            get bit from bottom line
        ROLA                  and roll into A. Remember
        ROR     W4            that the leftmost point of
        ROLA                  the screen displayed word
        ROR     W3            is the rightmost bit of the
        ROLA                  word in screen memory
        ROR     W2            ...
        ROLA                  ...
        ROR     W1            ...
        ROLA                  at this point one data frame
                              has been assembled in the low
                              five bits of A.
        ORA     #$40          now force the #6 bit on as
                              required by the 5080 printer
        JSR     STORE         and store the frame in the
                              buffer
        DEC     BINDEX        done with these five words?
        BNZ     RC6           if not do it again
        INC     WINDEX        done, go to next 5 words
        LDY     WINDEX        check for end of line
        CPY     #40           done with this five line
        BNE     RC5           group?
        LDA     #$D           yes, add carriage return
        JSR     STORE         to buffer
        LDA     #$1A          and special half line feed
        JSR     STORE         ...
        JSR     PRT           now print the line
        LDA     SCNT          end of screen?
        CMP     #190          ...
        BLS     RC2           if not do 5 more lines
        RTS                   return to calling program
```

The STORE subroutine which assembles the data and special control characters into BUFFER is:

```
STORE   LDY     #0            clear index for store
        STA     *SPTR6,Y      store data frame
        IDB     SPTR6         increment pointer
```

```
        RTS             and return
```

And the actual physical output routine is:

```
PRT     CDB   SPTR6,BUFFER  set pointer for
                          buffer access
        LDX   #0          clear both index registers
                          Y will be used to access the
                          buffer and X to access the
        LDY   #0          printer locations.
PR2     LDA   *SPTR6,Y get data frame from buffer
        STA   PRTR,X  indexing by X is required by
                          the Apple hardware
PR3     LDA   PRFLAG,X get ready flag, also by X
        BMI   PR3         high bit of this word is
                          printer ready flag
        IDB   SPTR6       increment buffer access
                          pointer
        IDB   COUNT       increment frame count
        BNZ   PR2         do it again if not zero
        RTS               return
PRTR    EQU   $C0B0       printer in slot 3
PRFLAG  EQU   $C3C1       ...
```

The data buffer definition is:

```
BUFFER  RES   283
```

The pointers and data words in scratchpad are defined:

```
        SPD   $80
SPTR1   $$$
SPTR2   $$$
SPTR3   $$$
SPTR4   $$$
SPTR5   $$$
SPTR6   $$$
W1      ***
W2      ***
```

```
W3      ***
W4      ***
W5      ***
SCNT    ***
BINDEX  ***
WINDEX  ***
```

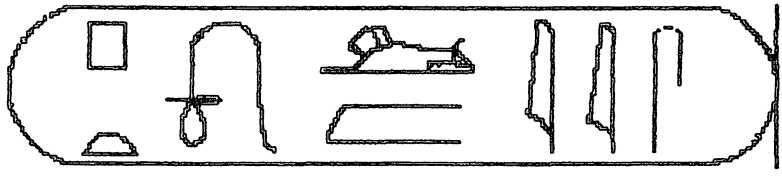
A printer having the capability of printing individual dots opens up a large number of possibilities. It can, for example, be made to print any typeface whatever, however exotic. In an earlier volume in this series, PRACTICAL MICROCOMPUTER PROGRAMMING: THE INTEL 8080 a dot matrix printer was used to print the Cyrillic (Russian) type face by means of selection of the matrix representation of the Cyrillic character to be printed. That effort was limited by the fact that the printer could only execute a full line feed, leaving a gap between lines which were seven dots high. The added capability present in the 5080 removes all of the previous restrictions.

It would be an instructive exercise for the reader to pursue methods for the generation of tables to assemble and print an exotic typeface. A great deal of experience with table lookup methods is generated in this way.

One of the most beautiful of all scripts was the tomb writing of ancient Egypt, known as the hieroglyphs. A sample of this writing is shown on the next page. This image was created manually using an Apple Graphics Tablet and copied to the printer with the program shown in example 16-9. It represents what is called the "cartouche" of King Ptolemy, one of the kings of ancient Egypt. The word cartouche is a French term for the oval which encloses the symbols. It was the custom in hieroglyphic writing to so enclose the names of royal persons. This cartouche, along with that of Queen Cleopatra, was one of the keys which allowed this ancient writing to be deciphered with the

aid of the famous Rosetta Stone. The writing had long defied efforts to decipher it because it was assumed that the symbols were literal pictograms, each symbol standing for an object or concept. The Rosetta Stone, found by Napoleon's soldiers in 1799 during the French expedition to Egypt, contained the same text in three different language forms, two of which were Greek and Hieroglyphic. The corresponding positions of the proper names Ptolemy and Cleopatra in the Greek and Hieroglyphic led to the identification of some of the Hieroglyphic symbols as having alphabetic value. The square in the cartouche is a p, the half circle a t and the lion an l, for example. The complete decipherment of the tomb writing followed rapidly after this. The individual credited with the major part of the work was a young Frenchman named Jean Champollion.

It would be a formidable programming challenge to create a graphics scheme to print text in this ancient language. Any reader interested in following up on this idea should consult Egyptian Grammar by Sir Alan Gardiner (Griffith Institute, Ashmolean Museum, Oxford), from which the Ptolemy cartouche was taken.



CHAPTER 17

ELEMENTARY CRYPTOGRAPHY

The word cryptography comes from two Greek words which mean secret writing, and the notion of secret writing has fired men's imaginations since ancient times. Many methods of concealing the content of a message have been used, some even concealing the fact that a message exists. One ancient method was to wrap the writing medium in a spiral around a cylinder and write the message on it. When the material was unwound from the cylinder it appeared as hash. Only by wrapping it around a cylinder of the same diameter could the message be read by its receiver. Alexander the Great is reputed to have sent messages by shaving the head of a slave, tatooing the message on his scalp, waiting for the hair to regrow and sending the slave on his way disguised as an ordinary traveler. The receding hairline which accompanies middle age eventually unmasked the spy, but by the time this happened, presumably, the information would be of no use to an enemy. Readers who have an interest in this subject should acquire a copy of a marvelous book by David Kahn, The Codebreakers. It is long but so well researched and detailed that it is difficult to put down.

Modern cryptography is done by means of **codes** and **ciphers**. A message is encoded by transliterating it word for word from a code book. The receiver of the message simply performs the reverse process to retrieve the original message. This requires that both sender and receiver have copies of a rather large reference book, making the method unsuitable for surreptitious operation. In particular, if the code book falls into an opponent's hands the whole game is over. Further if the code book is in the opponents hands and you don't know it, he can now read all of your communications in the clear and by applying some leverage to one of your operatives feed false information to you. So much for

bulky code books. The next time you see a man in a trench coat carrying a large book and leaning against a lamppost reading a newspaper with two holes cut in it, he's probably not a KGB agent. We won't get into the question of the meaning of the trench coat here.

Modern cryptography also uses a method of message concealment known as a **cipher**. The process of transforming a message to its concealed form is known as **encipherment**, that of retrieving the message from its enciphered form is known as **decipherment**. The message in its original readable form is known as the **plaintext** and the enciphered form is called the **ciphertext**.

The principal form of modern ciphers is known as a **substitution cipher**, and it is this form that will concern us in this chapter. A message is enciphered by substitution for each of its characters another character, according to some rule known to both sender and receiver. The receiver deciphers the message by performing the inverse substitution. This sounds very simple and, in principle, it is. Children everywhere love the idea of "secret messages" and use a simple substitution cipher to write them. Typically this involves turning the alphabet end for end resulting in the substitution of Z for A, Y for B, X for C, and so forth. The note is then passed around the classroom to the sender's cronies and, until intercepted by the disapproving teacher, is the cause of great excitement to the receivers as they enjoy being the sole possessors of the "secret" information. An imaginative teacher could create a real learning opportunity by showing how to decipher such messages rather than simply throwing them into the wastebasket with a stern admonition, as is the usual case. This should be done with some judgment as there may be some embarrassment caused if the message says something like "Johnny loves Susie".

A scheme like this in which a constant substitution of one character for another is done is known as a **monoalphabetic cipher**. Such cipher schemes

have been used in fairly modern times. In the time of Queen Elizabeth I, when Elizabeth held her cousin Mary Stuart (Queen of Scots) prisoner, Mary used a monoalphabetic cipher to send and receive messages from her forces in Scotland, who were planning to depose and, it was alleged, assassinate Queen Elizabeth. Monoalphabetic encipherment is extremely easy to do and it may have been this that led Mary to use it. A 6502 program for monoalphabetic encipherment is shown in example 17-1.

EXAMPLE 17-1

Create a program to perform monoalphabetic encipherment on a string of characters from the Apple keyboard. The plaintext should be displayed on the lowest line of the Apple screen and the ciphertext on the line just above it. When the line has been filled begin again at the left after clearing both lines. Ignore all characters except A through Z.

```
label   inst   operand
MACFR   LDY    #39       set index to clear screen
        LDA    #' '      lines to blank
MA1     STA    LAST,Y    clear last line
        STA    NTL,Y     clear next to last line
        DEY              ...
        BPL    MA1       ...
        LDX    #0        clear index for line pointer
MA2     JSR    RKB       get character from keyboard
        CMP    #'A'      check for alphabetic
        BLS    MA2       ...
        CMP    #$DB      $DB follows 'Z' in ASCII
        BGE    MA2       ...
        STA    LAST,X    store plaintext character
        AND    #$1F      when the top 3 bits of an
```

```
                        ASCII alphabetic character
                        are set to zero the result
                        is a number 1 - 26, the
                        position of that letter in
                        the alphabet.
        TAY             move number to index
        LDA   CHRTBL-1,Y get ciphertext character
        STA   NTL,X     display it on next to last
        INX             line, increment index
        CPX   #40       last character?
        BLS   MA2       not yet
        BGE   MACFR     yes, clear and start over
LAST    EQU   $BD0      last screen line
NTL     EQU   $B50      next to last screen line
CHRTBL  ASC   'MZUCPNGOALBWYERQHISDKJVTXF'
```

CHRTBL is the hashed alphabet to be substituted, M being substituted for A, Z for B, U for C and so forth.

Using the program of example 17-1 the text:

DO YOU KNOW MY BEARD IS MADE OF YAK HAIR
NO BUT HUM A FEW BARS AND I'LL FAKE IT

becomes:

CRXRKB ERVYXZPMICASYMCPRGXMBOMAI
ERZKDOKYMGPVZMISMECAWWGMBPAD

which would seem to be such total chaos that it could never be deciphered by somebody not having the cipher alphabet as a key. But things are not as they seem.

Returning now to the saga of Mary Stuart, the political disagreement which had made her Elizabeth's prisoner was over a religious question. Mary was Catholic and Elizabeth and England were Protestant, England having been made Protestant by Elizabeth's father, Henry VIII. Mary was sending these

monoalphabetic enciphered messages to her allies in Scotland by means of a Catholic courier, a man named Gifford who was allowed to visit her. The messages were smuggled out, so the story goes, in a beer barrel. Gifford was then sending the messages on their way by means of intermediaries, who also brought the replies. This would have worked out if Gifford had not already been compromised by Elizabeth's security chief, Sir Francis Walsingham, one of the less pleasant people in English history. Instead of sending and receiving Mary's messages directly, as Mary thought he was doing, both outgoing and incoming messages were first passed to Walsingham. Walsingham was a fanatical anti-Catholic whose ideas on security and prisoner interrogation would make a KGB agent blanch, but he was also a very clever man. He deciphered Mary's correspondence almost instantly. From that point on Mary's fate was sealed. Walsingham allowed the correspondence to continue for about two years, copying and deciphering all of it. He then presented the evidence to Elizabeth who, after a star chamber trial, signed the warrant for Mary's execution by beheading. Politics was a rough game in those days.

If Mary Stuart were alive to talk she would, I am sure, testify most eloquently to the inadequacy of the monoalphabetic cipher. Walsingham had had no trouble whatever breaking her cipher, and it cost Mary her head. Let's see how Walsingham broke the cipher so easily, and then construct a 6502 program to do the same thing.

If a large sample of English text is examined and the frequency of appearance of the letters of the alphabet determined, a curious thing appears. About one letter in seven is an E. About one letter in ten is a T, one in twelve an A or O, and about one in fifteen an N or R. It does not matter what kind of text is involved. Whether the frequency determination is performed on a Shakespeare sonnet, a technical book like this one, or Lady Chatterly's Lover, the result is the same if enough text is involved. This means that

for a message of any length, and political conspiracy is verbose even if unsuccessful, these "natural" frequencies of the letters will emerge. A table of these "natural" frequencies is shown in example 17-2.

EXAMPLE 17-2

Frequency of appearance of letters in English text.

Letter	Freq./1000 letters
E	131
T	105
A	82
O	80
N	71
R	68
I	63
S	61
H	53
D	38
L	34
F	29
C	28
M	25
U	25
G	20
Y	20
P	20
W	15
B	14
V	9
K	4
X	2
J	1
Q	1
Z	1

The first step in "breaking" a monoalphabetic cipher is to determine the frequencies of appearance of all the letters and symbols which appear in it. The one which appears most frequently is tentatively E, and the second most frequent is tentatively T. The process from this point on becomes somewhat intuitive. If you see the combination:

T@E

where @ is an unknown letter, you might strongly suspect that the unknown symbol @ stands for H. This suspicion would be reenforced if this combination appeared very many times in the message. It could still be O for TOE or I for TIE. When a few letters have been identified this way the rest fall into place very rapidly. This was the scheme used by Walsingham to break Mary Stuart's simple cipher. A program to determine the frequency of appearance of the characters in a ciphertext is shown in example 17-3.

EXAMPLE 17-3

A one thousand word buffer begins at location CIPHER. Clear a 26 word memory block beginning at FREQ, and determine the frequency of appearance of the ASCII characters in the CIPHER array. Ignore any non-alphabetic characters.

```
label   inst   operand
FREQ    LDY    #0          clear index to clear array
        TYA                and accumulator
F2      STA    FARRAY,Y    clear frequency array cells
        INY                bump index
        CPY    #26         finished?
        BLS    F2          not yet
        CDB    COUNT,-1000 set CIPHER array count
```

```
        CDB   SPTR,CIPHER create scratchpad
                          pointer to CIPHER
        LDY   #0          clear index for indirect
                          reference
F3      LDA   *SPTR,Y     get character from CIPHER
        CMP   #'A'        is it alphabetic?
        BLS   F4          ...
        CMP   #$DB        ...
        BGE   F4          ...
        AND   #$1F        clear high three bits
        TAX               move to index
        INC   FARRAY-1,X  increment FARRAY member
F4      IDB   SPTR        increment scratchpad pointer
        IDB   COUNT       and count
        BNZ   F3          loop until finished
        ...               program continues

COUNT, FARRAY, and SPTR are defined by:

COUNT   $$$               main loop control count
FARRAY  RES   26          frequency array
        SPD   $80
SPTR    $$$               scratchpad pointer
```

Should the reader be interested in verifying the frequency table in example 17-2, if the line:

```
F3    LDA  *SPTR,Y
```

in example 17-3 is replaced by the line:

```
F3    JSR  RKB
```

the text can be typed in directly via the RKB routine given in example 11-2, and the frequency table built.

The reason that the monoalphabetic cipher is broken so easily is that the natural frequency of the letters emerges so quickly, even in a moderate sized

message. In a short message the frequency analysis method can fail if the plaintext reads something like:

> The pooped purple parrot pensively poured peas
> on a poorly painted plastic pepper plate.

or:

> The nattering nabobs of negativism nudged
> Nancy nonchalantly northward.

but this kind of artificial disturbance of the natural frequencies will diminish with the length of the message and in the end the cipher must be broken. Clearly, there is no way that this method can be used for anything more serious than notes passed around a grammar school classroom, as someone should have advised poor Mary.

The problem, clearly, is how to obscure the natural frequencies of the letters of the message. To this end another kind of substitution cipher was devised, which, if properly implemented, is unbreakable. This new frequency obscuring scheme is known as a **polyalphabetic cipher**. A message is enciphered in this system with the aid of a **key**. The plaintext message and the key are laid out side by side, like this:

message: HAPPINESS IS HOWARD COSELL WITH LOCKJAW

key: FPENWKDVW SI NFGOTG MFGSLS THSV PQLNEBJ

The key should be incoherent like the one shown.

The message is enciphered like this. The first character of the plaintext message is H, the eighth letter. The first key character is F, the sixth letter. The eight and the six are added to form fourteen. The fourteenth letter of the alphabet is N and this forms the first ciphertext character. The second plaintext letter is A, the first letter of the

alphabet, and the second key letter is N, the fourteenth. The second ciphertext letter is, therefore, the sum of these, the fifteenth letter, O. The third ciphertext letter is U, the sum of the sixteenth letter P and the fifth letter E.

The fourth ciphertext letter is the sum of P and N, the sixteenth and fourteenth letters. But this sum is thirty, and there are only 26 letters to the alphabet. The character is selected by subtracting 26 from 30 to get 4, and the ciphertext letter is a D, the fourth letter of the alphabet. This process continues until the entire plaintext has been enciphered. This completely obscures the natural frequency and renders frequency analysis useless. It sounds too good to be true, and it is.

The problem with this system is that both sender and receiver must have copies of the key, and a physical copy of the key in the receiver's hands is a dead giveaway if your man in the enemy camp is caught. The key, therefore, must be some common public document which is available to both sender and receiver without any special arrangements. A daily newspaper available in both places will serve nicely. The published shipping tonnages in the daily papers of a seacost city or the stock exchange quotes can be used. Alternately, both sender and receiver can keep copies of a given edition and printing of a common book, Milton's Paradise Lost, for example. Sender and receiver agree on which section of the book is to be used and when, perhaps with a chapter and paragraph number which depend on the date the message is sent.

The use of a commonly available document like a book or newspaper creates the problem that the key is not incoherent, and if your man is caught, the nasties on the other side will regard anything like a book in his posession with great suspicion and immediately go to work to discover the key system. If they employ methods like Sir Francis Walsingham's they are likely to be aided by your man who, after all, is susceptible to pain.

It is tempting here to think of using a key based on a random number generator like the ones described earlier in the book, but remember that these generators produce only a pseudorandom sequence of numbers and if the seed and constant multiplier are ever compromised the entire cipher system falls apart. Remember that what can be done by computer can generally be undone by computer.

A program to perform polyalphabetic encipherment is shown in example 17-4.

EXAMPLE 17-4

Using the polyalphabetic encipherment scheme just described, construct a 6502 program to encipher the 100 character message beginning at PLNTXT, using the key beginning at KEYTXT, leaving the resulting ciphertext in the array beginning at CFRTXT. Assume that all blanks, punctation, control and other superfluous characters have already been edited out of the plaintext and key strings.

```
label   inst   operand
NCFR    LDY    #0          clear index
NCFR1   LDA    PLNTXT,Y    get plaintext character
        AND    #$1F        strip ASCII code bits
        STA    T           save intermediate result
        LDA    KEYTXT,Y    get key character
        AND    #$1F        clear code bits
        ADD    T           add character values
        CMP    #27         greater than 26?
        BLS    NCFR2       no
        SUB    #26         yes, subtract 26
NCFR2   ORA    #$C0        merge with ASCII alphabetic
                           prefix bits
        STA    CFRTXT,Y    save ciphertext character
```

```
        INY             bump index/count
        CPY   #100      end of loop?
        BLS   NCFR1     do it again if necessary
        ...             program continues
```

In view of the simplicity of this process the security it affords is surprising. Remember that security is a relative term. The information only needs to be secure until it would not be of any further use to the opponent. Thus a routine field cipher which will be changed in 12 hours need not be as secure as one which is protecting the plans for the continental defense.

The decipherment of the polyalphabetic cipher is equally simple, being the simple inverse of the encipherment process. Each key character is subtracted from the corresponding ciphertext character. if the result is negative, 26 is added to it. This forms the final deciphered plaintext character. The process is shown in example 17-5.

EXAMPLE 17-5

Create the program required to decipher the text which was enciphered by the program in the previous example.

```
label   inst   operand
DCFR    LDY    #0          clear index/count
DCFR1   LDA    KEYTXT,Y    get key character
        AND    #$1F        clear ASCII bits
        STA    T           save
        LDA    CFRTXT,Y    get ciphertext character
        SUB    T           form difference
        BPL    DCFR2       skip if positive
```

```
        ADD   #26        bias up by 26
DCFR2   ORA   #$C0       merge ASCII bits
        STA   PLNTXT,Y   save plaintext character
        INY              bump index/count
        CPY   #100       end of loop?
        BLS   DCFR1      do it again if required
        ...              program continues
```

We have enciphered and deciphered by the polyalphabetic method, a simple, elegant and secure system.

There are a number of approaches to breaking a polyalphabetic cipher, all described in detail in Kahn's fascinating book. One possible entry point into one of these ciphers is to know the standard form of greeting of the people sending the messages. If the cipher is used by diplomats, the messages might all begin with "Honored Ambassador" or "My Dear Ambassador", or some such salutation.

A broken Japanese cipher was used by the Americans to discover the object of the Japanese MidPacific attack early in 1942. The Japanese, having been so meticulous about so many aspects of their preparations, were curiously careless about communication security. The Americans knew the attack was coming but had no idea of its target, since the target was never mentioned by name in the Japanese radio traffic. Opinion was divided between Midway Island and the Hawaiian Islands. A false message was broadcast in the clear from Midway, therefore, to the effect that the garrison there was experiencing difficulty with its water distillation equipment. Sure enough, a short time later the Japanese radio carried the announcement that the target had reported difficulty with its water distillation equipment, and Midway was confirmed as the target. A broken cipher had been used to elicit information from the opponent, allowing the Americans to prepare the defense of Midway, which proved to be

the turning point of the Pacific war.

There are many other instances in which ciphers have played a pivotal role in historic events, the famous Zimmerman telegram, for example, or the Yamamoto incident. This makes fascinating reading and the interested reader is again referred to Kahn's book. We can go no further in a book which is supposed to be about computer programming.

CHAPTER 18

INTERRUPT DRIVEN PROCESSES

Probably no single area of programming causes so much trouble as interrupts. There are two main reasons for this. The first is that the dimension of time is introduced into the programming process. Time is something that programmers have had to consider little if at all during their previous activities. Until interrupts are considered a program may be thought of as a series of steps, one following the other inexorably, leading to a predetermined result. The only place time is considered is that in which the program must pause waiting for a response from an operator, like poor Lambros in chapter 11.

The second difficulty follows from the first. If a random event can occur causing program execution to be suspended, and perhaps disrupted, there is no way to trace directly the source of the disruption since the exact circumstances under which it happened cannot, in general, be recreated.

Let's try to get over these difficulties by beginning with a definition which will spell out exactly what an interrupt is and what it is not. An interrupt is a suspension of the activity of one program so that another program, whose results are presumed to be more urgent, may be executed. The suspension need not be disruptive, indeed if it is properly handled it will be impossible to determine after the fact the program point at which the suspension occurred. The concept of suspended activity is not as strange as might initially appear. Consider the case of someone in an office, sitting at a desk typing. On the desk are a cup of coffee and a telephone. Every few lines the typist pauses for a sip of coffee, presumably driven by a randomly occurring urge for caffeine. There is no way of telling when such an urge will arise, as all coffee drinkers know. It is very likely, though, that the typist will finish

the current sentence before picking up the cup. There is at least some elasticity in this demand for caffeine which will allow an orderly suspension of the typing before the coffee cup is picked up. Now suppose that in the middle of a sip of coffee the telephone on the desk rings. The ringing telephone represents an immediate demand for attention which occurs at random. There is no way of telling in advance when it will ring. Here too there is elasticity in the demand for attention. It is permitted to put the coffee cup down on the desk before picking up the phone. The demand represented by the ringing phone is not so urgent as to require letting go of the coffee cup in midair to grab the phone. This would result in disruption of the ongoing activity, i.e., coffee would go all over everything, possible ruining the letter being typed and shorting out the electric typewriter, preventing any more work getting done until a mop and a new typewriter had been procured. This illustrates the first necessary characteristic of an interrupt, i.e., that the task suspension must be orderly. In the computer this means that: the instruction in progress is always finished before the interrupt is recognized. This is the simple analog of putting the coffee cup down before grabbing the telephone. The hardware in this case prevents the instruction in progress from being disrupted.

Recall the heavy emphasis laid in the early chapters of this book upon the fact that the program counter always points to the instruction after the one currently being executed. This being the case, and since the interrupt will not be recognized until the current instruction is finished, it follows that when the interrupt is finally recognized, the program counter is pointing to the first word of the instruction which would have been fetched next if the interrupt had not occurred.

In order for any interrupt from any source to occur the interrupt system of the computer must be enabled. In the 6502 this means that the I bit in the

processor status word must be set to zero by means of the CLI instruction, which is supposed to mean clear interrupt disable. CLI and its companion instruction SEI, which is supposed to mean set interrupt disable are two of the most atrocious pieces of nomenclature that this writer has ever seen. They will not be used further here. The assembler will assemble these forms, along with all other defined 6502 mnemonics, but we will use alternate forms here, i.e., EI and DI, meaning enable interrupts and disable interrupts respectively. This is in keeping with the practice in the rest of the computer industry.

The meanings of the terms enable and disable are quite specific. They have nothing to do with any particular device which may be a source of interrupt. The condition of being enabled or disabled is a condition of the 6502 CPU itself.

Individual devices which may be capable of causing interrupts are said to **armed** and **disarmed**. The system may operate with some devices armed and others disarmed, i.e., devices may be armed selectively, allowing some to generate interrupts while others cannot. The arming procedures for individual devices are peculiar to those devices and no general procedure is possible. Again, the armed or disarmed state is a condition of the interrupting device and its interface and has nothing to do with the 6502 CPU state.

In order for an interrupt to occur, three conditions must be met. These are:

1. The interrupt system of the 6502 must be enabled by the EI instruction or other means which will be discussed later in this chapter.

2. The device which is to generate the interrupt must be armed, by whatever procedure is required for the individual device.

3. The interrupting event must occur, setting the

ready flag of the interrupting device to the true state. This event will be different from one device to another. For a real time clock the event will be the expiration of a time delay which causes a pulse to set the ready flag true. For a keyboard the event will be the striking of a key. For a printer it will mean that a character or line has finished printing.

When all three of these conditions are met the interrupt occurs. When the instruction in progress is finished the CPU refrains from fetching the next instruction as it normally would. Instead, it pushes the program counter onto the stack, just as a JSR would. In addition, however, the processor status word is also pushed onto the stack after the program counter just as a PHP instruction would do. When the PSW has been pushed onto the stack, thus saving all of the status bits including I, the I bit of the status word is set to 1. This disables the interrupt system just as if a DI instruction had been executed. An indirect jump is then performed through a pointer located at $FFFE and $FFFF, i.e., control is transferred to the location named in this pointer. The pointer must contain the address of the program which is to respond to the interrupt. This program is known as the **interrupt service routine**. In the Apple II $FFFE and $FFFF contain an address within the ROM monitor. Control is sent to this address, where some processing is done to see whether the interrupt is real or simply the result of a BRK instruction. If the interrupt was real the Apple performs another indirect jump through location $3FE and $3FF. In the Apple, then, $3FE and $3FF should contain a pointer to the interrupt service routine. In other 6502 configurations the procedure by which the interrupt service routine is reached will be different.

The Apple II has no normal interrupt capability in

the hardware. The writer's machine was therefore modified to add this capability. The device arming and disarming sequences shown in this chapter should in consequence be considered as examples only. They are peculiar to the special hardware added to allow interrupt capability. Other than the arming and disarming sequences the examples are valid for any Apple. If the procedure by which the interrupt service routine is reached is also generalized the examples are valid for any 6502 based computer. The principles of interrupt service illustrated are perfectly general and apply to computers in general. With this in mind, we will proceed.

When control reaches the interrupt service routine there are two things which must be done in addition to whatever computational action is implied by the interrupt. First the ultimate source of the interrupt must be removed. This ultimate source, the event which finally, in concert with the enablement of the interrupt system and the arming of the device, caused the interrupt was the transition of the device ready flag to the true state. How this flag is cleared varies from device to device. Particulars will be taken up a little later. If it is not done another interrupt, the second from the same source, will occur immediately when the interrupt system is re-enabled at the end of the interrupt service routine. Second, since the interrupt service routine uses the same registers and PSW as the program which was interrupted, these must be saved and restored before returning to the interrupted program. Failure to do so causes chaos in the interrupted program. In the 6502 the PSW is automatically pushed onto the stack when the interrupt occurs, so the flags are safe, but the programmer MUST, repeat, MUST save and restore any registers used in the interrupt service. Failure to do this is the single most frequent error in interrupt service programming. The consequence of this disaster is that control is returned to the interrupted program with the registers changed. The interrupted program, unaware that the

interrupt has taken place, uses these changed registers as if they were the same as before the interrupt. The result: DISASTER.

The interrupt service routine ends with the execution of an RTI, i.e., return from interrupt instruction. RTI is different from RTS in that it loads the PSW as well as the program counter. Since the interrupt system was, by definition, enabled at the time the interrupt occurred, the PSW which was pushed onto the stack contains an I bit set to enable interrupts. This insures that the interrupt system is automatically reenabled, a positive feature of the 6502. On most other microprocessors the programmer must remember to re-enable, introducing another possible source of error.

Perhaps the simplest type of interrupting device is known as a **real time clock** or **RTC**. A real time clock is really not a clock at all, but a generator of a series of synchronous pulses which generate interrupts. If there is any timekeeping to be done it must be done by the interrupt service routine. A simple application of a real time clock is shown in example 18-1. Please keep in mind that the interrupt clearing and arming functions are peculiar to the writer's system. This example might seem almost insultingly simple, but work through it until you understand it. Better yet, if your machine is capable of interrupts or can be made so, try it out. This may seem tedious, but you have to learn to walk before trying to run.

EXAMPLE 18-1

A teletypewriter is attached to the 6502. Data may sent to it by storing into location $C102. In particular, the TTY bell may be rung by storing $87 into this location. An RTC has been constructed which will interrupt at the line

frequency, i.e., 60 times per second. Construct an RTC interrupt service routine which will keep count of the RTC interrupts and a wait loop which will monitor this count and ring the TTY bell when the count reaches 60, i.e., each second.

```
label   inst    operand
BIGBEN  CLA             clear interrupt count
        STA     TIX     location
        JSR     CLRTC   this clears any waiting
                        interrupt and arms the RTC
                        interrupt
        EI              enable the 6502 interrupt
                        system. Unless this is done
                        no interrupt from the RTC or
                        any other source can occur.
WAIT    LDA     TIX     get interrupt count
        CMP     #60     one second passed yet?
        BLS     WAIT    no
        CLA             yes, first clear the count
        STA     TIX     ...
        LDA     #$87    and then ring this guy's
        STA     TTY     teletype chimes
        JMP     WAIT    and wait for next second
```

The interrupt service routine itself is:

```
TICK    PHA             save the A register
        INC     TIX     bump interrupt count
        JSR     CLRTC   clear the interrupt
        PLA             restore A
        RTI             and return
TIX     ***             interrupt count cell
```

The following routine clears the flag which caused the interrupt and rearms the RTC which was disarmed by the clearing process. Note that this is shown for completeness only. It is peculiar to the writer's system and will not run on the

standard Apple II.

```
CLRTC   CLA             store zero into arming
        STA   ARMLOC    location to clear interrupt
        LDA   #$C       now rearm the RTC by setting
        STA   ARMLOC    bits 2 and 3
        RTS             return
```

The required pointer to the interrupt service routine is created by:

```
        ORG   $3FE
        DBL   TICK
```

The TTY data and RTC arming ports in the writer's system are defined:

```
TTY     EQU   $C102
ARMLOC  EQU   $C0A0
```

Example 18-1 is a kind of "stunt" program. It doesn't really do anything useful, but it does prove that the RTC interrupt works if the bell rings. A real interrupt service routine has something else to contend with. Unlike the situation in example 18-1, the program being interrupted is not a simple wait loop. The program which is interrupted will be doing actual useful work. In fact, it may be doing decimal arithmetic. If the interrupt service routine needs to do binary arithmetic it will be necessary to set the CPU into the binary arithmetic mode with a CLD instruction. The decimal state will, in this case, be restored automatically when the PSW is reloaded by the RTI instruction at the end of the service routine. With this understood we can proceed to do something really useful, construct an interrupt driven software clock which will really keep time. This clock is shown in example 18-2.

EXAMPLE 18-2

A software elapsed time clock is to be constructed which will display elapsed hours, minutes and seconds on the Apple primary text screen screen, which begins at location $400. This display is to be updated every second. The RTC initialization routine is to be written in subroutine form so it can be called by any program resident in memory. The clock should be set to all zero upon initialization, and the binary time kept in cells HOUR, MINUTE and SECOND, accessible to any other resident program.

The initialization for the software clock is:

```
label   inst    operand

TICTOC  CLA             clear count cells
        STA     SECOND  ...
        STA     MINUTE  ...
        STA     HOUR    ...
        STA     TIX     and RTC interrupt counter
        JSR     CLRTC   clear pending interrupt and
                        arm the RTC
        EI              enable the interrupts
        RTS             return to calling program
```

And the interrupt service routine is:

```
TOCK    CLD             don't forget this. The
                        service routine does binary
                        arithmetic and the program
                        that was interrupted may
                        have been using decimal mode
        PHA             save A
        TXA             ...
        PHA             save X
```

```
        INC   TIX      increment interrupt count
        LDA   TIX      fetch count
        CMP   #60      one second gone yet
        BLS   TRTN     no, reload and return
        CLA            yes, clear the
        STA   TIX      interrupt count
        INC   SECOND   increment seconds
        LDA   SECOND   get seconds
        CMP   #60      minute gone yet?
        BLS   CVS      no, convert new seconds
        CLA            yes, clear
        STA   SECOND   seconds count
        INC   MINUTE   and increment minutes
        LDA   MINUTE   get minutes
        CMP   #60      hour gone yet?
        BLS   CVM      no, convert minutes & seconds
        CLA            yes, clear
        STA   MINUTE   minutes count
        INC   HOUR     and increment hours
        LDA   HOUR     get hours
        CMP   #24      day gone yet?
        BLS   CVH      no, convert new hours,
                       minutes and seconds
        CLA            yes, clear hours
        STA   HOUR     ...
CVH     LDA   HOUR     get hours
        JSR   CVD      convert to decimal
        STX   HLOC     store into screen
        STA   HLOC+1   ...
CVM     LDA   MIN      get minutes
        JSR   CVD      convert to decimal
        STX   MLOC     store into screen
        STA   MLOC+1   ...
CVS     LDA   SECOND   get seconds
        JSR   CVD      convert to decimal
        STX   SLOC     store into screen
        STA   SLOC+1   ...
TRTN    JSR   CLRTC    clear interrupt and rearm
        PLA            retrieve old X
        TAX            move it there
```

```
        PLA             restore A
        RTI             return to interrupted
                        program.
```

The binary-decimal converter is:

```
CVD     LDX   #'0'     ASCII zero to X
CVD1    SUB   #10      convert by subtraction
        BMI   CVD2     ...
        INX            ...
        BNZ   CVD1     ...
CVD2    ADD   #$BA     restore units
        RTS            and return
```

The various working locations are defined:

```
HOUR   ***
MINUTE ***
SECOND ***
```

TIX and CLRTC are defined as in example 18-1. The screen locations are defined by:

```
        ORG   $400
HLOC    ASC   '00 '
MLOC    ASC   '00 '
SLOC    ASC   '00 '
```

The pointer to the interrupt service routine is defined by:

```
        ORG   $3FE
        DBL   TOCK
```

It should be noted before leaving example 18-2 that the A and X registers were saved by pushing onto the stack, rather than with STA-STX. There is a reason for this. In some situations it is desirable to re-

enable the interrupt system with an EI instruction before the processing of the current interrupt is complete. This opens the possibility that another interrupt from the same source may arrive before the current one is finished processing. This cannot occur with the simple clock of example 18-2 interrupting only 60 times per second. There are high speed "burst mode" data transfer devices which can do this, however, and if it happens the saved A and X registers from the earlier interrupt are destroyed by being overlaid by the new ones if they are saved in memory rather than on the stack. The result is catastrophe for the interrupted program and no clue whatever to what caused it. Interrupt programming is difficult enough that it is prudent to lock out potential sources of error like this when possible. Save registers on the stack. You don't need this kind of grief.

The clock of example 18-2 interrupted 60 times per second, which raises another interesting problem. Suppose the program which the RTC interrupted was transferring data from disk at the time of the interrupt. The servicing of the interrupt took only a few dozen microseconds, but what happened to the disk data in the meantime? Disk transfer rates are fast enough that it is very likely that data will be lost if an interrupt occurs, so don't expect reliable disk operation in an environment where interrupts from any source can occur.

If a process must be protected from interruption, a problem arises in that simply executing a DI instruction not only disables interrupts but destroys knowledge of the previous state of enablement. If the interrupt system was enabled it is now disabled. If it was disabled it is still disabled. The problem is how to restore the system to its state previous to the disablement. There is a clean solution to this problem. If the PSW is pushed onto the stack before the DI instruction is executed, it can be pulled back into PSW after the section of program to be protected has been finished. Remember, when PLP is executed, <u>all</u>

of the status bits are loaded from the stack. Whatever value is in bit 2 of the top word on the stack becomes the enablement status. This will work in all cases.

Now we must consider a somewhat different interrupt situation, that in which data are transferred under interrupt control. Again, special hardware is required to use this function on the Apple II. The writer's machine has been modified to allow interrupts for both input and output function of an ASR-33 Teletype, and this will be used as the device for which these functions will be illustrated.

There are two principal advantages to the transfer of data under interrupt control. The first of these is the obvious one that the processor is not tied up waiting for a ready flag. It can go about its other essential business, e.g., monitoring reactor core temperature, and when data are ready for transfer, take only the few dozen or so microseconds required to actually transfer the data. It is never necessary to wait for a flag. The second is that this kind of transfer allows the most efficient use of computer resources, i.e., time is never wasted.

We will consider the input and output processes separately here, input first. The terminal input and output interrupts may be armed separately so that input may interrupt while output may not, and vice versa. Let's go back to the case of our duty officer in the reactor room. What we need for him is a scheme for handling his input which won't stop the core temperature monitoring while he's typing. The main temperature monitoring program should be able to execute an initialization sequence for the duty officer's terminal and then go about the business of monitoring without interference from chance disasters like slow typists with long names. When a key is struck only the time required to transfer the character should be taken from the main task. The sequence necessary to do this with the writer's teletype is shown in example 18-3. As with the earlier examples, the arming sequences shown are peculiar to the writer's

system. Other than these and the location of the pointer to the interrupt service routine, which is peculiar to the Apple II, the techniques shown are applicable for any 6502 based computer.

EXAMPLE 18-3

An initialization subroutine and an interrupt service subroutine are to be written which will allow the transfer of a ten character string from the terminal keyboard to a buffer area beginning at location STRING. The initialization sequence should clear a flag. When the tenth character has been read by the service routine this flag should be set to one as a signal to the main operating program that the string is complete.

The initialization subroutine is:

```
label   inst    operand

INI     LDA     TTYIN     this clears the terminal
                          ready flag. See note below
        LDA     #1        arm input interrupt
        STA     ARMLOC    ...
        CLA               clear character index
        STA     INDEX     ...
        STA     IFLAG     clear message ready flag
        EI                enable the interrupt system
        RTS               and return to main operating
                          program which called INI.
```

The interrupt service routine is:

```
ISVCE   CLD               just in case ...
        PHA               save A and Y
        TYA               ...
        PHA               ...
```

```
        LDY   INDEX    get character index
        LDA   TTYIN    this loads the character
                       just typed into A and clears
                       the input ready flag which
                       was the cause of the
                       interrupt.
        STA   STRING,Y save new character in string
        INY            increment index
        CPY   #10      10th character?
        BLS   ISVC2    no
        CLA            yes, clear index for next
        STA   INDEX    message
        INC   IFLAG    set "message ready" flag
ISV2    PLA            restore the registers
        TAY            ...
        PLA            ...
        RTI            and return

The TTY input and arming locations are defined:

ARMLOC  EQU   $C0A0
TTYIN   EQU   $C100
```

This procedure will allow a ten character string to be input through the terminal, but since the input and output functions are separate, the characters typed do not appear on the terminal. This is sometimes desirable in cases where the typed input is to remain secure.

If the typed material is to appear on the terminal display it must be sent there by the interrupt service routine. The best way to do this is to keep the output interrupt disarmed and "echo" each typed character back to the terminal as it is received.

In the first line of the initialization routine INI above there is something not obvious which must be expanded. Recall that clearing the flag which caused the RTC interrupt required special action, the storing

of zero into ARMLOC. No such action was necessary here because the act of reading the data clears the flag. You will find that this is usually the case with terminal devices, but find out precisely how any new one works. Don't assume that the flag is automatically cleared as it is here.

It can be said in general that the ready flag is true when the terminal device is ready to transmit data in either direction. In the input case this means that the flag is false until a key has been struck, meaning that the flag is false most of the time. In the case of output, however, the flag is true whenever no output is being done, i.e., the output ready flag is true whenever the terminal is idle. This means that if interrupts are enabled and the output interrupt armed there will always be an output interrupt waiting.

This results in different arming procedures for input and output. While the input interrupt must be kept always armed, the output interrupt must be kept disarmed until output is actually ready to be done. When the last item is sent out, the output interrupt must be again disarmed to prevent a continuous output interrupt from happening. It is a prudent practice to disable the interrupt system while arming the output interrupt. The transfer of a character string under interrupt control is shown in example 18-4.

EXAMPLE 18-4

A character string has been assembled by the main operating program beginning in location MSG. Write the string onto the TTY under interrupt control, stopping when a carriage return code ($8D) is encountered in the string.

The initialization sequence is:

label inst operand

```
INO     LDA   #2        arm output interrupt. This
        STA   ARMLOC    is peculiar to the writer's
                        system.
        CLA             clear output pointer
        STA   INDEX     ...
        EI              enable interrupts
        RTS             return to main operating
                        program
```

The interrupt service routine is:

```
OSVCE   CLD             set binary mode
        PHA             save A and Y
        TYA             ...
        PHA             ...
        LDY   INDEX     get character pointer
        INC   INDEX     and increment for next pass
        LDA   MSG,Y     get message character
        CMP   #$8D      test for carriage return
        BEQ   OS3       end of message?
        STA   TTYOUT    no, send it to TTY. This
                        also clears the ready flag
                        which caused the interrupt
OS2     PLA             restore the registers
        TAY             ...
        PLA             ...
        RTI             return to interrupted
                        program
OS3     CLA             disarm output interrupt
        STA   ARMLOC    ...
        BZ    OS2       and go to restore registers
```

The ARMLOC definition is the same as before. The TTYOUT location is defined by:

```
TTYOUT EQU   $C102
```

The management of simultaneous input and output along with other interrupts like the RTC becomes somewhat more complex and is beyond the scope of an elementary book like this. The introduction of interrupts allows us for the first time to consider the application of the computer to situations in which there may be computational demand at random from a number of different sources. A system which can meet such demands, responding to each in an appropriate way is known as a **real time** system.

The random and unpredictable character of the inputs to a real time system, and the consequent unpredictable demand on its resources, requires that much attention be paid to keeping an orderly flow of work going through the machine. In order to do this programs have been developed which are capable of overseeing the activities of the component programs of a real time system, in much the same manner in which a human manager or executive oversees the activities of a group of people working under him. A program which can perform this supervisory function is known as a **real time executive**. The principles involved in a real time executive are far beyond the reach of a book like this. Real time executive programs are the topic of a planned later volume in this series.

CHAPTER 19

PROGRAM DEBUGGING

The purpose of this chapter is to introduce the reader to the use of the debugging monitor whose source listing appears in appendix D. A debugging monitor, or debug, provides a number of facilities to aid the programmer in testing and correcting a new program. It is the principal tool used for this purpose, and the time taken to learn how to use it is well invested. The debug program presented here is specific to the Apple II, using many features peculiar to it, though in no case does it depend on any programming in the Apple ROM. For readers who have other 6502 configurations, the source listing in appendix D should provide sufficient information about the necessary techniques to allow the creation of functionally equivalent programs for their computers.

When, at last, you have proceeded to the point that examination of the assembly listings shows no obvious error you are ready to attempt to execute it. With the program in the assembler source buffer, you type:

```
#0.
```

and the assembler responds:

```
OBJECT DEVICE ON, SPACE
```

Prepare the object device and tap the space bar. With the standard Apple II version of the assembler this is a cassette. Make sure it is rewound and press record and play together, and then the keyboard space bar. There will be a pause while the object code of your program is written, followed by a prompt from the assembler. When you see the prompt rewind the tape and transfer control to the debug program by striking the $ key. Before transferring control be sure you have

saved a copy of your source program. New programs often behave in an unpredictable way and have been known to run wild, destroying memory randomly, including the assembler and any text in the source buffer. You disregard this warning at your peril. You do tend to remember to save the source after you have lost it a few times and have to type it in again, especially on long programs. It is a good idea to save source often during the typing of a long program. If a power failure occurs, this way you haven't lost it all.

When the debug program is first entered the top two thirds of the screen are blank and the bottom third looks like this.

```
                    EXAMPLE 19-1

A  X  Y  S  D  NVBDIZC   STACK   BKPT  RTS
00 00 00 FF 00 0000100   1FF>04        0000
                         1FE 44
00 00 00 00 00 0000000   1FD 20
                         1FC F2
00 00 00 00 00 0000000   1FB BE
                         1FA 41
                         1F9 17        SCR
#.                       1F8 FF
```

Don't let this mass of meaningless information frighten you. It will all be explained in due time. At the lower left corner of the screen you will see a period flashing. That is the cursor, just as it was with the assembler. With the rewound cassette containing the object code of your program in the tape machine type:

L.

and the debug will respond with:

OBJECT READY, SPACE

Push the PLAY key on the tape machine and tap the space bar. The word LOAD will appear at the right hand side of the screen. About 7 seconds later the character below the L in LOAD will begin to flash rapidly. This is just a visual indicator that the debug has found the object program on the tape. The flashing will stop for short periods and then resume. When it finally stops the flashing character will go blank and a number will appear below the word LOAD. This is the number of object records that were loaded and needn't concern us now. If the load was not successful the word CKSM appears beneath the word LOAD. This means that a checksum error has occurred and that the material loaded cannot be relied upon. If this happens rewind the tape and try the load again. If it happens the second time the tape is probably bad. Go back to the assembler and generate another object program on a different cassette.

When you get a successful load you will want to verify at least a few program locations in memory. To do this type an M followed by a hexadecimal memory address and a period. All debug commands end with a period. The command line at the bottom of the screen will look something like this:

#M5000.

This is immediately changed to read:

#M5000 B0 0 .

with the period flashing. The B0 is the hexadecimal content of 5000. The 0 to its right is the ASCII value of B0. If you want to change the contents of 5000 to something else, say A0, just type A0. The line now looks like this:

```
#M5000 B0  0  A0.
```

again with the period flashing. If you make a mistake and type A2 there is no damage done. Just type the correct value so the line looks like this:

```
#M5000 B0  0  A2A0.
```

Only the last two typed characters count. This is generally true of debug typeins. If you type in something that is not hexadecimal the entire command line will fill with question marks like this:

```
????????????????????????
```

This will persist for about a second and the prompt character # will reappear, meaning debug is waiting for another command.

Let's return now to our examination of location 5000. We had seen the content of 5000 to be B0 and indicated that it should be changed to A0. Now we must type a terminating character that will enter the change. There are three choices here, depending on what we want to do next. The first is the familiar period. If you type a period here the contents of 5000 will be changed to A0 and the line will disappear. If you had not typed the A0 no change would have been made in memory. The flashing period at the left will reappear, indicating that debug is awaiting another command. If a space is typed the next higher memory location will be examined, the display now looking like this:

```
#M5001 AB  + .
```

If 5001 is to be changed, just type the new contents. If a terminating character is typed without any indicated change the contents of the location are unchanged. The third possibility is to type comma. If comma is used as the terminator the next lower memory

location is examined. You'll get the feel of this best by trying it.

At the lower right corner of the screen you will see the characters SCR, meaning screen. The debug uses the primary text screen but since you may want to look at other Apple screens the debug provides a way to do this. It is done by typing V followed by one of the following combinations, plus period.

T1	Text screen 1
T2	Text screen 2
L1	LORES screen 1
L2	LORES screen 2
L3	Mixed LORES screen 1
L4	Mixed LORES screen 2
H1	HIRES screen 1
H2	HIRES screen 2
H3	Mixed HIRES screen 1
H4	Mixed HIRES screen 2

When this is done the screen will change immediately to the indicated one. To go back to the normal debug screen type VN. The debug screen will reappear, but something will have changed. Look below the SCR at the lower right. This space will contain the identification of the screen you just examined. Something more has happened here than meets the eye, and we will return to this screen business a little later.

If you want to examine a large block of memory and have a parallel printer debug provides a means of dumping memory in blocks to the printer. Type the D command followed by the inclusive dump limits and the block will be printed out. To dump 100 through 1FF inclusive type:

#D100 1FF.

The result looks like this:

EXAMPLE 19-2

Memory Dump to Printer

```
0100   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
0118   FF FF D6 12 B1 A4 1D 7E 1D FF 00 00 FF FF 00 00 FF FF 00 60 BE 57 BE B9
0130   33 BE B9 B7 B4 9B B7 3A B7 69 FD 6F FF FF 00 00 FF FF 00 00 FF FF 00 00
0148   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
0160   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
0178   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
0190   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
01A8   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00
01C0   FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF FF 00 00 FF 60 BE 57
01D8   BE 6A BA 57 60 60 BE 57 60 BE 60 BE 57 BE B9 33 BE B9 B7 B4 92 80 AE 80
01F0   AE 90 B3 36 70 A7 F3 A2 93 A3 7C A1 D2 E3 D6 E2
```

Now that we know something about examining and changing memory, let's get to the business of actually debugging. A debug program allows you to execute small sections of a program at a time to verify the function of these sections. It does this by a process known as **breakpoimting**. Breakpointing is the most powerful single debugging technique you have at your disposal, at least in conventional programming systems. It operates by replacing three memory words of the program being debugged with a JMP instruction whose target is in the debug program. The address at which this substitution is made is known as the **breakpoimt address**. Another address, that of the beginning word of the first program instruction to be executed, is given. This address is known as the **transfer address**. The breakpoint works by replacing the three words at the breakpoint address with a JMP instruction back to the debug and transferring control to the transfer address. Execution of the program being tested is then begun at the transfer address. When the program reaches the breakpoint address control returns to a

special location within debug.

When control reaches debug from the breakpoint the first thing done is to restore the three program words at the breakpoint address to their original values. These three words were saved by debug when the breakpoint was set. The values of the machine registers upon return from the breakpoint are stored and converted for display in the line so labeled on the screen. To execute a section of program which runs from $300 to $32E, for example, we type:

#B300 32E.

If the breakpoint is successful the message:

BKPT REACHED

will be briefly displayed and the top line of registers will show the values of the machine registers when the return from the breakpoint occurred. The former values of the registers, i.e., those resulting from a previous breakpoint, are moved down to the second register display line.

Some other things have happened here too. Under the BKPT on the right side of the screen an address has appeared, the address at which the breakpoint was reached. Each time a breakpoint is successfully executed this display will be moved down one line and the new breakpoint address displayed at the top. A history of the breakpoints is thus created.

You might be wondering at this point what all of this is worth, since the execution of the program began with registers of unknown value. The values of the registers are right in front of you! Before the jump took place to the transfer address the real machine registers were loaded with the values displayed in the top display line. Since these are mostly zeros, let's find out how to set them to the values we want. To set the A register to 3 we type:

#RA3.

and the register display will reflect the change. Now, when the next breakpoint is executed A will be loaded with 3 before the jump to the transfer address. Registers X, Y and S are changed in a similar way. The D in the display is not a register at all. It means difference, and it indicates the arithmetic difference between the stack pointer S before and after the breakpoint.

The procedure for changing the P register is a little different. Here we are interested in the individual bits of the register and debug provides a way to change them individually. Suppose you wanted to force the N and Z bits to ones. Just type:

#RPN1Z1.

The top register display line will change immediately to show the new values. Any or all of the bits may be changed with a single RP command. Note, by the way, that the I bit is set to one initially. This insures that the interrupts are disabled at the jump to the transfer address. If you want them enabled type:

#RPI0.

Let's dispose of the last unexplained item on the screen now, the stack display. The initial value of debug's stack pointer is set to FF. You can change this easily with an RS command followed by the value you want. If you do this the display will change to accomodate. Under the word STACK are displayed the addresses and contents of the highest eight locations in your stack. The section of the stack shown will not change unless you change the stack pointer with RS. When you return from a breakpoint, however, the stack pointer may be different. If this is so the little ">" between the address and contents columns will point to the current value of the stack pointer.

Finally, there is that mysterious looking RTS at the right side of the screen. The number displayed under the RTS is the location to which control would go if the next executed program instruction were an RTS. This is very handy when debugging subroutines which manipulate the stack. If the subroutine makes an error like one too many pushes or pulls, setting a breakpoint just before the subroutine's RTS will let you see where control will go when the RTS is executed. This lets you avoid those situations in which a wrong value is pulled from the stack to the program counter, with the result that the program counter points to someplace in the Arizona desert or Soviet Central Asia. The program then ping-pongs all over the inside of the machine, destroying memory randomly, probably including all evidence of the error that caused the disaster.

A little bit back we promised to return to the subject of the screen specification. The screen that you selected with the V command will also be turned on when any breakpoint is executed so that that screen will be visible during the program execution. This is especially valuable when debugging graphics routines on the Apple.

Now let's try to pull all of this together by applying it to a real debugging situation. The program whose assembly listing is shown on the next page contains a deliberate error which we will find using debug. It is a validity check routine which is supposed to examine the ASCII character in A and set the N flag if the character is period, V if carriage return ($8D), Z if blank, and C if comma. The routine fails to properly detect carriage return. Let's run through some of the valid cases first, just to verify that they work. The A register is first set to the period code, $AE, and a breakpoint executed from the beginning of the VCHK routine, $301, to the RTS at $324. Note that the instruction at the breakpoint address, the RTS, is <u>not</u> executed. Execution goes from the transfer address up to, but not including, the breakpoint address. The A register is set to the

period code by:

#RAAE.

and the screen now looks like example 19-4.

EXAMPLE 19-3

```
PAGE 001          VALIDITY CHECK

002 0300               N       EQU  128
003 0300               V       EQU  32
004 0300               Z       EQU  2
005 0300               C       EQU  1
006 0300 00            T       ***
007 0301 A080          VCHK    LDY  #N
008 0303 C9AE                  CMP  #'.'
009 0305 F014                  BEQ  VRTN
010 0307 A020                  LDY  #V
011 0309 C98D                  CMP  #$8D
012 030B F00E                  BEQ  VRTN
013 030D A002                  LDY  #Z
014 030F C9A0                  CMP  #' '
015 0311 F008                  BEQ  VRTN
016 0313 A001                  LDY  #C
017 0315 C9AC                  CMP  #','
018 0317 F002                  BEQ  VRTN
019 0319 A000                  LDY  #0
020 031B 8D0003        VRTN    STA  T
021 031E 98                    TYA
022 031F 48                    PHA
023 0320 AD0003                LDA  T
024 0323 28                    PLP
025 0324 60                    RTS
026 0325                       END
```

EXAMPLE 19-4

```
A  X  Y  S  D  NVBDIZC   STACK   BKPT  RTS
AE 00 00 FF 00 0000100   1FF>04         0000
                         1FE 44
00 00 00 00 00 0000000   1FD 20         LOAD
                         1FC F2          01
00 00 00 00 00 0000000   1FB BE
                         1FA 41
                         1F9 17         SCR
#.                       1F8 FF
```

The breakpoint is now executed by:

#B301 324.

and the screen now looks like this:

EXAMPLE 19-5

```
A  X  Y  S  D  NVBDIZC   STACK   BKPT  RTS
AE 00 80 FF 00 1010000   1FF>B0 0324  0000
                         1FE 44
AE 00 00 FF 00 0000100   1FD 20         LOAD
                         1FC F2          01
00 00 00 00 00 0000000   1FB BE
                         1FA 41
                         1F9 17         SCR
#.                       1F8 FF
```

The N bit is a one as it should be. Next we try the blank code. A is set to $A0 by:

#RAA0.

and the breakpoint executed again by:

#B301 324.

with this result:

EXAMPLE 19-6

```
A  X  Y  S  D  NVBDIZC  STACK   BKPT  RTS
A0 00 02 FF 00 0010010  1FF>32  0324  0000
                        1FE 44  0324
A0 00 80 FF 00 1010000  1FD 20        LOAD
                        1FC F2         01
AE 00 00 FF 00 0000100  1FB BE
                        1FA 41
                        1F9 17        SCR
#.                      1F8 FF
```

The Z bit is properly set. The reader may verify the remaining case of the comma for himself.

Now let's try the failing case, that of carriage return. A is set to the carriage return code by:

#RA8D.

and same breakpoint executed, with this result:

EXAMPLE 19-7

```
A  X  Y  S  D  NVBDIZC   STACK   BKPT  RTS
8D 00 20 FF 00 0010000   1FF>30  0324  0000
                         1FE 44  0324
8D 00 02 FF 00 0010010   1FD 20  0324  LOAD
                         1FC F2        01
A0 00 80 FF 00 1010000   1FB BE
                         1FA 41
                         1F9 17        SCR
#.                       1F8 FF
```

The misfunction is verified. The overflow bit is still zero when the carriage return code should have caused it to be set to 1. The A register is still set to the carriage return code, so let's try to isolate the error by executing a part of the routine. In particular, it would be interesting to know exactly what is in the Y register when the comparison of A to the carriage return code is done. This comparison is at $309. The section of code to be executed is, therefore, from $301 to $309. This breakpoint is executed by:

#B301 309.

At the return from the breakpoint the screen looks like example 19-8.

EXAMPLE 19-8

```
A  X  Y  S  D  NVBDIZC   STACK   BKPT  RTS
8D 00 20 FF 00 0010000   1FF>30  0309  0000
                         1FE 44  0324
8D 00 20 FF 00 0010000   1FD 20  0324  LOAD
                         1FC F2  0324   01
8D 00 02 FF 00 0010010   1FB BE
                         1FA 41
                         1F9 17        SCR
#.                       1F8 FF
```

The Y register contains the number $20. Referring to the layout of the PSW at the top of page 125, the problem becomes immediately apparent. What is being set to 1 is the unused PSW bit, not overflow. Looking now at the definitions at the top of the program, it can be seen that the line:

```
V       EQU   32
```

is incorrect. It should read:

```
V       EQU   64
```

One way to correct this is to go back to the assembler and change the source. This will have to be done ultimately, of course, but it would be nice to know that this will really fix the problem. What about changing the binary program in memory?

We would like to change the LDY instruction at location $307 to A040 from its present A020. Nothing could be simpler. The offending word is at $308. The M command can be used to change it, thus typing M308. yields:

#M0308 20 @ .

where @ is actually an inverse blank which can't be printed here. The debug waits for you to type in the correct value followed by a period, i.e.:

#M0308 20 @ 40.

This ought to fix the problem. Let's try another breakpoint to see if it did. Typing:

#B301 324.

yields the following result:

EXAMPLE 19-9

```
A  X  Y  S  D  NVBDIZC  STACK  BKPT  RTS
8D 00 40 FF 00 0110000  1FF>70 0324 0000
                        1FE 44 0309
8D 00 20 FF 00 0010000  1FD 20 0324 LOAD
                        1FC F2 0324  01
8D 00 20 FF 00 0010000  1FB BE 0324
                        1FA 41
                        1F9 17       SCR
#.                      1F8 FF
```

This result is correct and we have fixed the problem. Make a note of this in the assembly listing so you don't forget to change the source the next time you assemble. It is the dumbest feeling in the world to have to chase down the same mistake twice.

There is one other possible source of trouble in this program. It will be left to the reader to identify it. It shouldn't be too difficult if you read

chapter 18 carefully.

The kind of error that this program contained is quite often the result of attempting to use decimal to specify a bit or bit pattern, as was done here. Use binary or, better yet, hexadecimal to specify bits and bit patterns. You won't make so many mistakes.

The changing of an instruction in memory is called **patching**. The patch we made here was an easy one, involving only the replacement of one word with another. There are two other cases to be covered, the extra instruction and the missing instruction. If you find an instruction that simply doesn't belong there it is easy to get rid of. Use the M command to replace each word of the instruction with $EA. This is a NOP instruction on the 6502, an instruction which does nothing but occupy space and waste two machine cycles. The function is similar to that of certain Federal bureaucrats.

The missing instruction is much tougher to fix, but the technique is worth learning for all the time it saves. If an instruction is missing and needs to be inserted there is no way to lever the program up to put it in where it belongs, so we have to be a little devious. Find an unused area of memory. At the point where the instruction is to be inserted use the M command to replace three words with the number $4C and the two word address of the first word of the unused memory area. Remember that this address must be specified with the low order word in the low memory address. What you have just done is overlay one, two or three program instructions with a JMP to the location in unused memory. The functions of the instruction which were destroyed must be duplicated as part of the patch. First put in the binary instruction which was to be inserted. Then put in instructions to duplicate the functions of those you overlaid. Finally, the patch must transfer control back to the program being debugged. This jump must go to the first word of the first instruction which was not overlaid by the jump to the patch. You can now proceed with the

remainder of your debugging.

This technique is admittedly complex. Its value lies in the fact that it allows more than one error to be found before going back to source corrections. It takes a bit of practice to get it right but it's worth it.

A few more notes about breakpointing are necessary before going on. Since the breakpoint instruction, a JMP, is three words long there must be space in the program to accomodate it. This means that you can always breakpoint to a three word instruction, but you must be careful about one and two word instructions. If there are data words immediately following an RTS, for example, you cannot breakpoint to the RTS, since the second and third word of the JMP will overlay the data. NEVER attempt to breakpoint to an instruction whose address is modified by the program, since the modification will destroy the breakpoint return instruction.

The debug program allows you to save a little typing with the second and successive breakpoints. If a previous breakpoint has been successfully executed, and if you type only one address after the B command, the debug will assume that you want to start where the previous breakpoint left off and breakpoint to the location named.

If the program runs wild and never reaches the breakpoint, stop it and get control back to the first location of debug. Debug will display the message:

```
BKPT FAILED
```

and restore the program instructions at the failed breakpoint. If the wild program destroyed the debug you are out of luck. Restart things from scratch and breakpoint up to your last successful breakpoint. Then breakpoint small sections until you locate the point where things go out of control.

Some more sophisticated readers may entertain a question about the breakpointing technique used here.

The 6502 has a single word instruction, BRK, which simulates the effect of an interrupt in many ways. The single word BRK could be used instead of the three word JMP, allowing the user greater freedom in the location of his breakpoints. There were two reasons that BRK was not used. The first is that executing BRK obliterates part of the user stack, an unhealthy interaction between the debug and the program being debugged. The second reason is that BRK is not available to many Apple users. In the standard version of the Apple II BRK sends control to the ROM monitor. It was with these things in mind that the JMP approach to breakpointing was decided upon.

A final note on breakpointing before we take up other topics, including that big blank space at the top of the screen. When a breakpoint is executed the debug will switch to whatever screen you last specified with a V command. When control returns to debug the normal debug screen will again be displayed. This is to allow you to watch the results of text or graphic writing operations during the breakpoint period.

There are two other debug functions dealing with memory, the fill and search operations. It is very often convenient to be able to set a block of memory to a value. The F command allows you to do this easily. By typing:

#Fbbbb eeee hh.

the block beginning at bbbb and ending at eeee is filled with the hexadecimal value hh. A block of memory may be searched for some value in a similar way. Type:

#Sbbbb eeee hh.

and debug will search bbbb through eeee inclusive for the value hh, stopping to display each location found to contain hh. Pressing the space bar continues the search. The command ends when eeee has been reached

and tested.

Now let's talk about all that wasted real estate at the top of the Apple screen. It's not really wasted since we can use it in a rather unusual way. It is used for something called **memory windowing**.

The most frequent single operation performed in debugging is the examination of memory. Up until now we have used the M command to do this. This is fine for looking at the contents of scattered memory locations. Very frequently, however, you will be interested in the contents of the same location or locations throughout the debugging process. It then becomes a nuisance to have to type all those M's and numerical addresses after every breakpoint. Memory windowing allows a line on the upper portion of the debug screen to constantly display a location or group of locations automatically. The address need only be typed in once and may be accompanied by a name which is six characters or less. This name will appear in the window chosen beside the memory address it represents. The obvious use of the name is to give the window the same name as that location has in the assembly listing. This saves the constant looking up of addresses in the listing. The contents of each window are updated by debug after every operation in which memory modification can occur.

The debug program maintains 15 windows, numbered 1 through F. A window may be one of five different types. Perhaps the simplest to understand is a hex window so we will begin there. A hex window with the name of ALBERT is specified like this:

```
#W1,H,300,ALBERT.
```

The 1 is the window number. Window 1 is at the top of the screen. The H is the hex specification. The 300 is the beginning location of the window, and ALBERT is the name which is to appear in the window. When this is typed the top line of the screen changes to:

0300 ALBERT 99 B9 00 08 0A 0A 0A 99

This represents the contents of the eight memory locations beginning at $300. ALBERT, presumably, was the label which appeared at this point in the assembly listing. This will be updated after every operation which might cause memory to be modified.

The next window type is ASCII. An ASCII window is specified by typing A as the window type, i.e.:

#W3,A,350,HARVEY.

and the third screen line becomes:

0350 HARVEY ABCDEFGHIJKLMNOPQRSTUVWXYZ12

The block of 28 characters beginning at $350 is displayed as ASCII.

The third window type is single precision decimal. It is specified by S as the window type, i.e.:

#W5,S,FE,LITTLE.

and the fifth line from the top of the screen becomes:

00FE LITTLE 131 -125

The 131 represents the contents of $FE interpreted as an unsigned magnitude. The -125 is the signed value of the number. If the sign bit of the number is zero only the magnitude is displayed.

A window may also display a double precision decimal number beginning at a specified address in the standard double precision format. This is a D type window. It is specified:

#W9,D,3AF7,DOUBLE.

The result, in the ninth screen line is:

3AF7 DOUBLE 65280-00256

The first number is the value of the unsigned double precision magnitude. The second is the value of the double precision signed number.

The final window type is a pointer window, specified by P. The command:

#WB,P,F0,PTR.

results in the following in the eleventh screen line:

00F0 PTR 01FF >E2 ,X>00 ,Y>D5

The 00F0 is the location of the pointer PTR. The 01FF is the contents of the pointer at F0 and F1. The E2 is the contents of the cell to which PTR is pointing, the number following ",X>" is the content of the location whose address would be given by adding the debug X register to the pointer, and the number which follows the ",Y>" is the content of the location whose address is the sum of debug's Y register and the pointer.

Any memory window may be cleared, i.e., set to blank by using the letter C in plae of the window type, e.g., window 7 is cleared by:

#W7,C.

This system of memory windows provides a very convenient and efficient means of keeping track of what is happening in memory during the program's trial execution under debug's control.

This chapter can only cover the rudiments of the debugging process. There are those who claim that debugging is more of a black art than a set of techniques. Whatever it is, you get much better at it with practice. It can be a very, very frustrating task. When a program error eludes you for a long time it is very tempting to begin to blame the hardware. This is most probably not the case. This writer has

been programming now for twenty years. In this twenty years he has had only one error which he could directly trace back to machine malfunction. It may prop up the ego a bit to blame the computer, but it's almost never true.

When reading a program listing one tends to see in that listing what he wants to see, what he intended should be there in the first place. It is easy to read over and over the same error without seeing it, including the dumb ones like writing LDA instead of LDX. In cases like this there are two possible remedies, both involving a fresh pair of eyes. The first is to go to a movie or take a walk. Forget the problem and come back to it when your anger and frustration have died down. The second is harder on the ego, since it involves admitting that you need help. Let somebody else look at the listing. Very often a bonehead error will leap right up off the page for somebody seeing the program for the first time.

Take courage. It does get easier with time and practice but however long you work at this craft you will still make errors, some of them awfully stupid. If the number of copies of this book sold equals or exceeds the number of programming errors that the writer has made in twenty years it will be very lucrative.

GOOD LUCK!!!

CHAPTER 20

STRUCTURE OF THE ASSEMBLY PROGRAM

It is a fair bet that anyone who has purchased a book like this, not to mention having read this far into it, is probably serious about the business of programming. At some stage in your programming career, perhaps not now but later, you will begin to be curious about how the programming vehicle itself works. That is what this chapter is about, how the assembler works. It is also the reason that the full source of the system has been published in appendix A. Only with the source available can you actually study how the system works in any but a theoretical way.

This assembler was written especially for this book. Its purpose is to serve as a learning vehicle for prospective 6502 assembly language programmers. It is not intended to serve as a development system for very large scale software efforts. Where there is no conflict between the needs of development and the primary educational purpose of the system all reasonable efforts have been expended to accomodate those needs. The reason for the existence of the system is educational, however, and this should be kept in mind by those who might apply it to other areas.

The key element of the system is simplicity. The methods used in creating the assembler are the simplest which would serve the purpose. It was intended that at some stage the reader of the book would be able to read and understand the assembly listing of the assembler. Speed and efficiency have both been sacrificed to simplicity of function. The system is not particularly fast, assembling about 2,000 lines per minute. It is not particularly compact, occupying about 8K before the I/O drivers are added. To have emphasized either speed or compactness would have meant sacrificing simplicity.

The text editor has the minimum command set required for line by line editing. It is not a general purpose editor, but one which is explicitly intended

for editing assembly language text on a line by line basis. Those wishing to augment the function of either the text editor or assembler have the means to do so at their disposal with the source listings in the appendices.

The assembly language itself is an extension of that found in the machine manual. The extensions have been made with an eye to education. There is no learning value whatever in having to write:

```
CLC
ADC  ***
```

or

```
LDA  #<SYMBOL
STA  PTR
LDA  #>SYMBOL
STA  PTR+1
```

over and over again, since the concepts of addition and pointer creation are nuclear to assembly language programming. Constructions like:

```
LDA  (PTR,X)
```

and

```
LDA  (PTR),Y
```

serve no cause other than confusion. The pre or post index condition is implicit in the X or Y, and the indirection is decided by the first parenthesis. A scheme like this may have some value in formal language design notation, but it requires the users of the system to prove to the inanimate programming system that they know the difference between the pre and post condition every time they write an indirectly addressed instruction, ad infinitum, an enormous unreasonable demand that this writer personally finds maddening. It

is partly this approach to programming language design that has given the 6502 the reputation of being difficult to program.

An assembler performs its translation task by scanning the program to be assembled at least twice. It is important to understand that the source program is examined one line at a time. Look at example 11-4 on pages 174-175. In particular, consider the fourth line of the example, i.e.:

```
STA  T+1
```

While the binary value of the STA instruction can be looked up in a table, the value of the symbol T is unknown, since the definition of T occurs at the end of the program. Further, the binary instruction cannot be fixed at this point because the assembler does not yet know whether the symbol T corresponds to a scratchpad or general memory location. It seems that we will need to know the values of all the symbols first. Since the 6502 generates a two word instruction for a scratchpad reference and a three word instruction for a general memory reference, we will also need to know whether each symbol used represents a scratchpad or general memory location. It is the need for this prior knowledge of symbol values which shapes our attack on the problem. In particular, the need to treat scratchpad symbols differently requires that we make an extra pass over the source program to define them separately. The alternative to this is to require that the user of the system define all scratchpad references at the beginning of the program. Which of these approaches is taken is a matter of choice. It is the writer's opinion that all possible clerical tasks be done by the programming system, and that imposing the extra burden of pre-definition of scratchpad symbols on the programmer is unacceptable.

The assembly of 6502 programs must, therefore, be composed of three logical subtasks, each accomplished in a separate scan of the source program. The subtasks

are:

1) Determine the values of all scratchpad symbols.

2) Determine the values of all general memory symbols.

3) Using the table of symbol values defined in the first two scans, generate the binary value of each source program line.

In the first pass over the source program the values of the scratchpad symbols must be determined. Since scratchpad symbols are only defined by inclusion within an SPD-ESP pseudo-op pair, this first pass ignores all lines which are not so included. A special counter, known as a **location counter** is initialized to zero. The program is scanned until an SPD is found. The value of the SPD's operand field is determined and placed in the location counter. The data defining pseudo-ops which follow may or may not have labels. Consider the following case:

```
          SPD   $80
M         DATA  'A',-1
BLOCK     RES   3
SAM       $$$
          ESP
```

The value of the symbol M is $80. At this point we are not interested in the individual items in the operand field of the DATA, only their number. This number is one greater than the number of commas in the field which are outside apostrophes. This is to allow the case in which the comma occurs in an ASCII context. The number of data items is 2 above, so 2 is added to the location counter. Its value is now $82. The next symbol, BLOCK, assumes this value. The operand field of the RES indicates a length of three, which is added to the location counter, whose value now becomes $85. The next symbol, SAM, thus assumes the value $85. The

scan terminates when the ESP is encountered, and the assembler scans forward until the next SPD is found, or END is encountered.

As each scratchpad symbol is defined it is entered into a table of symbols along with its value. This table is printed in alphabetical order at the end of the assembly listing. It is the usual practice to sort the symbol table just prior to the final assembly pass. In this system the symbol table is built alphabetically so the sort is omitted. Scratchpad symbols are specially marked in the symbol table so that when they are subsequently looked up their scratchpad identity can be determined.

On the second pass over the source program all lines which are outside SPD-ESP pairs are scrutinized. The location counter is now set, not to zero, but to $300 as a default value. This is in recognition of the special character of the memory from 0 through $1FF. The Apple II also uses the area beginning at $200 for special purposes. If no ORG is given the program will thus be assembled beginning at $300. If an ORG is encountered the value in its operand field replaces the current location counter. Note that anything which appears in the field of an ORG or SPD must be numeric or have been previously defined if it is not numeric. An undefined operand in either of these cases makes it impossible to continue with the assembly and the terminal pass 1 error message is generated. This is also true of EQU and RES. To understand this consider this situation:

```
AREA   RES  $3G
```

The $3G is nonsense, so the length of the block being defined by the RES is unknown. This means that the assembler has no value to add to the current location counter, i.e., the value of any symbol which follows the bad RES cannot be known. In the case of EQU consider this:

```
IOPORT EQU  $C0BL
```

The operand is nonsense and the value of IOPORT is unknown. A subsequent instruction like:

```
PRINT  STA  IOPORT
```

has an indeterminate length, since it is not known whether IOPORT is a scratchpad symbol or not.

To see what goes on during the second pass, consider the following program:

```
        ORG   $1D00
START   LDA   #5
        STA   LOC
INIT    LDY   N1
        STY   D
        RTS
LOC     ***
        SPD   $A0
D       ***
        END
```

The ORG is processed and the location counter set to $1D00. This is its value at the time the START label is encountered and, therefore, becomes the value of START in the symbol table. The LDA operation is looked up in the operation code table and, based on a type code contained in that table along with the binary skeleton of the LDA operation, is found to be of length 2 or 3, depending on the addressing mode. The presence of the "#" in the operand sets the length at 2, the length for all immediate mode instructions. The 2 is added to the location counter and the next instruction, STA LOC is examined. There is no label to be processed so no symbol table entry is made. The STA may also be of length 2 or 3, length 2 being the scratchpad addressing case. The symbol LOC is not in the symbol table, not yet having been processed, so it cannot be a scratchpad symbol. The length is,

therefore, 3. The 3 is added to the location counter, making its value $1D05 when the INIT label is encountered. INIT is assigned the value $1D05. LDY has length 2 or 3. Since the symbol N1 cannot be found in the symbol table it cannot represent a scratchpad value. N1 is, of course, undefined, but there is no way to know that at this point. This will be caught during the final assembly pass. The 3 is added to the location counter, whose value becomes $1D08. The STY instruction is now processed. It too can have length 2 or 3. The length is fixed at 2 because the operand of the STY, D, is found to be a scratchpad symbol, already defined during the previous pass. The 2 is added to the location counter to make its value $1D0A. The RTS is now processed. This instruction takes no operand and is of fixed length 1 in all circumstances. The location counter now becomes $1D0B. Since the location counter has this value when the label LOC is encountered, $1D0B is entered into the symbol table as the value of LOC. The pseudo operation *** is found from its type to require no operand and be of fixed length 1. The 1 is added to the location counter to make its value $1D0C.

Since the next line contains an SPD it is ignored as the assembler scans forward looking for an ESP or END. The END is encountered and the symbol defining passes are finished.

In the third and final pass over the source program each line is examined again. Whether the assembly is made as an error scan (#E.), an object generation assembly (#O.) or a full assembly with assembly listing (#Ln m.), the procedure in the third pass is fundamentally the same. Each reformatted source line is copied into a buffer since it may be necessary to print the line if an error occurs. The line number is converted to decimal and placed at the left end of the print line.

The location counter, set by default to $300, is immediately changed by the ORG to $1D00. In the second line the symbol START is looked up in the symbol table

to see if it was defined more than once. If it was an M error flag is entered against this line. The LDA is now looked up and its basic binary skeleton found. This skeleton is modified to reflect the addressing mode used, immediate, and the binary word so generated entered into an object buffer. At this point the operand itself must be examined. If the operand to the right of the "#" is a legal immediate operand its value is entered as the second word of the object buffer. If it is not legal a U error is entered against the line. For each line the length it represents is computed and added to the location counter, just as was done in pass 2. After each line is so processed control is transferred to a program section which converts the location counter to hexadecimal and places it just to the right of the decimal line number in the print area. The object words generated by this line are also converted and placed in the print area. If the assembly is an error scan the error flags are examined and the line printed if any error was generated. The next line is then processed. During an object generation assembly the object code is sent to an object buffer where it is held until 127 or more words have been generated, then written to the external object medium. Any generated errors are also printed. The next line is then processed. In a listing assembly the line is printed unconditionally, along with any error flags associated with it. The only exception to this is when the listing limits have been specified as something other than 1 F. In this case the listing limits control the printing.

When all lines have been processed in a listing assembly the symbol table is printed and the assembly is finished.

The detailed working of the assembly program is, of course, a good deal more complex than this simple synopsis would indicate.

The text editor is a very simple program in concept, if not in actual form. Text is typed in at the system keyboard and added to a memory buffer which

begins at the first word available after the assembler input/output package. The assembler begins the symbol table at the first word above the source buffer.

Lines are added by finding the place in the source buffer where they are to be inserted and moving the entire buffer above this point up by the necessary amount, adjusting the line count.

Lines are deleted by moving the entire source buffer above the deleted lines down by the amount of space generated by the deletion. There are no line numbers kept internally. These are generated as needed for listing and search purposes by counting carriage return characters beginning at the source buffer origin.

Text lines are kept in memory exactly as they were typed in, without superfluous blanks. Source text written to an external medium like cassette is also in this form. The automatic formatting observed on the screen and print devices is done at the time the line is extracted from the source buffer for processing or listing.

APPENDIX A

ASSEMBLY LISTING OF THE ASSEMBLY PROGRAM

```
PAGE 01    6502 RESIDENT ASSEMBLER 5/5/80

002                     ***************************
003                     *                         *
004                     * RESIDENT ASSEMBLER FOR  *
005                     * 6502 BASED COMPUTERS    *
006                     *                         *
007                     * COPYRIGHT W.J. WELLER   *
008                     * DECEMBER 1979           *
009                     *                         *
010                     ***************************
011                     *
012                             ORG   $2000
013                     *
014 2000 A2FF           MON     LDX   #$FF
015 2002 9A                     TXS
016 2003 D8                     CLD                 JUST IN CASE...
017 2004 209B20                 JSR   INITLZ
018 2007 AD1024                 LDA   CLFLAG
019 200A F019                   BZ    MON1
020 200C ADC520         CLEAR   LDA   BFR
021 200F 8D0E24                 STA   BFA
022 2012 857A                   STA   NXTCHR
023 2014 ADC620                 LDA   BFR+1
024 2017 8D0F24                 STA   BFA+1
025 201A 857B                   STA   NXTCHR+1
026 201C A900                   LDA   #0
027 201E 8578                   STA   LNCNT
028 2020 8579                   STA   LNCNT+1
029 2022 8D1024                 STA   CLFLAG
030 2025 A2FF           MON1    LDX   #$FF
031 2027 9A                     TXS
032 2028 4C0C26                 JMP   CSPACE
033 202B A9258560       MON1A   CDB   MONPTR,MON1
034 2033 A94F8562               CDB   ERPTR,ERR
035 203B A999           MON2    LDA   #<PROMPT
036 203D A220                   LDX   #>PROMPT
037 203F 20D420                 JSR   DISP
038 2042 20EA20                 JSR   RKB
039 2045 A00D                   LDY   #JTABLE-CTABLE-1
040 2047 D96F20         MON3    CMP   CTABLE,Y
041 204A F00B                   BEQ   MON4
042 204C 88                     DEY
043 204D 10F8                   BPL   MON3
044 204F A9BF           ERR     LDA   #'?'
045 2051 20A120                 JSR   OUT
046 2054 4C3B20                 JMP   MON2
047 2057 98             MON4    TYA
048 2058 0A                     ASLA
049 2059 A8                     TAY
050 205A B97E20                 LDA   JTABLE+1,Y
051 205D 48                     PHA
052 205E B97D20                 LDA   JTABLE,Y
053 2061 48                     PHA
054 2062 60                     RTS
055 2063                        RES   12,0
056                     *
057 206F AFCCC4C1       CTABLE  ASC   '/LDARW#MU'
058 2078                        RES   5,0
059 207D C8202424       JTABLE  DBL   CL1-1,LIST-1,DELETE-1
060 2083 95221F26               DBL   ADD-1,READ-1,WRITE-1,ASMBL-1
```

```
061 208B 5E24B220           DBL   MEMORY-1,CNCT-1
062 208F                    RES   10,0
063                   *
064 2099 BE8D         PROMPT DATA '>',$8D
065                   *
066                   ******************
067                   * I/O JUMP TABLE *
068                   ******************
069                                             INITIALIZE ALL DEVICES
070 209B 60           INITLZ RTS
071 209C 0000                DBL   0
072                                             ADVANCE TERMINAL LINE
073 209E 60           TCRLF  RTS
074 209F 0000                DBL   0
075                                             OUTPUT CHAR. IN A TO TERMINA
076 20A1 60           OUT    RTS
077 20A2 0000                DBL   0
078                                             FETCH CHAR. FROM TERMINAL KE
079 20A4 60           FKCH   RTS
080 20A5 0000                DBL   0
081                                             SKIP HEADER ON INPUT MEDIUM
082 20A7 60           RHDR   RTS
083 20A8 0000                DBL   0
084                                             READ ONE BYTE FROM INPUT MED
085 20AA 60           RDBYTE RTS
086 20AB 0000                DBL   0
087                                             WRITE RECORD TO OUTPUT MEDIU
088 20AD 60           WRTAPE RTS
089 20AE 0000                DBL   0
090
091                                             PRINT LINE ON SYSTEM PRINTER
092 20B0 60           PRT    RTS
093 20B1 0000                DBL   0
094                                             CONNECT LOGICAL I/O
095 20B3 60           CNCT   RTS
096 20B4 0000                DBL   0
097                                             DETECT CONTROL-C
098 20B6 60           DTCC   RTS
099 20B7 0000                DBL   0
100                                             WAIT FOR SPACE ON TERMINAL
101 20B9 60           SB     RTS
102 20BA 0000                DBL   0
103                                             WRITE OBJECT LEADER
104 20BC 60           WOBL   RTS
105 20BD 0000                DBL   0
106                                             WRITE OBJECT RECORD
107 20BF 60           WOBJR  RTS
108 20C0 0000                DBL   0
109                                             WRITE OBJECT TRAILER
110 20C2 60           WOBTR  RTS
111 20C3 0000                DBL   0
112
113 20C5 0000         BFR    DBL   0          SOURCE BUFFER ADDRESS ***
114 20C7 FFBF         EOB    DBL   $BFFF      END OF SOURCE BUFFER
115                   *
116                   *
117 20C9 20EA20       CL1    JSR   RKB        WAIT FOR PERIOD
118 20CC 1003                BPL   CLERR
119 20CE 4C0C20              JMP   CLEAR
```

```
120 20D1 4C4F20      CLERR   JMP   ERR
121                  *
122                  * DISPLAY LINE WHOSE BEGINNING ADDRESS IS IN
123                  * A&X ON ENTRY ON TERMINAL. LOW ADDRESS IS
124                  * EXPECTED IN A, HIGH IN X. DISPLAY TERMINATES
125                  * WHEN CARRIAGE RETURN IS FOUND IN OUTPUT
126                  * STRING ($8D).
127                  *
128 20D4 8574        DISP    STA   UPTR8      SAVE ADDRESS
129 20D6 8675                STX   UPTR8+1
130 20D8 209E20              JSR   TCRLF      ADVANCE TERMINAL LINE
131 20DB A000                LDY   #0         CLEAR INDEX
132 20DD B174        DSP1    LDA   *UPTR8,Y
133 20DF C98D                CMP   #$8D       TERMINATOR?
134 20E1 F006                BEQ   DSPRTN
135 20E3 20A120              JSR   OUT        NO, DISPLAY IT
136 20E6 C8                  INY
137 20E7 D0F4                BNZ   DSP1
138 20E9 60          DSPRTN  RTS              RETURN
139                  *
140                  * READ CHAR FROM KEYBOARD, RETURN WITH FLAGS:
141                  *
142                  * BCS - LEGAL DECIMAL
143                  * BZ  - BLANK
144                  * BMI - PERIOD
145                  * BVS - CARRIAGE RETURN
146                  *
147 20EA 8E2921      RKB     STX   RKBX
148 20ED 8C2A21              STY   RKBY
149 20F0 20A420              JSR   FKCH
150 20F3 0980                ORA   #$80
151 20F5 8D2821              STA   RKBT
152 20F8 A000        RKB2    LDY   #0
153 20FA C9B0                CMP   #'0'
154 20FC 9007                BLS   RKB3
155 20FE C9BA                CMP   #$BA
156 2100 B003                BGE   RKB3
157 2102 C8                  INY
158 2103 D016                BNZ   RKBRTN
159 2105 C9A0        RKB3    CMP   #' '
160 2107 D004                BNE   RKB4
161 2109 A002                LDY   #2
162 210B D00E                BNZ   RKBRTN
163 210D C9AE        RKB4    CMP   #'.'
164 210F D004                BNE   RKB5
165 2111 A080                LDY   #$80
166 2113 D006                BNZ   RKBRTN
167 2115 C98D        RKB5    CMP   #$8D
168 2117 D002                BNE   RKBRTN
169 2119 A040                LDY   #$40
170 211B 98          RKBRTN  TYA
171 211C 48                  PHA
172 211D AD2821              LDA   RKBT
173 2120 AE2921              LDX   RKBX
174 2123 AC2A21              LDY   RKBY
175 2126 28                  PLP
176 2127 60                  RTS
177 2128 00          RKBT    ***
178 2129 00          RKBX    ***
```

```
179 212A 00          RKBY    ***
180                  *
181                  * GET HEX CHAR FROM KEYBOARD, SET FLAGS ON RETURN:
182                  *
183                  * BCS - LEGAL HEX
184                  * BZ  - BLANK
185                  * BMI - PERIOD
186                  * BVS - CARRIAGE RETURN
187                  *
188 212B 20EA20      FHEX    JSR   RKB
189 212E 08                  PHP
190 212F B00B                BCS   FHEX2
191 2131 C9C1                CMP   #'A'
192 2133 900C                BLS   FHEX3
193 2135 C9C7                CMP   #'G'
194 2137 B008                BGE   FHEX3
195 2139 38E907              SUB   #7
196 213C 290F        FHEX2   AND   #$F
197 213E 28                  PLP
198 213F 38                  SEC
199 2140 60                  RTS
200 2141 28          FHEX3   PLP
201 2142 18                  CLC
202 2143 60                  RTS
203                  *
204                  * GET DECIMAL PARAMETER FROM KEYBOARD, SET FLAGS
205                  *
206                  * BCS - AT LEAST ONE DIGIT TYPED
207                  * BZ  - TERM. WAS A BLANK
208                  * BMI - TERM. WAS PERIOD
209                  * BVS - TERM WAS C.R.
210                  *
211                  *
212 2144 A900        IDC     LDA   #0
213 2146 857C                STA   UT1
214 2148 857D                STA   UT1+1
215 214A 857E                STA   UT2
216 214C 20EA20      IDC1    JSR   RKB
217 214F B020                BCS   IDC2
218 2151 08                  PHP
219 2152 3055                BMI   EOIDC
220 2154 F053                BZ    EOIDC
221 2156 C9C6                CMP   #'F'
222 2158 D014                BNE   JM2
223 215A 68                  PLA
224 215B 20EA20              JSR   RKB         GET TERMINATOR
225 215E 100E                BPL   JM2         TERMINATOR AFTER F MUST BE .
226 2160 08                  PHP
227 2161 857E                STA   UT2
228 2163 A578                LDA   LNCNT
229 2165 857C                STA   UT1
230 2167 A579                LDA   LNCNT+1
231 2169 857D                STA   UT1+1
232 216B 4CA921              JMP   EOIDC
233 216E 4C4F20      JM2     JMP   ERR
234 2171 290F        IDC2    AND   #$F
235 2173 48                  PHA               SAVE NEW DIGIT, MULTIPLY
236 2174 067C                ASL   UT1         PREVIOUS RESULT BY 10
237 2176 267D                ROL   UT1+1
```

```
238 2178 B0F4              BCS   JM2
239 217A A57D              LDA   UT1+1
240 217C 48                PHA
241 217D A57C              LDA   UT1
242 217F 067C              ASL   UT1
243 2181 267D              ROL   UT1+1
244 2183 B0E9              BCS   JM2        CHECK FOR O'FLOW
245 2185 067C              ASL   UT1
246 2187 267D              ROL   UT1+1
247 2189 B0E3              BCS   JM2
248 218B 18657C            ADD   UT1
249 218E 857C              STA   UT1
250 2190 68                PLA
251 2191 657D              ADC   UT1+1
252 2193 B0D9              BCS   JM2
253 2195 857D              STA   UT1+1
254 2197 68                PLA
255                                         ADD NEW DIGIT
256 2198 18657C            ADD   UT1
257 219B 857C              STA   UT1
258 219D A57D              LDA   UT1+1
259 219F 6900              ADC   #0
260 21A1 B0CB              BCS   JM2
261 21A3 857D              STA   UT1+1
262 21A5 E67E              INC   UT2
263 21A7 90A3              BCC   IDC1
264 21A9 68          EOIDC PLA               SET FLAGS
265 21AA 29FE              AND   #$FE
266 21AC A67E              LDX   UT2
267 21AE F002              BZ    EOIDC1
268 21B0 0901              ORA   #1
269 21B2 48          EOIDC1 PHA
270 21B3 A57C              LDA   UT1
271 21B5 A67D              LDX   UT1+1
272 21B7 28                PLP
273 21B8 60                RTS
274                  *
275                  * GET ONE OR TWO DECIMAL NUMBERS FROM
276                  * KEYBOARD, SECOND # > FIRST #. COMPUTE
277                  * THEIR INCLUSIVE NEGATIVE DIFFERENCE,
278                  * LEAVE IN IDIFF,+1. ADDRESSES OF LINES
279                  * CORRESPONDING TO THESE NUMBERS ARE
280                  * LEFT IN BADR1, EADR1 AND BADR2, EADR2.
281                  * BADR IS ADDRESS OF BEGINNING OF LINE,
282                  * EADR IS ADDRESS OF TERMINATING C.R.
283                  *
284 21B9 204421     FPRM JSR   IDC
285 21BC 9015              BCC   BCCFPE
286 21BE 08                PHP
287 21BF 867E              STX   UT2
288 21C1 48                PHA
289 21C2 057E              ORA   UT2
290 21C4 F05B              BZ    FPERR
291 21C6 68                PLA
292 21C7 8591              STA   FIRST
293 21C9 8692              STX   FIRST+1
294 21CB 28                PLP
295 21CC 3009              BMI   FPRM2
296 21CE D051              BNZ   FPERR
```

```
297 21D0 204421          JSR   IDC
298 21D3 904C     BCCFPE BCC   FPERR
299 21D5 104A            BPL   FPERR
300 21D7 8593     FPRM2  STA   SECOND
301 21D9 8694            STX   SECOND+1
302 21DB 48              PHA             FORM ONE'S COMPLEMENT
303 21DC 8A              TXA
304 21DD 49FF            CMA
305 21DF AA              TAX
306 21E0 68              PLA
307 21E1 49FF            CMA
308 21E3 186591          ADD   FIRST
309 21E6 859D            STA   IDIFF
310 21E8 8A              TXA
311 21E9 6592            ADC   FIRST+1
312 21EB 859E            STA   IDIFF+1
313 21ED 1032            BPL   FPERR
314 21EF A578            LDA   LNCNT
315 21F1 38E593          SUB   SECOND
316 21F4 A579            LDA   LNCNT+1
317 21F6 E594            SBC   SECOND+1
318 21F8 3027            BMI   FPERR
319 21FA A591            LDA   FIRST
320 21FC A692            LDX   FIRST+1
321 21FE 203322          JSR   FBA
322 2201 8595            STA   BADR1
323 2203 8696            STX   BADR1+1
324 2205 A566            LDA   UPTR1
325 2207 8597            STA   EADR1
326 2209 A567            LDA   UPTR1+1
327 220B 8598            STA   EADR1+1
328 220D A593            LDA   SECOND
329 220F A694            LDX   SECOND+1
330 2211 203322          JSR   FBA
331 2214 8599            STA   BADR2
332 2216 869A            STX   BADR2+1
333 2218 A566            LDA   UPTR1
334 221A 859B            STA   EADR2
335 221C A567            LDA   UPTR1+1
336 221E 859C            STA   EADR2+1
337 2220 60              RTS
338 2221 4C4F20   FPERR  JMP   ERR
339                      *
340                      * TWO'S COMPLEMENT DOUBLE PRECISION NUMBER
341                      * LOW ORDER IN A, HIGH ORDER IN X. RETURN
342                      * WITH TRUE ZERO FLAG
343                      *
344 2224 48       TCD    PHA
345 2225 8A              TXA
346 2226 49FF            CMA
347 2228 AA              TAX
348 2229 68              PLA
349 222A 49FF1869        TCA
350 222F D001            BNZ   TCDR
351 2231 E8              INX
352 2232 60       TCDR   RTS
353                      *
354                      * GET ADDRESS OF LINE WHOSE NUMBER
355                      * IS IN A & X ON ENTRY, LOW ORDER IN A.
```

```
                        * BEGINNING ADDRESS IN A-X, ADDRESS OF
                        * TERMINATING C.R. IN UPTR1, UPTR1+1
                        *
2233 202422     FBA     JSR   TCD
2236 F048               BZ    FBA7
2238 8568               STA   UPTR2
223A 8669               STX   UPTR2+1
223C ADC520             LDA   BFR
223F 8566               STA   UPTR1
2241 ADC620             LDA   BFR+1
2244 8567               STA   UPTR1+1
2246 E668D002           IDB   UPTR2
224C F018               BZ    FBA4
224E A000               LDY   #0
2250 B166       FBA2    LDA   *UPTR1,Y
2252 E666D002           IDB   UPTR1
2258 C98D               CMP   #$8D
225A F002               BEQ   FBA3
225C D0F2               BNZ   FBA2
225E E668D002   FBA3    IDB   UPTR2
2264 D0EA               BNZ   FBA2
2266 A566       FBA4    LDA   UPTR1
2268 48                 PHA
2269 A567               LDA   UPTR1+1
226B 48                 PHA
226C A000       FBA5    LDY   #0
226E B166               LDA   *UPTR1,Y
2270 C98D               CMP   #$8D
2272 F008               BEQ   FBA6
2274 E666D002           IDB   UPTR1
227A D0F0               BNZ   FBA5
227C 68         FBA6    PLA
227D AA                 TAX
227E 68                 PLA
227F 60                 RTS
2280 ADC520     FBA7    LDA   BFR
2283 38E901             SUB   #1
2286 8566               STA   UPTR1
2288 ADC620             LDA   BFR+1
228B E900               SBC   #0
228D 8567               STA   UPTR1+1
228F ADC520             LDA   BFR
2292 AEC620             LDX   BFR+1
2295 60                 RTS
                        *
                        * ADD FUNCTION
                        *
2296 A900       ADD     LDA   #0
2298 8580               STA   UT3
229A 204421             JSR   IDC       GET LINE #
229D 106C               BPL   ADERR
229F 8591               STA   FIRST
22A1 8692               STX   FIRST+1
22A3 38E578             SUB   LNCNT     CHECK LEGAL LINE #
22A6 857E               STA   UT2
22A8 8A                 TXA
22A9 E579               SBC   LNCNT+1
22AB 3006               BMI   ADD0
22AD 057E               ORA   UT2       LINE # = LINE COUNT?
```

```
415 22AF D05A               BNZ   ADERR
416 22B1 E680               INC   UT3         YES, SET FLAG
417 22B3 208E25     ADD0    JSR   GLINE       GET NEW LINE FROM KEYBOARD
418 22B6 8C1124             STY   ACNT
419 22B9 A580               LDA   UT3
420 22BB F003               BZ    ADD01
421 22BD 4C6C23             JMP   ADDF        AF. IS SPECIAL CASE
422 22C0 E691D002   ADD01   IDB   FIRST
423 22C6 A591               LDA   FIRST
424 22C8 A692               LDX   FIRST+1
425 22CA 203322             JSR   FBA         COMPUTE LINE ADDRESSES
426                                           SAVE 1ST ADDR. OF LINE TO BE
427                                           MOVED FOR "INSERT" ADDR
428 22CD 8568               STA   UPTR2
429 22CF 8669               STX   UPTR2+1
430 22D1 A57A               LDA   NXTCHR
431 22D3 A67B               LDX   NXTCHR+1
432 22D5 1869FF             DEA
433 22D8 C9FF               CMP   #$FF
434 22DA D001               BNE   ADD1
435 22DC CA                 DEX
436 22DD 8566       ADD1    STA   UPTR1       "FROM" ADDR = OLD NXTCHR-1
437 22DF 8667               STX   UPTR1+1
438 22E1 186D1124           ADD   ACNT
439 22E5 9001               BCC   ADD2
440 22E7 E8                 INX
441 22E8 856A       ADD2    STA   UPTR3       "TO" ADDR = "FROM" ADDR
442                                           + LINE LENGTH
443 22EA 866B               STX   UPTR3+1
444 22EC A568               LDA   UPTR2       UPTR4 = - COUNT TO BE MOVED
445 22EE 38E57A             SUB   NXTCHR      = BEGINNING ADDR OF 1ST LINE
446 22F1 856C               STA   UPTR4       2B MOVED - OLD NXTCHR
447 22F3 A569               LDA   UPTR2+1
448 22F5 E57B               SBC   NXTCHR+1
449 22F7 856D               STA   UPTR4+1
450                                           NEW NXTCHR = OLD NXTCHR + AC
451 22F9 A57A               LDA   NXTCHR
452 22FB A67B               LDX   NXTCHR+1
453 22FD 186D1124           ADD   ACNT
454 2301 9001               BCC   ADD3
455 2303 E8                 INX
456 2304 ECC820     ADD3    CPX   EOB+1       BUFFER O'FLOW IF OLD NXTCHR
457 2307 9014               BLS   ADD5        +ACNT >= EOB
458 2309 F003               BEQ   ADD4
459 230B 4C4F20     ADERR   JMP   ERR
460 230E CDC720     ADD4    CMP   EOB
461 2311 900A               BLS   ADD5
462                                           WRITE BUFFER O'FLOW MESSAGE
463 2313 A915       ADD4A   LDA   #<BFOMS
464 2315 A224               LDX   #>BFOMS
465 2317 20D420             JSR   DISP
466 231A 4C2520             JMP   MON1
467                                           SAVE NEW NXTCHR
468 231D 857A       ADD5    STA   NXTCHR
469 231F 867B               STX   NXTCHR+1
470 2321 A000               LDY   #0
471                                           MOVE UPPER BUFFER
472 2323 B166       ADD6    LDA   *UPTR1,Y
473 2325 916A               STA   *UPTR3,Y
```

```
474 2327 C666                DEC   UPTR1
475 2329 A566                LDA   UPTR1
476 232B C9FF                CMP   #$FF
477 232D D002                BNE   ADD7
478 232F C667                DEC   UPTR1+1
479 2331 C66A        ADD7    DEC   UPTR3
480 2333 A56A                LDA   UPTR3
481 2335 C9FF                CMP   #$FF
482 2337 D002                BNE   ADD8
483 2339 C66B                DEC   UPTR3+1
484 233B E66CD002    ADD8    IDB   UPTR4
485 2341 D0E0                BNZ   ADD6
486 2343 A93E856C    ADD9    CDB   UPTR4,KBFR
487 234B AE1124              LDX   ACNT
488 234E A000                LDY   #0
489                                        MOVE NEW LINE TO BUFFER
490 2350 B16C        ADD9A   LDA   *UPTR4,Y
491 2352 9168                STA   *UPTR2,Y
492 2354 E66CD002    ADD9B   IDB   UPTR4
493 235A E668D002            IDB   UPTR2
494 2360 CA          ADD10   DEX
495 2361 D0ED                BNZ   ADD9A
496 2363 E678D002            IDB   LNCNT
497 2369 4CB322              JMP   ADD0
498                                        SPECIAL CASE - ADD TO
499                                        END OF BUFFER
500 236C A57A        ADDF    LDA   NXTCHR
501 236E A67B                LDX   NXTCHR+1
502 2370 8568                STA   UPTR2
503 2372 8669                STX   UPTR2+1
504 2374 186D1124            ADD   ACNT
505 2378 9001                BCC   ADDF1
506 237A E8                  INX
507 237B ECC820      ADDF1   CPX   EOB+1
508 237E 900A                BLS   ADDF4
509 2380 F003                BEQ   ADDF3
510 2382 4C1323      ADDF2   JMP   ADD4A
511 2385 CDC720      ADDF3   CMP   EOB
512 2388 B0F8                BGE   ADDF2
513 238A 857A        ADDF4   STA   NXTCHR
514 238C 867B                STX   NXTCHR+1
515 238E 4C4323              JMP   ADD9
516                  *
517                  * DELETE FUNCTION
518                  *
519                  * "TO" ADDR = BADR1
520                  * "FROM" ADDR = EADR2+1
521                  * COUNT = NXTCHR-EADR2+1
522                  * NXTCHR = NXTCHR - (EADR2+1-BADR1)
523                  * LNCNT = LNCNT - (SECOND-FIRST+1)
524                  *
525 2391 20B921        DELETE  JSR   FPRM       GET DELETE LIMITS
526 2394 A595                LDA   BADR1       "TO" ADDR
527 2396 8566                STA   UPTR1
528 2398 A596                LDA   BADR1+1
529 239A 8567                STA   UPTR1+1
530 239C A59B                LDA   EADR2       "FROM" ADDR
531 239E 8568                STA   UPTR2
532 23A0 A59C                LDA   EADR2+1
```

```
533 23A2 8569              STA   UPTR2+1
534 23A4 E668D002          IDB   UPTR2
535
536 23AA A568              LDA   UPTR2     COUNT = -(NXTCHR-EADR2+1)
537 23AC 38E57A            SUB   NXTCHR
538 23AF 856A              STA   UPTR3
539 23B1 A569              LDA   UPTR2+1
540 23B3 E57B              SBC   NXTCHR+1
541 23B5 856B              STA   UPTR3+1
542 23B7 056A              ORA   UPTR3
543 23B9 8D1124            STA   ACNT
544
545 23BC A566              LDA   UPTR1     COMPUTE NEW NXTCHR
546 23BE 38E568            SUB   UPTR2
547 23C1 AA                TAX
548 23C2 A567              LDA   UPTR1+1
549 23C4 E569              SBC   UPTR2+1
550 23C6 48                PHA
551 23C7 8A                TXA
552 23C8 18657A            ADD   NXTCHR
553 23CB 857A              STA   NXTCHR
554 23CD 68                PLA
555 23CE 657B              ADC   NXTCHR+1
556 23D0 857B              STA   NXTCHR+1
557
558 23D2 A593              LDA   SECOND    COMPUTE NEW LNCNT
559 23D4 49FF              CMA
560 23D6 186591            ADD   FIRST
561 23D9 AA                TAX
562 23DA A594              LDA   SECOND+1
563 23DC 49FF              CMA
564 23DE 6592              ADC   FIRST+1
565 23E0 48                PHA
566 23E1 8A                TXA
567 23E2 186578            ADD   LNCNT
568 23E5 8578              STA   LNCNT
569 23E7 68                PLA
570 23E8 6579              ADC   LNCNT+1
571 23EA 8579              STA   LNCNT+1
572 23EC AD1124            LDA   ACNT
573 23EF F01A              BZ    DL3
574 23F1 A000              LDY   #0
575 23F3 B168        DL2   LDA   *UPTR2,Y
576 23F5 9166              STA   *UPTR1,Y
577 23F7 E666D002          IDB   UPTR1
578 23FD E668D002          IDB   UPTR2
579 2403 E66AD002          IDB   UPTR3
580 2409 D0E8              BNZ   DL2
581 240B 4C0C26      DL3   JMP   CSPACE
582 240E 0000        BFA   DBL   0
583 2410 01          CLFLAG DATA 1
584 2411 00          ACNT  ***
585 2412 00          APTR  ***
586 2413 0000        MSPACE DBL  0
587 2415 C2D5C6C6    BFOMS ASC   'BUFFER OVERFLOW'
588 2424 8D                DATA  $8D
589                  *
590                  * LIST FUNCTION
591                  *
```

```
592 2425 20B921      LIST    JSR   FPRM
593 2428 A595                LDA   BADR1
594 242A 856E                STA   UPTR5
595 242C A596                LDA   BADR1+1
596 242E 856F                STA   UPTR5+1
597 2430 201227      LIST1   JSR   FORMAT    RETRIEVE & FORMAT LINE
598 2433 A944856A            CDB   UPTR3,LABEL-6 POINTER FOR CONVERT
599 243B A591                LDA   FIRST
600 243D A692                LDX   FIRST+1
601 243F E691D002            IDB   FIRST     BUMP LINE NUMBER
602 2445 20E724              JSR   CNVLNO    CONVERT LINE #
603 2448 A946                LDA   #<LABEL-4
604 244A A22A                LDX   #>LABEL-4
605 244C 20B020              JSR   PRT
606 244F 20B620              JSR   DTCC
607 2452 B008                BCS   LIST3
608 2454 E69DD002            IDB   IDIFF
609 245A D0D4                BNZ   LIST1
610 245C 4C2520      LIST3   JMP   MON1
611                  *
612                  * CONVERT AND DISPLAY MEMORY LIMITS
613                  *
614 245F 20EA20      MEMORY  JSR   RKB       WAIT FOR PERIOD
615 2462 103B                BPL   MERR
616 2464 A9CE856A            CDB   UPTR3,MLIMIT UPTR3 IS STORE PTR
617 246C AD1324              LDA   MSPACE    CONVERT AVAILABLE BUFFER SPA
618 246F AE1424              LDX   MSPACE+1
619 2472 20E724              JSR   CNVLNO
620 2475 A9E1856A            CDB   UPTR3,BFE CONVERT END OF
621 247D ADC820              LDA   EOB+1     BUFFER ADDRESS
622 2480 209826              JSR   CNV2HX
623 2483 ADC720              LDA   EOB
624 2486 209826              JSR   CNV2HX
625 2489 A9CE                LDA   #<MLIMIT
626 248B A224                LDX   #>MLIMIT
627 248D 20D420              JSR   DISP      DISPLAY NUMBERS
628 2490 A900                LDA   #0
629 2492 8566                STA   UPTR1     CLEAR RESULT SPACE
630 2494 8567                STA   UPTR1+1
631 2496 857C                STA   UT1       AND DIGIT COUNTER
632 2498 202B21      MEM1    JSR   FHEX      GET HEX DIGIT
633 249B 301D                BMI   MEM3      MINUS IF PERIOD
634 249D B003                BCS   MEM2      CARRY SET IF LEGAL HEX
635 249F 4C4F20      MERR    JMP   ERR
636 24A2 0666        MEM2    ASL   UPTR1
637 24A4 2667                ROL   UPTR1+1
638 24A6 0666                ASL   UPTR1
639 24A8 2667                ROL   UPTR1+1
640 24AA 0666                ASL   UPTR1
641 24AC 2667                ROL   UPTR1+1
642 24AE 0666                ASL   UPTR1
643 24B0 2667                ROL   UPTR1+1
644 24B2 0566                ORA   UPTR1     MERGE NEW DIGIT
645 24B4 8566                STA   UPTR1
646 24B6 E67C                INC   UT1       BUMP DIGIT COUNT
647 24B8 D0DE                BNZ   MEM1
648 24BA A57C        MEM3    LDA   UT1
649 24BC D003                BNZ   MEM4      ANYTHING TYPED?
650 24BE 4C2520              JMP   MON1      NO
```

```
24C1 A566       MEM4    LDA   UPTR1     YES, SET NEW MEMORY LIMIT
24C3 8DC720             STA   EOB
24C6 A567               LDA   UPTR1+1
24C8 8DC820             STA   EOB+1
24CB 4C0C26             JMP   CSPACE    UPDATE MEMORY DATA
24CE            MLIMIT  RES   5
24D3 A0CCC5C6           ASC   ' LEFT  '
24DA A0C5CFC2           ASC   ' EOB = '
24E1            BFE     RES   4
24E5 A0                 DATA  ' '
24E6 8D                 DATA  $8D
                *
                * CONVERT NUMBER IN A-X TO DECIMAL, LEAVE
                * RESULT IN (UPTR3), (UPTR3)+4
                *
24E7 857C       CNVLNO  STA   UT1
24E9 867D               STX   UT1+1
24EB A936856C           CDB   UPTR4,DTAB
24F3 A000               LDY   #0
24F5 A904               LDA   #4
24F7 857E               STA   UT2
24F9 B16C       CNV2    LDA   *UPTR4,Y
24FB 8D3425             STA   SUBTRA
24FE C8                 INY
24FF B16C               LDA   *UPTR4,Y
2501 8D3525             STA   SUBTRA+1
2504 C8                 INY
2505 A2B0       CNV3    LDX   #'0'
2507 A57C               LDA   UT1
2509 38ED3425   CNV4    SUB   SUBTRA
250D 48                 PHA
250E A57D               LDA   UT1+1
2510 ED3525             SBC   SUBTRA+1
2513 9008               BCC   CNV5
2515 857D               STA   UT1+1
2517 68                 PLA
2518 857C               STA   UT1
251A E8                 INX
251B D0EC               BNZ   CNV4
251D 68         CNV5    PLA
251E 8A                 TXA
251F A200               LDX   #0
2521 816A               STA   *UPTR3,X
2523 E66AD002           IDB   UPTR3
2529 C67E               DEC   UT2
252B D0CC               BNZ   CNV2
252D A57C               LDA   UT1
252F 09B0               ORA   #'0'
2531 816A               STA   *UPTR3,X
2533 60                 RTS
2534 0000       SUBTRA  DBL   0
2536 1027E803   DTAB    DBL   10000,1000,100,10
253E            KBFR    RES   80
                *
                * GET INPUT TEXT LINE FROM KEYBOARD
                *
258E 209E20     GLINE   JSR   TCRLF     ADVANCE TERMINAL LINE
2591 A000               LDY   #0
2593 20EA20     GL1     JSR   RKB       GET CHAR
```

```
710 2596 7022               BVS   EOL        OV ON IF CARRIAGE RETURN
711 2598 C9AE               CMP   #'.'
712 259A D004               BNE   CMP88
713 259C C000               CPY   #0
714 259E F06C               BEQ   CSPACE
715 25A0 C988       CMP88   CMP   #$88       BACKSPACE
716 25A2 D009               BNE   GL2
717 25A4 8DD425             STA   GLCHR
718 25A7 88                 DEY               YES, BUMP POINTER BACK
719 25A8 10E9               BPL   GL1         CHECK FOR BEG. OF LINE
720 25AA C8                 INY               WHOOPS!
721 25AB 10E6               BPL   GL1
722 25AD C04F       GL2     CPY   #79         CHECK RECORD LENGTH
723 25AF B024               BGE   TULONG      TOO BIG
724 25B1 993E25             STA   KBFR,Y      STORE CHAR IN STRING
725 25B4 8DD425             STA   GLCHR
726 25B7 C8                 INY               BUMP INDEX
727 25B8 D0D9               BNZ   GL1         AND GET NEXT CHAR.
728 25BA 993E25     EOL     STA   KBFR,Y      STORE C.R.
729 25BD ADD425             LDA   GLCHR
730 25C0 C988               CMP   #$88
731 25C2 F0CA               BEQ   GLINE
732 25C4 C8                 INY               BUMP COUNT
733 25C5 AD3E25             LDA   KBFR        CHECK FOR INITIAL PERIOD
734 25C8 C9AE               CMP   #'.'
735 25CA F040               BEQ   CSPACE
736 25CC AD2821             LDA   RKBT
737 25CF C000               CPY   #0
738 25D1 F0C0               BEQ   GL1         NULL LINE IF ZERO COUNT
739 25D3 60                 RTS
740 25D4 00         GLCHR   ***
741 25D5 A9DF       TULONG  LDA   #<TLM
742 25D7 A225               LDX   #>TLM       DISPLAY TOO LONG MESSAGE
743 25D9 20D420             JSR   DISP
744 25DC 4C2520     TL2     JMP   MON1
745 25DF CCC9CEC5   TLM     ASC   'LINE > 80 CHAR'
746 25ED 8D                 DATA  $8D
747                 *
748                 * WRITE TO OUTPUT MEDIUM
749                 *
750 25EE 20B921     WRITE   JSR   FPRM        GET WRITE LIMITS
751 25F1 A59B               LDA   EADR2       COMPUTE # OF CHAR'S
752 25F3 49FF               CMA
753 25F5 186595             ADD   BADR1
754 25F8 859D               STA   IDIFF
755 25FA A59C               LDA   EADR2+1
756 25FC 49FF               CMA
757 25FE 6596               ADC   BADR1+1
758 2600 859E               STA   IDIFF+1
759 2602 A595               LDA   BADR1
760 2604 A696               LDX   BADR1+1
761 2606 20AD20             JSR   WRTAPE      WRITE THE RECORD
762 2609 4C2520             JMP   MON1
763                                           COMPUTE REMAINING BUFFER SPA
764 260C ADC720     CSPACE  LDA   EOB
765 260F 38E57A             SUB   NXTCHR
766 2612 8D1324             STA   MSPACE
767 2615 ADC820             LDA   EOB+1
768 2618 E57B               SBC   NXTCHR+1
```

```
769 261A 8D1424            STA   MSPACE+1
770 261D 4C2B20            JMP   MON1A
771                 *
772                 * READ RECORDS FROM INPUT MEDIUM AND APPEND
773                 * TO EXISTING BUFFER TEXT
774                 *
775                 * UPTR4 = # OF LINES READ
776                 * UPTR1 = STORE POINTER
777                 * UPTR2 = - REMAINING AVAILABLE MEMORY
778                 * UT1 = NEW NXTCHR
779                 *
780 2620 A900      READ  LDA   #0
781 2622 856C            STA   UPTR4      CLEAR LINES READ
782 2624 856D            STA   UPTR4+1
783 2626 A57A            LDA   NXTCHR     INITIALIZE STORE POINTER
784 2628 8566            STA   UPTR1
785 262A 38EDC720        SUB   EOB
786 262E 8568            STA   UPTR2      AND SPACE COUNTER
787 2630 A57B            LDA   NXTCHR+1
788 2632 8567            STA   UPTR1+1
789 2634 EDC820          SBC   EOB+1
790 2637 8569            STA   UPTR2+1
791 2639 20EA20    READ2 JSR   RKB        WAIT FOR PERIOD
792 263C 1057            BPL   RERR
793 263E 20A720          JSR   RHDR       SKIP OVER TAPE HEADER
794 2641 20AA20    READ3 JSR   RDBYTE     GET TEXT BYTE
795 2644 B02B            BCS   READ5      CARRY ON IF END OF FILE
796 2646 A000            LDY   #0
797 2648 9166            STA   *UPTR1,Y   STORE NEW CHARACTER
798 264A E666D002        IDB   UPTR1      BUMP STORE POINTER
799 2650 C98D            CMP   #$8D       CARRIAGE RETURN?
800 2652 D00E            BNE   READ4
801 2654 E66CD002        IDB   UPTR4      YES, BUMP LINE COUNT
802 265A A566            LDA   UPTR1      AND SAVE NEW NXTCHR
803 265C 857C            STA   UT1
804 265E A567            LDA   UPTR1+1
805 2660 857D            STA   UT1+1
806 2662 E668D002  READ4 IDB   UPTR2
807 2668 D0D7            BNZ   READ3      GET NEXT CHAR
808 266A A989            LDA   #<BFM      BUFFER FULL
809 266C A226            LDX   #>BFM      TELL HIM SO
810 266E 20D420          JSR   DISP
811 2671 A57C      READ5 LDA   UT1        UPDATE NXTCHR
812 2673 857A            STA   NXTCHR
813 2675 A57D            LDA   UT1+1
814 2677 857B            STA   NXTCHR+1
815 2679 A578            LDA   LNCNT
816 267B 18656C          ADD   UPTR4      UPDATE EDITOR LINE COUNT
817 267E 8578            STA   LNCNT
818 2680 A579            LDA   LNCNT+1
819 2682 656D            ADC   UPTR4+1
820 2684 8579            STA   LNCNT+1
821 2686 4C0C26          JMP   CSPACE
822 2689 C2D5C6C6  BFM   ASC   'BUFFER FULL'
823 2694 8D              DATA  $8D
824 2695 4C4F20    RERR  JMP   ERR
825                 *
826                 * CONVERT A DATA TO HEX/ASCII, LEAVE RESULT
827                 * IN (UPTR3), (UPTR3)+1
```

```
828                     *
829 2698 A200           CNV2HX  LDX   #0
830 269A 48                     PHA             SAVE CHAR
831 269B 6A                     RORA
832 269C 6A                     RORA
833 269D 6A                     RORA            SHIFT 4 BITS RIGHT
834 269E 6A                     RORA            VERY SLOWLY
835 269F 20A726                 JSR   CNVN      CONVERT 4 BITS
836 26A2 68                     PLA
837 26A3 20A726                 JSR   CNVN      CONVERT RIGHT FOUR BITS
838 26A6 60                     RTS
839 26A7 290F           CNVN    AND   #$F
840 26A9 C90A                   CMP   #$A
841 26AB 9003                   BLS   CNVN2
842 26AD 186907                 ADD   #7
843 26B0 1869B0         CNVN2   ADD   #$B0
844 26B3 816A                   STA   *UPTR3,X
845 26B5 E66AD002               IDB   UPTR3
846 26BB 60                     RTS
847                     *
848                     * XMIT FIELD FROM (UPTR5) TO (UPTR6). END ON
849                     * BLANK OR CR. MINUS ON IF CR. UPDATES UPTR5 TO
850                     * NEXT NONBLANK FIELD. BLANKS PERMITTED BETWEEN
851                     * APOSTROPHES. HIGH BIT OF A AND UT1 ON FOR CR.
852                     * Y AND UT2 CONTAIN # OF CHAR'S IN STRING.
853                     * NUMBER OF COMMAS IN FIELD NOT IN ASCII
854                     * STRINGS IS LEFT IN UT1+1.
855                     *
856 26BC A000           FSCAN   LDY   #0
857 26BE 847C                   STY   UT1
858 26C0 847D                   STY   UT1+1
859 26C2 8480                   STY   UT3       CLEAR COMMA COUNT
860 26C4 B16E           FS1     LDA   *UPTR5,Y
861 26C6 C98D                   CMP   #$8D
862 26C8 F03F                   BEQ   FS5
863 26CA C9A7                   CMP   #$A7
864 26CC D008                   BNE   FS2
865 26CE 48                     PHA
866 26CF A57C                   LDA   UT1
867 26D1 4940                   EOR   #$40
868 26D3 857C                   STA   UT1
869 26D5 68                     PLA
870 26D6 C9A0           FS2     CMP   #' '
871 26D8 D004                   BNE   FS3
872 26DA 247C                   BIT   UT1
873 26DC 5011                   BVC   FS4
874 26DE C9AC           FS3     CMP   #','
875 26E0 D006                   BNE   FS3A
876 26E2 247C                   BIT   UT1
877 26E4 7002                   BVS   FS3A
878 26E6 E67D                   INC   UT1+1     BUMP COMMA COUNT
879 26E8 9170           FS3A    STA   *UPTR6,Y
880 26EA C8                     INY
881 26EB 847E                   STY   UT2
882 26ED D0D5                   BNZ   FS1
883 26EF C8             FS4     INY
884 26F0 B16E                   LDA   *UPTR5,Y
885 26F2 C9A0                   CMP   #' '
886 26F4 F0F9                   BEQ   FS4
```

```
887 26F6 C98D                CMP   #$8D
888 26F8 F00F                BEQ   FS5
889 26FA 98          FS4B    TYA
890 26FB 18656E              ADD   UPTR5
891 26FE 856E                STA   UPTR5
892 2700 9002                BCC   FS4C
893 2702 E66F                INC   UPTR5+1
894 2704 A47E        FS4C    LDY   UT2
895 2706 A57C                LDA   UT1
896 2708 60                  RTS
897 2709 A57C        FS5     LDA   UT1
898 270B 0980                ORA   #$80
899 270D 857C                STA   UT1
900 270F C8                  INY
901 2710 D0E8                BNZ   FS4B
902                  *
903                  * FORMAT RECORD FROM SOURCE BUFFER LOCATION
904                  * POINTED TO BY UPTR5. RESULT LEFT IN LABEL,
905                  * INST, OPND, CMT
906                  *
907 2712 A27A        FORMAT  LDX   #EOCMT-LNO-1
908 2714 A9A0                LDA   #' '
909 2716 9D362A      FM1     STA   LNO,X
910 2719 CA                  DEX
911 271A 10FA                BPL   FM1
912 271C A94A8570            CDB   UPTR6,LABEL
913 2724 A9FF                LDA   #-1
914 2726 858F                STA   SKEL
915 2728 8590                STA   CLASS
916 272A A000                LDY   #0
917 272C B16E                LDA   *UPTR5,Y
918 272E C9AA                CMP   #'*'
919 2730 F060                BEQ   XCL
920 2732 20BC26              JSR   FSCAN
921 2735 1006                BPL   XMTOP
922 2737 A98D                LDA   #$8D
923 2739 8D502A              STA   LABEL+6
924 273C 60                  RTS
925 273D A9528570    XMTOP   CDB   UPTR6,INST
926 2745 20BC26              JSR   FSCAN
927 2748 20B027              JSR   SRCOP
928 274B 247C                BIT   UT1
929 274D 1007                BPL   OPTST
930 274F A98D                LDA   #$8D
931 2751 8D562A              STA   INST+4
932 2754 D056                BNZ   FMTEND
933 2756 B023        OPTST   BCS   CLY
934 2758 A9578570            CDB   UPTR6,OPND
935 2760 A590                LDA   CLASS     LOOK FOR "TITL"
936 2762 D006                BNZ   OPT2
937 2764 A58F                LDA   SKEL
938 2766 C903                CMP   #3
939 2768 F026                BEQ   XMTC3
940 276A 20BC26      OPT2    JSR   FSCAN
941 276D 08                  PHP
942 276E A57D                LDA   UT1+1
943 2770 8580                STA   UT3
944 2772 28                  PLP
945 2773 1008                BPL   XMTCMT
```

```
946 2775 A98D              LDA   #$8D
947 2777 9170              STA   *UPTR6,Y
948 2779 D031              BNZ   FMTEND
949 277B A000       CLY    LDY   #0
950 277D 98         XMTCMT TYA
951 277E 38E909            SUB   #CMT-OPND-1 ADJUST COMMENT FIELD
952 2781 1002              BPL   XMTC2
953 2783 A900              LDA   #0
954 2785 186961     XMTC2  ADD   #<CMT
955 2788 8570              STA   UPTR6
956 278A A92A              LDA   #>CMT
957 278C 6900              ADC   #0
958 278E 8571              STA   UPTR6+1
959 2790 A000       XMTC3  LDY   #0
960                                    XMIT COMMENT FIELD
961 2792 B16E       XCL    LDA   *UPTR5,Y
962 2794 9170              STA   *UPTR6,Y
963 2796 C8                INY
964 2797 C98D              CMP   #$8D
965 2799 D0F7              BNE   XCL
966 279B A98D              LDA   #$8D
967 279D 8D852A            STA   LNO+79
968 27A0 847E              STY   UT2
969 27A2 98                TYA
970 27A3 18656E            ADD   UPTR5
971 27A6 856E              STA   UPTR5
972 27A8 9002              BCC   FMTEND
973 27AA E66F              INC   UPTR5+1
974 27AC A590       FMTEND LDA   CLASS
975 27AE 60                RTS
976 27AF 60                RTS
977                 *
978                 * SEARCH OPCODE TABLE FOR INSTRUCTION AT INST
979                 * CARRY RETURNS ON IF INST. REQUIRES OPERAND
980                 *
981 27B0 A9FF       SRCOP  LDA   #-1
982 27B2 858F              STA   SKEL       CLEAR INST. SKELETON
983 27B4 A9068572          CDB   UPTR7,OPTBL
984 27BC A25D              LDX   #93
985 27BE A000       SRCOP1 LDY   #0
986 27C0 AD522A            LDA   INST
987 27C3 D172              CMP   *UPTR7,Y
988 27C5 F002              BEQ   SR1
989 27C7 D02A              BNE   SRCOP3
990 27C9 C8         SR1    INY
991 27CA AD532A            LDA   INST+1
992 27CD D172              CMP   *UPTR7,Y
993 27CF F002              BEQ   SR2
994 27D1 D020              BNE   SRCOP3
995 27D3 C8         SR2    INY
996 27D4 AD542A            LDA   INST+2
997 27D7 D172              CMP   *UPTR7,Y
998 27D9 D018              BNE   SRCOP3
999 27DB C8         SR3    INY
000 27DC AD552A            LDA   INST+3
001 27DF D172              CMP   *UPTR7,Y
002 27E1 D010              BNE   SRCOP3
003 27E3 C8                INY
004 27E4 B172              LDA   *UPTR7,Y
```

```
005 27E6 858F              STA   SKEL
006 27E8 C8                INY
007 27E9 B172              LDA   *UPTR7,Y
008 27EB 48                PHA
009 27EC 297F              AND   #$7F
010 27EE 8590      SRCOP2  STA   CLASS
011 27F0 68                PLA
012 27F1 0A                ASLA
013 27F2 60                RTS
014 27F3 A572      SRCOP3  LDA   UPTR7
015 27F5 186906            ADD   #6
016 27F8 8572              STA   UPTR7
017 27FA 9002              BCC   SRCOP4
018 27FC E673              INC   UPTR7+1
019 27FE CA        SRCOP4  DEX
020 27FF D0BD              BNZ   SRCOP1
021 2801 A9FF      ILOP    LDA   #-1
022 2803 48                PHA
023 2804 30E8              BMI   SRCOP2
024                *
025 2806 A4A4A4A0  OPTBL   ASC   '$$$ '
026 280A 0C80              DBL   $800C
027 280C AAAAAAA0          ASC   '*** '
028 2810 0A80              DBL   $800A
029 2812 C1C4C3A0          ASC   'ADC '
030 2816 6101              DBL   $161
031 2818 C1C4C4A0          ASC   'ADD '
032 281C 610D              DBL   $D61
033 281E C1CEC4A0          ASC   'AND '
034 2822 2101              DBL   $121
035 2824 C1D3C3A0          ASC   'ASC '
036 2828 0900              DBL   9
037 282A C1D3CCA0          ASC   'ASL '
038 282E 0602              DBL   $206
039 2830 C1D3CCC1          ASC   'ASLA'
040 2834 0A83              DBL   $830A
041 2836 C2BCA0A0          ASC   'B<  '
042 283A 3004              DBL   $430
043 283C C2BEBDA0          ASC   'B>= '
044 2840 1004              DBL   $410
045 2842 C2C3C3A0          ASC   'BCC '
046 2846 9004              DBL   $490
047 2848 C2C3D3A0          ASC   'BCS '
048 284C B004              DBL   $4B0
049 284E C2C5A0A0          ASC   'BE  '
050 2852 F004              DBL   $4F0
051 2854 C2C5D1A0          ASC   'BEQ '
052 2858 F004              DBL   $4F0
053 285A C2C7C5A0          ASC   'BGE '
054 285E B004              DBL   $4B0
055 2860 C2C9D4A0          ASC   'BIT '
056 2864 2405              DBL   $524
057 2866 C2CCD3A0          ASC   'BLS '
058 286A 9004              DBL   $490
059 286C C2CDA0A0          ASC   'BM  '
060 2870 3004              DBL   $430
061 2872 C2CDC9A0          ASC   'BMI '
062 2876 3004              DBL   $430
063 2878 C2CEC5A0          ASC   'BNE '
```

```
064 287C D004               DBL   $4D0
065 287E C2CEDAA0           ASC   'BNZ '
066 2882 D004               DBL   $4D0
067 2884 C2D0A0A0           ASC   'BP  '
068 2888 1004               DBL   $410
069 288A C2D0CCA0           ASC   'BPL '
070 288E 1004               DBL   $410
071 2890 C2D2CBA0           ASC   'BRK '
072 2894 0083               DBL   $8300
073 2896 C2D6C3A0           ASC   'BVC '
074 289A 5004               DBL   $450
075 289C C2D6D3A0           ASC   'BVS '
076 28A0 7004               DBL   $470
077 28A2 C2DAA0A0           ASC   'BZ  '
078 28A6 F004               DBL   $4F0
079 28A8 C3C4C2A0           ASC   'CDB '
080 28AC 050D               DBL   $D05
081 28AE C3CCC1A0           ASC   'CLA '
082 28B2 028D               DBL   $8D02
083 28B4 C3CCC3A0           ASC   'CLC '
084 28B8 1883               DBL   $8318
085 28BA C3CCC4A0           ASC   'CLD '
086 28BE D883               DBL   $83D8
087 28C0 C3CCC9A0           ASC   'CLI '
088 28C4 5883               DBL   $8358
089 28C6 C3CCD6A0           ASC   'CLV '
090 28CA B883               DBL   $83B8
091 28CC C3CDC1A0           ASC   'CMA '
092 28D0 038D               DBL   $8D03
093 28D2 C3CDD0A0           ASC   'CMP '
094 28D6 C101               DBL   $1C1
095 28D8 C3D0D8A0           ASC   'CPX '
096 28DC E006               DBL   $6E0
097 28DE C3D0D9A0           ASC   'CPY '
098 28E2 C006               DBL   $6C0
099 28E4 C4C1D4C1           ASC   'DATA'
100 28E8 0700               DBL   7
101 28EA C4C2CCA0           ASC   'DBL '
102 28EE 0800               DBL   8
103 28F0 C4C5C1A0           ASC   'DEA '
104 28F4 008D               DBL   $8D00
105 28F6 C4C5C3A0           ASC   'DEC '
106 28FA C602               DBL   $2C6
107 28FC C4C5D8A0           ASC   'DEX '
108 2900 CA83               DBL   $83CA
109 2902 C4C5D9A0           ASC   'DEY '
110 2906 8883               DBL   $8388
111 2908 C4C9A0A0           ASC   'DI  '
112 290C 7883               DBL   $8378
113 290E C5C9A0A0           ASC   'EI  '
114 2912 5883               DBL   $8358
115 2914 C5CEC4A0           ASC   'END '
116 2918 0280               DBL   $8002
117 291A C5CFD2A0           ASC   'EOR '
118 291E 4101               DBL   $141
119 2920 C5D1D5A0           ASC   'EQU '
120 2924 0400               DBL   4
121 2926 C5D3D0A0           ASC   'ESP '
122 292A 0B80               DBL   $800B
```

```
123 292C C9C4C2A0           ASC   'IDB '
124 2930 060D               DBL   $D06
125 2932 C9CEC1A0           ASC   'INA '
126 2936 018D               DBL   $8D01
127 2938 C9CEC3A0           ASC   'INC '
128 293C E602               DBL   $2E6
129 293E C9CED8A0           ASC   'INX '
130 2942 E883               DBL   $83E8
131 2944 C9CED9A0           ASC   'INY '
132 2948 C883               DBL   $83C8
133 294A CACDD0A0           ASC   'JMP '
134 294E 4C07               DBL   $74C
135 2950 CAD3D2A0           ASC   'JSR '
136 2954 2008               DBL   $820
137 2956 CCC4C1A0           ASC   'LDA '
138 295A A101               DBL   $1A1
139 295C CCC4D8A0           ASC   'LDX '
140 2960 A209               DBL   $9A2
141 2962 CCC4D9A0           ASC   'LDY '
142 2966 A00A               DBL   $AA0
143 2968 CCD3D2A0           ASC   'LSR '
144 296C 4602               DBL   $246
145 296E CCD3D2C1           ASC   'LSRA'
146 2972 4A83               DBL   $834A
147 2974 CECFD0A0           ASC   'NOP '
148 2978 EA83               DBL   $83EA
149 297A CFD2C1A0           ASC   'ORA '
150 297E 0101               DBL   $101
151 2980 CFD2C7A0           ASC   'ORG '
152 2984 0000               DBL   0
153 2986 D0C8C1A0           ASC   'PHA '
154 298A 4883               DBL   $8348
155 298C D0C8D0A0           ASC   'PHP '
156 2990 0883               DBL   $8308
157 2992 D0CCC1A0           ASC   'PLA '
158 2996 6883               DBL   $8368
159 2998 D0CCD0A0           ASC   'PLP '
160 299C 2883               DBL   $8328
161 299E D2C5CDA0           ASC   'REM '
162 29A2 0580               DBL   $8005
163 29A4 D2C5D3A0           ASC   'RES '
164 29A8 0600               DBL   6
165 29AA D2CFCCA0           ASC   'ROL '
166 29AE 2602               DBL   $226
167 29B0 D2CFCCC1           ASC   'ROLA'
168 29B4 2A83               DBL   $832A
169 29B6 D2CFD2A0           ASC   'ROR '
170 29BA 6602               DBL   $266
171 29BC D2CFD2C1           ASC   'RORA'
172 29C0 6A83               DBL   $836A
173 29C2 D2D4C9A0           ASC   'RTI '
174 29C6 4083               DBL   $8340
175 29C8 D2D4D3A0           ASC   'RTS '
176 29CC 6083               DBL   $8360
177 29CE D3C2C3A0           ASC   'SBC '
178 29D2 E101               DBL   $1E1
179 29D4 D3C5C3A0           ASC   'SEC '
180 29D8 3883               DBL   $8338
181 29DA D3C5C4A0           ASC   'SED '
```

```
182 29DE F883                DBL   $83F8
183 29E0 D3C5C9A0            ASC   'SEI '
184 29E4 7883                DBL   $8378
185 29E6 D3D0C4A0            ASC   'SPD '
186 29EA 0100                DBL   1
187 29EC D3D4C1A0            ASC   'STA '
188 29F0 8101                DBL   $181
189 29F2 D3D4D8A0            ASC   'STX '
190 29F6 860B                DBL   $B86
191 29F8 D3D4D9A0            ASC   'STY '
192 29FC 840C                DBL   $C84
193 29FE D3D5C2A0            ASC   'SUB '
194 2A02 E10D                DBL   $DE1
195 2A04 D4C1D8A0            ASC   'TAX '
196 2A08 AA83                DBL   $83AA
197 2A0A D4C1D9A0            ASC   'TAY '
198 2A0E A883                DBL   $83A8
199 2A10 D4C3C1A0            ASC   'TCA '
200 2A14 048D                DBL   $8D04
201 2A16 D4C9D4CC            ASC   'TITL'
202 2A1A 0300                DBL   3
203 2A1C D4D3D8A0            ASC   'TSX '
204 2A20 BA83                DBL   $83BA
205 2A22 D4D8C1A0            ASC   'TXA '
206 2A26 8A83                DBL   $838A
207 2A28 D4D8D3A0            ASC   'TXS '
208 2A2C 9A83                DBL   $839A
209 2A2E D4D9C1A0            ASC   'TYA '
210 2A32 9883                DBL   $8398
211                  *
212                  * LISTING BUFFER
213                  *
214 2A34 0000                DBL   0
215 2A36             LNO     RES   4
216 2A3A             LOC     RES   5
217 2A3F             CONT    RES   11
218 2A4A             LABEL   RES   8,' '
219 2A52             INST    RES   5
220 2A57             OPND    RES   10,' '
221 2A61             CMT     RES   80
222 2AB1 8D          EOCMT   DATA  $8D
223                  * SYMBOL TABLE HANDLERS
224                  *
225                  * ENTER SYMBOL AT LOCATION "LABEL" INTO SYMBOL
226                  * TABLE WITH VALUE FOUND AT UT1, UT1+1.
227                  *
228 2AB2 A9FF        ESYMTB  LDA   #-1
229 2AB4 8581                STA   UT3+1      SET MOVE FLAG
230 2AB6 A59F                LDA   NSYMS
231 2AB8 8568                STA   UPTR2
232 2ABA 49FF                CMA
233 2ABC 85A1                STA   SYMCNT     SYMCNT = -(NSYMS+1)
234 2ABE A5A0                LDA   NSYMS+1
235 2AC0 8569                STA   UPTR2+1
236 2AC2 49FF                CMA
237 2AC4 85A2                STA   SYMCNT+1
238 2AC6 A5A5                LDA   SYMTAB     SET SYMBOL POINTER
239 2AC8 85A3                STA   SYMPTR
240 2ACA A5A6                LDA   SYMTAB+1
```

```
241 2ACC 85A4              STA   SYMPTR+1
242 2ACE E6A1D002  EST0    IDB   SYMCNT     BUMP SYMBOL COUNTER
243 2AD4 F022              BZ    ESYM0
244 2AD6 A000      EST1    LDY   #0
245 2AD8 A206              LDX   #6
246 2ADA B1A3      EST2    LDA   *SYMPTR,Y  COMPARE SYMBOLS
247 2ADC 0980              ORA   #$80       FORCE CODE BIT ON
248 2ADE D94A2A            CMP   LABEL,Y
249 2AE1 9008              BLS   EST4
250 2AE3 F002              BEQ   EST3
251 2AE5 B013              BGE   ESYM
252 2AE7 C8        EST3    INY
253 2AE8 CA                DEX
254 2AE9 D0EF              BNZ   EST2
255 2AEB A5A3      EST4    LDA   SYMPTR
256 2AED 186908            ADD   #8         FORM ADDRESS OF NEXT
257 2AF0 85A3              STA   SYMPTR     SYMBOL IN TABLE
258 2AF2 90DA              BCC   EST0
259 2AF4 E6A4              INC   SYMPTR+1
260 2AF6 D0D6              BNZ   EST0
261 2AF8 E681      ESYM0   INC   UT3+1
262 2AFA A568      ESYM    LDA   UPTR2
263 2AFC 0A                ASLA
264 2AFD 2669              ROL   UPTR2+1
265 2AFF 0A                ASLA
266 2B00 2669              ROL   UPTR2+1
267 2B02 0A                ASLA
268 2B03 2669              ROL   UPTR2+1
269 2B05 0907              ORA   #7
270 2B07 1865A5            ADD   SYMTAB
271 2B0A 8568              STA   UPTR2
272 2B0C A569              LDA   UPTR2+1
273 2B0E 65A6              ADC   SYMTAB+1
274 2B10 8569              STA   UPTR2+1
275 2B12 CDC820            CMP   EOB+1
276 2B15 900B              BLS   ESYM3
277 2B17 F002              BEQ   ESYM2
278 2B19 B052              BGE   STOFLO
279 2B1B A568      ESYM2   LDA   UPTR2
280 2B1D CDC720            CMP   EOB
281 2B20 B04B              BGE   STOFLO
282 2B22 E681      ESYM3   INC   UT3+1
283 2B24 D02A              BNZ   ENTSYM
284 2B26 A568              LDA   UPTR2
285 2B28 38E908            SUB   #8
286 2B2B 856C              STA   UPTR4
287 2B2D A569              LDA   UPTR2+1
288 2B2F E900              SBC   #0
289 2B31 856D              STA   UPTR4+1
290 2B33 A000              LDY   #0
291 2B35 A208      ESYM3A  LDX   #8
292 2B37 B16C      ESYM4   LDA   *UPTR4,Y
293 2B39 9168              STA   *UPTR2,Y
294 2B3B 88                DEY
295 2B3C 98                TYA
296 2B3D C9FF              CMP   #$FF
297 2B3F D004              BNE   ESYM5
298 2B41 C66D              DEC   UPTR4+1
299 2B43 C669              DEC   UPTR2+1
```

```
2B45 CA          ESYM5   DEX
2B46 D0EF                BNZ   ESYM4
2B48 E6A1D002            IDB   SYMCNT
2B4E D0E5                BNZ   ESYM3A
2B50 A206        ENTSYM  LDX   #6
2B52 A000                LDY   #0
2B54 B94A2A      ENTSM2  LDA   LABEL,Y
2B57 91A3                STA   *SYMPTR,Y
2B59 C8                  INY
2B5A CA                  DEX
2B5B D0F7                BNZ   ENTSM2
2B5D A57C                LDA   UT1        XMIT SYMBOL VALUE
2B5F 91A3                STA   *SYMPTR,Y
2B61 C8                  INY
2B62 A57D                LDA   UT1+1
2B64 91A3                STA   *SYMPTR,Y
2B66 E69F                INC   NSYMS
2B68 D002                BNZ   ENTSM3
2B6A E6A0                INC   NSYMS+1
2B6C 60          ENTSM3  RTS
2B6D A977        STOFLO  LDA   #<STOFM
2B6F A22B                LDX   #>STOFM
2B71 20B020              JSR   PRT
2B74 6C0020              JMP   *MON
2B77 D3D9CDC2    STOFM   ASC   'SYMBOL TABLE OVERFLOW'
2B8C 8D                  DATA  $8D
                 *
                 * SEARCH SYMBOL TABLE FOR SYMBOL POINTED TO BY
                 * UPTR1. RETURN WITH CARRY ON IF SYMBOL FOUND.
                 * VALUE OF SYMBOL IS LEFT IN UT1, UT1+1. HIGH
                 * 6 BITS OF A ARE CODE BITS.
                 * UPTR1 IS UPDATED TO POINT TO THE SYMBOL
                 * DELIMITER.
                 * HIGH BITS ARE NORMALLY ON FOR ALL
                 * SIX CHAR'S. BITS OFF HAVE FOLLOWING
                 * MEANINGS:
                 *
                 * BIT 7 - SCRATCHPAD SYMBOL
                 *
                 * BIT 6 - MULTIPLY DEFINED SYMBOL
                 *
                 * BIT 5 - SYMBOL HAS ASCII DEFINITION
                 *
2B8D A59F        SRCSTB  LDA   NSYMS      SET UP COUNT
2B8F 49FF                CMA
2B91 85A1                STA   SYMCNT
2B93 A5A0                LDA   NSYMS+1
2B95 49FF                CMA
2B97 85A2                STA   SYMCNT+1
2B99 A006                LDY   #6
2B9B A9A0                LDA   #' '       CLEAR SYMBOL AREA
2B9D 99522C      SRCST2  STA   SYMBOL-1,Y
2BA0 88                  DEY
2BA1 D0FA                BNZ   SRCST2
2BA3 A5A5                LDA   SYMTAB     SET UP SYMBOL TABLE
2BA5 85A3                STA   SYMPTR     ADDRESS FOR SEARCH
2BA7 A5A6                LDA   SYMTAB+1
2BA9 85A4                STA   SYMPTR+1
                                          TRANSFER SYMBOL
```

```
359 2BAB A000               LDY   #0
360 2BAD A206               LDX   #6
361 2BAF B166       SRCST3  LDA   *UPTR1,Y
362 2BB1 201F2C             JSR   DLMTR     CHECK FOR END OF FIELD
363 2BB4 B022               BCS   SVPTR     FOUND IT, SEARCH TABLE
364 2BB6 D003               BNZ   SRST3A
365 2BB8 C8                 INY
366 2BB9 D0F4               BNZ   SRCST3    WASTE RIGHT PARENTHESIS
367 2BBB 99532C     SRST3A  STA   SYMBOL,Y
368 2BBE C8                 INY
369 2BBF CA                 DEX
370 2BC0 D0ED               BNZ   SRCST3
371                                         SYMBOL 6 CHAR'S. NEXT CHAR
372                                         MUST BE DELIMITER.
373 2BC2 B166               LDA   *UPTR1,Y
374 2BC4 201F2C             JSR   DLMTR
375 2BC7 B00F               BCS   SVPTR     OK, SEARCH TABLE
376 2BC9 C8         SRCST4  INY             SYMBOL TOO LONG
377 2BCA B166               LDA   *UPTR1,Y  SCAN FOR DELIMITER
378 2BCC 201F2C             JSR   DLMTR
379 2BCF 90F8               BCC   SRCST4
380 2BD1 847E               STY   UT2
381 2BD3 20472C     NOFND   JSR   UPDPTR
382 2BD6 18                 CLC
383 2BD7 60                 RTS
384 2BD8 847E       SVPTR   STY   UT2
385 2BDA E6A1D002   SRCST5  IDB   SYMCNT
386 2BE0 F0F1               BZ    NOFND
387 2BE2 A206               LDX   #6
388 2BE4 A000               LDY   #0
389 2BE6 8C592C             STY   CBITS     CLEAR CODE BITS
390 2BE9 B1A3       SRCST6  LDA   *SYMPTR,Y
391 2BEB 0980               ORA   #$80
392 2BED D9532C             CMP   SYMBOL,Y
393 2BF0 D020               BNE   NXTSYM    TRY NEXT SYMBOL
394 2BF2 B1A3               LDA   *SYMPTR,Y GET CODE BIT
395 2BF4 2A                 ROLA
396 2BF5 2E592C             ROL   CBITS
397 2BF8 C8                 INY
398 2BF9 CA                 DEX
399 2BFA D0ED               BNZ   SRCST6
400 2BFC B1A3               LDA   *SYMPTR,Y GET SYMBOL VALUE
401 2BFE 857C               STA   UT1
402 2C00 C8                 INY
403 2C01 B1A3               LDA   *SYMPTR,Y
404 2C03 857D               STA   UT1+1
405 2C05 20472C             JSR   UPDPTR
406 2C08 AD592C             LDA   CBITS     GET CODE BITS
407 2C0B 0A                 ASLA            LEFT JUSTIFY
408 2C0C 0A                 ASLA
409 2C0D 8D592C             STA   CBITS
410 2C10 38                 SEC
411 2C11 60                 RTS
412 2C12 A5A3       NXTSYM  LDA   SYMPTR    UPDATE POINTER
413 2C14 186908             ADD   #8        TO GET NEXT SYMBOL
414 2C17 85A3               STA   SYMPTR
415 2C19 90BF               BCC   SRC8T5
416 2C1B E6A4               INC   SYMPTR+1
417 2C1D D0BB               BNZ   SRC8T5
```

```
418                     *
419                     * SEARCH FOR LEGAL DELIMITER
420                     * RETURN WITH CARRY OFF IF NOT
421                     * DELIMITER. A=0 FOR RIGHT PAREN.
422                     *
423 2C1F C9AA           DLMTR   CMP   #'*'
424 2C21 F01E                   BEQ   EOS
425 2C23 C9AF                   CMP   #'/'
426 2C25 F01A                   BEQ   EOS
427 2C27 C9A0                   CMP   #' '
428 2C29 F016                   BEQ   EOS
429 2C2B C98D                   CMP   #$8D
430 2C2D F012                   BEQ   EOS
431 2C2F C9A9                   CMP   #')'
432 2C31 F010                   BEQ   RP
433 2C33 C9AC                   CMP   #','
434 2C35 F00A                   BEQ   EOS
435 2C37 C9AB                   CMP   #'+'
436 2C39 F006                   BEQ   EOS
437 2C3B C9AD                   CMP   #'-'
438 2C3D F002                   BEQ   EOS
439 2C3F 18                     CLC
440 2C40 60                     RTS
441 2C41 38             EOS     SEC
442 2C42 60                     RTS
443 2C43 A900           RP      LDA   #0
444 2C45 18                     CLC
445 2C46 60                     RTS
446                     *
447                     * UPDATE UPTR1 TO POINT TO DELIMITER
448                     *
449 2C47 A566           UPDPTR  LDA   UPTR1
450 2C49 18657E                 ADD   UT2
451 2C4C 8566                   STA   UPTR1
452 2C4E 9002                   BCC   UPDRTN
453 2C50 E667                   INC   UPTR1+1
454 2C52 60             UPDRTN  RTS
455 2C53                SYMBOL  RES   6
456 2C59 00             CBITS   ***
457                     *
458                     * FIELD POINTED TO BY UPTR1 IS SCANNED.
459                     * ITS VALUE IS RETURNED IN UT4, UT4+1. THE
460                     * A REGISTER BITS RETURN SET TO THE FOLLOWING
461                     * VALUES:
462                     *
463                     * BIT 7 - ON IF UNDEFINED OR OVERFLOW
464                     * BIT 6 - ON IF LEGAL SCRATCHPAD ADDRESS
465                     * BIT 5 - ON IF LEGAL GEN. MEM. ADDRESS
466                     * BIT 4 - ON IF LEGAL IMMEDIATE ADDRESS
467                     * BIT 3 - UNUSED
468                     * BIT 2 - ON IF TERMINATOR WAS A COMMA
469                     * BIT 1 - ON IF FIELD WAS ASCII
470                     * BIT 0 - ON IF OPERAND WAS HALF SYMBOL
471                     *
472                     * UPTR1 IS UPDATED TO POINT TO CHARACTER AFTER
473                     * THE ONE WHICH CAUSED TERMINATION OF THE SCAN.
474                     *
475 2C5A A900           OPDSCN  LDA   #0
476 2C5C 8582                   STA   UT4
```

```
2C5E 8583              STA   UT4+1      CLEAR FLAGS AND RESULT
2C60 85AB              STA   NTERMS
2C62 A900              LDA   #0
2C64 858E              STA   OPSTAT
2C66 A9FF              LDA   #$FF
2C68 858B              STA   HMASK      AND HALF SYMBOL MASKS
2C6A 858A              STA   LMASK
2C6C A9AB              LDA   #'+'       INITIAL OPERATOR '+'
2C6E 858C      OPSC2   STA   OPR8R
2C70 A000      OPSC3   LDY   #0
2C72 8485              STY   TVAL       CLEAR TEMP SPACE
2C74 8486              STY   TVAL+1
2C76 B166              LDA   *UPTR1,Y   GET CHAR
2C78 D9A42C    TSRC    CMP   CHTBL,Y    SEARCH TABLE
2C7B 9008              BLS   TSRC1
2C7D C8                INY
2C7E D9A42C            CMP   CHTBL,Y
2C81 B003              BGE   TSRC2
2C83 9009              BLS   CHFND
2C85 C8        TSRC1   INY
2C86 C8        TSRC2   INY
2C87 C01A              CPY   #FJTBL-CHTBL
2C89 90ED              BLS   TSRC
2C8B 4C7E2E            JMP   UNDFT
2C8E B9BE2C    CHFND   LDA   FJTBL,Y
2C91 48                PHA
2C92 88                DEY              FORCE TARGET
2C93 B9BE2C            LDA   FJTBL,Y    ADDRESS ONTO STACK
2C96 48                PHA
2C97 A000              LDY   #0         CLEAR Y FOR SCAN
2C99 60                RTS              GO SCAN FIELD
2C9A E666D002  SVOP    IDB   UPTR1      BUMP PAST + OR -
2CA0 A58D              LDA   TMN8R
2CA2 D0CA              BNZ   OPSC2
                                        CHARACTER TABLE
2CA4 ABACADAE  CHTBL   ASC   '+,-.'
2CA8 A0A1ACAD          ASC   ' !,-'
2CAC 8D8EA7A8          DATA  $8D,$8E,$A7,$A8
2CB0 BEBFBCBD          ASC   '>?<='
2CB4 A5A6C0C1          ASC   '%&@A'
2CB8 A4A5B0BA          ASC   '$%0:'
2CBC C1DB              ASC   'A['
                                        JUMP TABLE
2CBE 992C      FJTBL   DBL   SVOP-1     +
2CC0 992C              DBL   SVOP-1     -
2CC2 A32E              DBL   EOSCAN-1   BLANK
2CC4 A32E              DBL   EOSCAN-1   ,
2CC6 A32E              DBL   EOSCAN-1   C.R.
2CC8 802D              DBL   FASC-1     APOSTROPHE
2CCA 822E              DBL   HSH-1      >
2CCC 9B2E              DBL   HSL-1      <
2CCE F32D              DBL   FBIN-1     %
2CD0 432E              DBL   FOCT-1     @
2CD2 4F2E              DBL   FHXDC-1    $
2CD4 0C2D              DBL   FDCML-1    0-9
2CD6 5F2E              DBL   FSYM-1     A-Z
               *
               * END OF TERM PROCEDURE. CONTROL REACHES
               * HERE WHEN ANY TERMINATING CHARACTER IS
```

```
                        * ENCOUNTERED DURING A SCAN
                        *
2CD8 A58C       EOTERM  LDA   OPR8R     GET LEADING OPERATOR
2CDA E6AB               INC   NTERMS
2CDC C9AB               CMP   #'+'      ADDITIVE TERM
2CDE F013               BEQ   EOT2
2CE0 A586               LDA   TVAL+1    NO, TWO'S COMPLEMENT
2CE2 49FF               CMA             THE TERM BEFORE
2CE4 8586               STA   TVAL+1    ADDING
2CE6 A585               LDA   TVAL
2CE8 49FF1869           TCA
2CED 8585               STA   TVAL
2CEF D002               BNZ   EOT2
2CF1 E686               INC   TVAL+1    PROPAGATE CARRY
2CF3 A582       EOT2    LDA   UT4
2CF5 186585             ADD   TVAL
2CF8 8582               STA   UT4
2CFA A583               LDA   UT4+1     ADD NEW TERM
2CFC 6586               ADC   TVAL+1
2CFE 8583               STA   UT4+1
2D00 98                 TYA             UPDATE UPTR1
2D01 186566             ADD   UPTR1
2D04 8566               STA   UPTR1
2D06 9002               BCC   EOT3
2D08 E667               INC   UPTR1+1
2D0A 4C702C     EOT3    JMP   OPSC3     AND SCAN NEXT TERM
                        *
                        * DECIMAL TERM PROCESSOR
                        *
2D0D B166       FDCML   LDA   *UPTR1,Y
2D0F C9B0               CMP   #'0'      CHECK FOR LEGAL DIGIT
2D11 9039               BLS   EODEC
2D13 C9BA               CMP   #$BA
2D15 B035               BGE   EODEC
2D17 290F               AND   #$F       CLEAR ASCII CONTROL BITS
2D19 8584               STA   ECODE     SAVE NEW DIGIT
2D1B 20EF2D             JSR   TSHF      OLD RESULT TIMES 10
2D1E B02F               BCS   NOFLO     FIELD OVERFLOW
2D20 A585               LDA   TVAL
2D22 A686               LDX   TVAL+1
2D24 20EF2D             JSR   TSHF
2D27 B026               BCS   NOFLO
2D29 20EF2D             JSR   TSHF
2D2C B021               BCS   NOFLO
2D2E 186585             ADD   TVAL
2D31 8585               STA   TVAL
2D33 8A                 TXA
2D34 6586               ADC   TVAL+1
2D36 B017               BCS   NOFLO
2D38 8586               STA   TVAL+1
2D3A A584               LDA   ECODE     NOW ADD NEW DIGIT
2D3C 186585             ADD   TVAL
2D3F 8585               STA   TVAL
2D41 A586               LDA   TVAL+1
2D43 6900               ADC   #0
2D45 B008               BCS   NOFLO
2D47 8586               STA   TVAL+1
2D49 C8                 INY
2D4A D0C1               BNZ   FDCML
```

```
595 2D4C 4C302E      EODEC   JMP   EONUM
596                  *
597                  * OVERFLOW WHILE SCANNING NUMERIC FIELD
598                  * OR ZERO LENGTH ASCII FIELD
599                  *
600 2D4F A9CC        NOFLO   LDA   #'L'
601 2D51 8584        NOFLO1  STA   ECODE
602 2D53 A980                LDA   #$80
603 2D55 858E                STA   OPSTAT
604 2D57 B166        NOFLO2  LDA   *UPTR1,Y  SCAN FOR TERMINATOR
605 2D59 C9A0                CMP   #' '
606 2D5B F013                BEQ   NF1
607 2D5D C98D                CMP   #$8D
608 2D5F F00F                BEQ   NF1
609 2D61 C9AC                CMP   #','
610 2D63 F003                BEQ   NF0
611 2D65 C8                  INY
612 2D66 D0EF                BNZ   NOFLO2
613 2D68 48          NF0     PHA
614 2D69 A904                LDA   #4
615 2D6B 058E                ORA   OPSTAT
616 2D6D 858E                STA   OPSTAT
617 2D6F 68                  PLA
618 2D70 858D        NF1     STA   TMN8R
619 2D72 98                  TYA
620 2D73 186566              ADD   UPTR1
621 2D76 8566                STA   UPTR1
622 2D78 9002                BCC   NF2
623 2D7A E667                INC   UPTR1+1
624 2D7C A68E        NF2     LDX   OPSTAT
625 2D7E 4CF42E              JMP   EOSC5
626                  *
627                  * ASCII FIELD SCAN. VALUE RETURNED IN
628                  * UT4 IS NUMBER OF CHARACTERS IN STRING
629                  *
630 2D81 A5AB        FASC    LDA   NTERMS
631 2D83 D067                BNZ   UNDFTP
632 2D85 A902                LDA   #2
633 2D87 858E                STA   OPSTAT
634 2D89 E666D002            IDB   UPTR1
635 2D8F A566                LDA   UPTR1     SAVE STRING POINTER
636 2D91 85AC                STA   ASCPTR
637 2D93 A567                LDA   UPTR1+1
638 2D95 85AD                STA   ASCPTR+1
639 2D97 B166        FASC1   LDA   *UPTR1,Y
640 2D99 C9A7                CMP   #$A7      APOSTROPHE?
641 2D9B F007                BEQ   FASC2
642 2D9D C98D                CMP   #$8D      C.R.?
643 2D9F F01F                BEQ   FASC4
644 2DA1 C8                  INY
645 2DA2 D0F3                BNZ   FASC1
646 2DA4 E666D002    FASC2   IDB   UPTR1     GET PAST APOSTROPHE
647 2DAA B166        FASC3   LDA   *UPTR1,Y  EXAMINE TERMINATOR
648 2DAC 858D                STA   TMN8R
649 2DAE C9A0                CMP   #' '      MUST BE BLANK
650 2DB0 F00E                BEQ   FASC4
651 2DB2 C9AC                CMP   #','      OR COMMA
652 2DB4 F004                BEQ   FASC3A
653 2DB6 C98D                CMP   #$8D      OR C.R.
```

```
654 2DB8 D032              BNE   UNDFTP
655 2DBA A58E       FASC3A LDA   OPSTAT
656 2DBC 0904              ORA   #4          MARK COMMA
657 2DBE 858E              STA   OPSTAT
658 2DC0 C000       FASC4  CPY   #0
659 2DC2 F08B              BEQ   NOFLO
660 2DC4 84B8              STY   ALNTH
661 2DC6 C001              CPY   #1          SUITABLE IMM.?
662 2DC8 D011              BNE   FASC4A
663 2DCA 88                DEY
664 2DCB B1AC              LDA   *ASCPTR,Y FILL IN IMM.
665 2DCD 85B9              STA   IMVAL
666 2DCF 8582              STA   UT4
667 2DD1 8483              STY   UT4+1
668 2DD3 A58E              LDA   OPSTAT
669 2DD5 0910              ORA   #$10
670 2DD7 858E              STA   OPSTAT
671 2DD9 D00E              BNZ   FASC5
672 2DDB C002       FASC4A CPY   #2
673 2DDD D00A              BNE   FASC5
674 2DDF 88                DEY
675 2DE0 B1AC              LDA   *ASCPTR,Y
676 2DE2 8582              STA   UT4
677 2DE4 88                DEY
678 2DE5 B1AC              LDA   *ASCPTR,Y
679 2DE7 8583              STA   UT4+1
680 2DE9 A58E       FASC5  LDA   OPSTAT
681 2DEB 60                RTS
682 2DEC 4C7E2E     UNDFTP JMP   UNDFT
683                 *
684                 * NUMERIC SHIFT
685                 *
686 2DEF 0685       TSHF   ASL   TVAL
687 2DF1 2686              ROL   TVAL+1
688 2DF3 60                RTS
689                 *
690                 * NON-DECIMAL NUMERIC FIELD PROCESSOR
691                 *
692 2DF4 A901       FBIN   LDA   #1
693 2DF6 85A7              STA   SCOUNT      SET SHIFT COUNT
694 2DF8 A9B0              LDA   #'0'        AND COMPARE LIMITS
695 2DFA 8587              STA   FLO2
696 2DFC A9B2              LDA   #'2'
697 2DFE 8588       FNUM1  STA   FHI1
698 2E00 8589       FNUM1A STA   FHI2
699 2E02 E666D002           IDB   UPTR1
700 2E08 B166       FNUM3  LDA   *UPTR1,Y
701 2E0A C9B0              CMP   #'0'
702 2E0C 9022              BLS   EONUM
703 2E0E C588              CMP   FHI1
704 2E10 900B              BLS   FNUM4
705 2E12 C587              CMP   FLO2
706 2E14 901A              BLS   EONUM
707 2E16 C589              CMP   FHI2
708 2E18 B016              BGE   EONUM
709 2E1A 38E907            SUB   #7
710 2E1D 290F       FNUM4  AND   #$F         CLEAR GARBAGE BITS
711 2E1F A6A7              LDX   SCOUNT      READY TO SHIFT
712 2E21 20EF2D     FNUM5  JSR   TSHF
```

```
713 2E24 B01B                BCS   FNOFLO
714 2E26 CA                  DEX
715 2E27 D0F8                BNZ   FNUM5
716 2E29 0585                ORA   TVAL
717 2E2B 8585                STA   TVAL
718 2E2D C8                  INY
719 2E2E D0D8                BNZ   FNUM3
720 2E30 858D        EONUM   STA   TMN8R
721 2E32 A586                LDA   TVAL+1
722 2E34 D007                BNZ   EONUM3
723 2E36 A940                LDA   #$40
724 2E38 858E        EONUM2  STA   OPSTAT
725 2E3A 4CD82C              JMP   EOTERM
726 2E3D A920        EONUM3  LDA   #$20
727 2E3F D0F7                BNZ   EONUM2
728 2E41 4C4F2D      FNOFLO  JMP   NOFLO
729                  *
730                  * OCTAL FIELD
731                  *
732 2E44 A903        FOCT    LDA   #3          SET SHIFT COUNT
733 2E46 85A7                STA   SCOUNT
734 2E48 A9B0                LDA   #'0'        AND COMPARE LIMITS
735 2E4A 8587                STA   FLO2
736 2E4C A9B8                LDA   #'8'
737 2E4E D0AE                BNZ   FNUM1
738                  *
739                  * HEXADECIMAL FIELD
740                  *
741 2E50 A904        FHXDC   LDA   #4          SET SHIFT COUNT
742 2E52 85A7                STA   SCOUNT
743 2E54 A9C1                LDA   #'A'        AND COMPARE LIMITS
744 2E56 8587                STA   FLO2
745 2E58 A9BA                LDA   #$BA
746 2E5A 8588                STA   FHI1
747 2E5C A9C7                LDA   #'G'
748 2E5E D0A0                BNZ   FNUM1A
749                  *
750                  * SYMBOLIC FIELD
751                  *
752 2E60 208D2B      FSYM    JSR   SRCSTB
753 2E63 9019                BCC   UNDFT       CARRY OFF FOR UNDEFINED
754 2E65 1006                BPL   FSYM2
755 2E67 A58E                LDA   OPSTAT
756 2E69 0920                ORA   #$20        GEN. MEM. SYMBOL
757 2E6B 858E                STA   OPSTAT
758 2E6D A57C        FSYM2   LDA   UT1
759 2E6F 8585                STA   TVAL
760 2E71 A57D                LDA   UT1+1
761 2E73 8586                STA   TVAL+1
762 2E75 A000                LDY   #0
763 2E77 B166                LDA   *UPTR1,Y
764 2E79 858D                STA   TMN8R
765 2E7B 4CD82C      EOFSYM  JMP   EOTERM
766                  *
767                  * UNDEFINED TERM
768                  *
769 2E7E A9D5        UNDFT   LDA   #'U'
770 2E80 4C512D              JMP   NOFLO1
771 2E83 A900        HSH     LDA   #0
```

```
772 2E85 858A              STA   LMASK
773 2E87 A9FF              LDA   #$FF
774 2E89 858B       HSH1   STA   HMASK
775 2E8B E6AB              INC   NTERMS
776 2E8D E666D002          IDB   UPTR1
777 2E93 A58E              LDA   OPSTAT
778 2E95 0901              ORA   #1
779 2E97 858E              STA   OPSTAT
780 2E99 4C702C            JMP   OPSC3
781 2E9C A9FF       HSL    LDA   #$FF
782 2E9E 858A              STA   LMASK
783 2EA0 A900              LDA   #0
784 2EA2 F0E5              BZ    HSH1
785                 *
786                 * END OF SCAN
787                 *
788 2EA4 A58A       EOSCAN LDA   LMASK      CHECK HALF SYMBOL
789 2EA6 258B              AND   HMASK
790 2EA8 D00C              BNZ   EOSC2
791 2EAA A58B              LDA   HMASK
792 2EAC F006              BZ    EOSC1
793 2EAE A583              LDA   UT4+1
794 2EB0 8582              STA   UT4
795 2EB2 A900              CLA
796 2EB4 8583       EOSC1  STA   UT4+1
797 2EB6 A000       EOSC2  LDY   #0
798 2EB8 B166              LDA   *UPTR1,Y  GET TERMINATOR
799 2EBA 858D              STA   TMN8R
800 2EBC A68E              LDX   OPSTAT
801 2EBE 8A                TXA
802 2EBF 3033              BMI   EOSC5
803 2EC1 8A         EOSC2A TXA             CHECK FOR HALF SYMBOL
804 2EC2 6A                RORA
805 2EC3 B008              BCS   MLTRL
806 2EC5 A583              LDA   UT4+1      CONSTANT < 256?
807 2EC7 F004              BZ    MLTRL
808 2EC9 C9FF              CMP   #$FF
809 2ECB D008              BNE   EOSC3
810 2ECD 8A         MLTRL  TXA
811 2ECE 0910              ORA   #$10
812 2ED0 AA                TAX
813 2ED1 A582              LDA   UT4
814 2ED3 85B9              STA   IMVAL
815 2ED5 8A         EOSC3  TXA             GEN. MEM. SYMBOL,?
816 2ED6 2920              AND   #$20
817 2ED8 D010              BNZ   EOSC4
818 2EDA A583              LDA   UT4+1
819 2EDC D006              BNZ   EOSC3A
820 2EDE 8A                TXA
821 2EDF 0940              ORA   #$40       MARK SCRATCHPAD
822 2EE1 AA                TAX
823 2EE2 D006              BNZ   EOSC4
824 2EE4 8A         EOSC3A TXA
825 2EE5 29BF              AND   #$BF
826 2EE7 0920              ORA   #$20
827 2EE9 AA                TAX
828 2EEA A58D       EOSC4  LDA   TMN8R
829 2EEC C9AC              CMP   #','
830 2EEE D004              BNE   EOSC5
```

```
2EF0 8A                 TXA
2EF1 0904               ORA   #4
2EF3 AA                 TAX
2EF4 E666D002   EOSC5   IDB   UPTR1
2EFA 868E       EOSC6   STX   OPSTAT
2EFC 8A                 TXA
2EFD 60                 RTS             AND RETURN
                *
                * ASSEMBLY MONITOR
                *
2EFE A900       ASMBL   CLA
2F00 85A8               STA   PAGE      CLEAR PAGE #
2F02 85A9               STA   LFLAG     LIST FLAG
2F04 85AA               STA   OBFLAG    OBJECT FLAG
2F06 8591               STA   FIRST     PRINT LIMIT FLAGS
2F08 8592               STA   FIRST+1
2F0A 8593               STA   SECOND
2F0C 8594               STA   SECOND+1
2F0E 20EA20             JSR   RKB       SEE WHAT HE WANTS
2F11 C9CC               CMP   #'L'      LISTING PASS?
2F13 F018               BEQ   STLIST
2F15 48                 PHA             SAVE CHAR
2F16 20EA20             JSR   RKB       GET TERMINATOR
2F19 C9AE               CMP   #'.'
2F1B D00D               BNE   ASMERR    MUST BE PERIOD
2F1D 68                 PLA             RETRIEVE CHAR
2F1E C9C5               CMP   #'E'      ERROR PASS
2F20 F03B               BEQ   STL4
2F22 C9CF               CMP   #'O'      OBJECT PASS
2F24 D004               BNE   ASMERR    GARBAGE
2F26 85AA               STA   OBFLAG
2F28 F033               BEQ   STL4
2F2A 4C4F20     ASMERR  JMP   ERR
2F2D 20B921     STLIST  JSR   FPRM      GET PRINT LIMITS
2F30 A592               LDA   FIRST+1   FORM NEGATIVE
2F32 49FF               CMA
2F34 AA                 TAX
2F35 A591               LDA   FIRST
2F37 49FF1869           TCA
2F3C D001               BNZ   STL2
2F3E E8                 INX
2F3F 8591       STL2    STA   FIRST
2F41 8692               STX   FIRST+1
2F43 E691D002           IDB   FIRST
2F49 D002               BNZ   STL2A
2F4B E6A9               INC   LFLAG
2F4D 8591       STL2A   STA   FIRST
2F4F 8692               STX   FIRST+1
2F51 A593               LDA   SECOND    ONES COMPLEMENT
2F53 49FF               CMA
2F55 8593               STA   SECOND    FOR INCLUSIVE
2F57 A594               LDA   SECOND+1
2F59 49FF               CMA             DIFFERENCE
2F5B 8594               STA   SECOND+1
2F5D A578       STL4    LDA   LNCNT     CHECK FOR NO PROGRAM
2F5F 0579               ORA   LNCNT+1
2F61 F044               BZ    NOPROG
2F63 A578               LDA   LNCNT
2F65 A679               LDX   LNCNT+1   CHECK FOR "END"
```

```
2F67 203322            JSR   FBA       GET ADDRESS OF LAST LINE
2F6A 856E              STA   UPTR5
2F6C 866F              STX   UPTR5+1
2F6E 201227            JSR   FORMAT    GET LAST LINE
2F71 D03E              BNZ   NOEND
2F73 A58F              LDA   SKEL
2F75 C902              CMP   #2
2F77 D038              BNE   NOEND
2F79 A57A              LDA   NXTCHR    SET SYMBOL TABLE
2F7B 85A5              STA   SYMTAB    ORIGIN
2F7D A57B              LDA   NXTCHR+1
2F7F 85A6              STA   SYMTAB+1
2F81 20962F            JSR   STSPTR    SET SOURCE POINTER FOR PASS
2F84 20CC2F            JSR   PASS1     EXECUTE PASS 1
2F87 20962F            JSR   STSPTR    DITTO
2F8A 203832            JSR   PASS2     LIKEWISE
2F8D 20962F            JSR   STSPTR    DITTO
2F90 206234            JSR   PASS3     LIKEWISE
2F93 4C2520            JMP   MON1
2F96 ADC520  STSPTR    LDA   BFR       SET SOURCE POINTER
2F99 856E              STA   UPTR5
2F9B ADC620            LDA   BFR+1
2F9E 856F              STA   UPTR5+1
2FA0 A900              CLA
2FA2 85AE              STA   SRCLNO
2FA4 85AF              STA   SRCLNO+1
2FA6 60                RTS
2FA7 A9B8    NOPROG    LDA   #<NPMSG
2FA9 A22F              LDX   #>NPMSG
2FAB 20D420  NP2       JSR   DISP
2FAE 4C2520            JMP   MON1
2FB1 A9C3    NOEND     LDA   #<NEMSG
2FB3 A22F              LDX   #>NEMSG
2FB5 4CAB2F            JMP   NP2
2FB8 CECFA0D0  NPMSG   ASC   'NO PROGRAM'
2FC2 8D                DATA  $8D
2FC3 CECFA0A2  NEMSG   ASC   'NO "END"'
2FCB 8D                DATA  $8D
                       *
                       * ASSEMBLER PASS 1
                       * THIS PASS DEFINES ALL SCRATCHPAD SYMBOLS
                       *
2FCC A900      PASS1   CLA             CLEAR FLAGS & INDICATORS
2FCE 85B0              STA   SPLC      LOCATION COUNTER
2FD0 859F              STA   NSYMS     SYMBOL COUNT
2FD2 85A0              STA   NSYMS+1
2FD4 85B1              STA   TLFLAG    TITLE FLAG
2FD6 85B5              STA   P1ERFL    PASS1 ERROR FLAG
2FD8 85B2              STA   SPDFLG    SCRATCHPAD FLAG
2FDA 857D              STA   UT1+1     HIGH RESULT
2FDC A98D              LDA   #$8D
2FDE 8DC831            STA   TITLE
2FE1 201227    P1B     JSR   FORMAT    GET SOURCE RECORD
2FE4 A934856A          CDB   UPTR3,LNO-2 PTR FOR LINE # CONVERT
2FEC E6AED002          IDB   SRCLNO
2FF2 A5AE              LDA   SRCLNO
2FF4 A6AF              LDX   SRCLNO+1
2FF6 20E724            JSR   CNVLNO    AND CONVERT IT
2FF9 A900              LDA   #0
```

```
2FFB 85B3            STA   LENGTH     CLEAR LENGTH
2FFD 857D            STA   UT1+1
2FFF A5B0            LDA   SPLC       SET TENTATIVE LABEL VLUE
3001 857C            STA   UT1
3003 AD4A2A          LDA   LABEL      COMMENT LINE?
3006 C9AA            CMP   #'*'
3008 F0D7            BEQ   P1B        YES, SKIP IT
300A A590            LDA   CLASS      CHECK CLASS
300C 303B            BMI   P1JUNK     GARBAGE
300E F006            BZ    P1D        SKIP IF PSEUDO-OP
3010 A900      P1C   CLA
3012 85B2            STA   SPDFLG     CLEAR SPD FLAG
3014 F0CB            BZ    P1B        AND GET NEXT LINE
3016 A58F      P1D   LDA   SKEL       GET INST. SKELETON
3018 0A              ASLA             TIMES 2
3019 AA              TAX              MOVE TO INDEX
301A E8              INX              BUMP
301B BD2F30          LDA   P1JTBL,X   GET HIGH JUMP ADDRESS
301E 48              PHA              SAVE ON STACK
301F CA              DEX
3020 BD2F30          LDA   P1JTBL,X   GET LOW JUMP ADDRESS
3023 48              PHA
3024 A9578566        CDB   UPTR1,OPND SET OPERAND SCAN POINTER
302C A5B2            LDA   SPDFLG
302E 60              RTS              POP JUMP ADDRESS TO P.C.

                                      PSEUDO-OP JUMP TABLE

302F 6A307030  P1JTBL DBL   P1ORG-1,P1SPD-1
3033 88309730        DBL   P1END-1,P1TTL-1
3037 AE30C230        DBL   P1EQU-1,P1REM-1
303B D630E730        DBL   P1RES-1,P1DTA-1
303F F530FD30        DBL   P1DBL-1,P1ASC-1
3043 1D318230        DBL   P1AST-1,P1ESP-1,P1$-1
               *
3049 A904      P1JUNK LDA  #4
304B 85B3            STA   LENGTH     DEFAULT LENGTH = 4
304D A5B3      P1SUM LDA   LENGTH
304F 1865B0          ADD   SPLC       FORM NEW L.C.
3052 85B0            STA   SPLC       SAVE
3054 908B            BCC   P1B        DID IT OVERFLOW?
3056 A9FF            LDA   #$FF       YES
3058 85B0            STA   SPLC
305A A936            LDA   #<LNO
305C A22A            LDX   #>LNO      PRINT LINE
305E 20B020          JSR   PRT
3061 A948            LDA   #<SPOMS    AND OVERFLOW MESSAGE
3063 A231            LDX   #>SPOMS
3065 20B020          JSR   PRT
3068 4CE12F          JMP   P1B
               *
                                      ORG
306B A900      P1ORG LDA   #0
306D 85B2            STA   SPDFLG
306F F00F            BZ    JP1B
                                      SPD
3071 A901      P1SPD LDA   #1
3073 85B2            STA   SPDFLG     SET SPD FLAG
3075 205A2C          JSR   OPDSCN
```

```
008 3078 2940              AND   #$40
009 307A F004              BZ    JP1B
010 307C A582              LDA   UT4
011 307E 85B0              STA   SPLC
012 3080 4CE12F    JP1B    JMP   P1B
013                        *
014 3083 A900      P1ESP   LDA   #0
015 3085 85B2              STA   SPDFLG
016 3087 F0F7              BZ    JP1B
017                        *
018 3089 A5B5      P1END   LDA   P1ERFL
019 308B D001              BNZ   P1END2
020 308D 60                RTS
021 308E A911      P1END2  LDA   #<TEMS
022 3090 A232              LDX   #>TEMS
023 3092 20D420            JSR   DISP
024 3095 4C2520            JMP   MON1
025                        *
026 3098 A5B1      P1TTL   LDA   TLFLAG
027 309A D0E4              BNZ   JP1B
028 309C A229              LDX   #41
029 309E A98D              LDA   #$8D
030 30A0 D003              BNZ   P1TTL2
031 30A2 BD572A    P1TTL1  LDA   OPND,X
032 30A5 9DC831    P1TTL2  STA   TITLE,X
033 30A8 CA                DEX
034 30A9 10F7              BPL   P1TTL1
035 30AB 86B1              STX   TLFLAG
036 30AD D0D1              BNZ   JP1B
037                        *
038 30AF F0CF      P1EQU   BZ    JP1B
039 30B1 205A2C            JSR   OPDSCN
040 30B4 3010              BMI   P1ERR
041 30B6 A582              LDA   UT4
042 30B8 857C              STA   UT1
043 30BA A583              LDA   UT4+1
044 30BC 857D              STA   UT1+1
045 30BE 203031            JSR   P1CKL
046 30C1 B003              BCS   P1ERR
047 30C3 4CE12F    P1REM   JMP   P1B
048 30C6 A936      P1ERR   LDA   #<LNO
049 30C8 A22A              LDX   #>LNO
050 30CA 20D420            JSR   DISP
051 30CD A9F2              LDA   #<ONPD
052 30CF A231              LDX   #>ONPD
053 30D1 20D420            JSR   DISP
054 30D4 4CE12F            JMP   P1B
055                        *
056 30D7 F0EA      P1RES   BZ    P1REM
057 30D9 205A2C            JSR   OPDSCN
058 30DC 30E8              BMI   P1ERR
059 30DE A582              LDA   UT4
060 30E0 85B3              STA   LENGTH
061 30E2 203031            JSR   P1CKL
062 30E5 4C4D30            JMP   P1SUM
063                        *
064 30E8 F0D9      P1DTA   BZ    P1REM
065 30EA E680              INC   UT3
066 30EC A580      P1DTA2  LDA   UT3
```

```
067 30EE 85B3               STA   LENGTH
068 30F0 203031             JSR   P1CKL
069 30F3 4C4D30             JMP   P1SUM
070                 *
071 30F6 F0CB       P1DBL   BZ    P1REM
072 30F8 E680               INC   UT3
073 30FA 0680               ASL   UT3
074 30FC D0EE               BNZ   P1DTA2
075                 *
076 30FE F0C3       P1ASC   BZ    P1REM
077 3100 203031             JSR   P1CKL
078 3103 A9578566           CDB   UPTR1,OPND
079 310B 205A2C             JSR   OPDSCN
080 310E 1003               BPL   P1ASC3
081 3110 4C4930     P1ASC2  JMP   P1JUNK
082 3113 2902       P1ASC3  AND   #2
083 3115 F0F9               BZ    P1ASC2
084 3117 A5B8               LDA   ALNTH
085 3119 85B3       P1ASC4  STA   LENGTH
086 311B 4C4D30             JMP   P1SUM
087                 *
088 311E F0A3       P1AST   BZ    P1REM
089 3120 203031             JSR   P1CKL
090 3123 A901               LDA   #1
091 3125 D0F2               BNZ   P1ASC4
092                 *
093 3127 F09A       P1$     BZ    P1REM
094 3129 203031             JSR   P1CKL
095 312C A902               LDA   #2
096 312E D0E9               BNZ   P1ASC4
097                 *
098 3130 205C31     P1CKL   JSR   CKLBL
099 3133 D001               BNZ   P1CKL1
100 3135 60                 RTS               LABEL BLANK OR ILLEGAL
101 3136 9009       P1CKL1  BCC   P1CKL3
102 3138 A001               LDY   #1          LABEL IS MULTIPLY
103 313A B1A3       P1CKL2  LDA   *SYMPTR,Y   DEFINED
104 313C 297F               AND   #$7F        MARK IT
105 313E 91A3               STA   *SYMPTR,Y
106 3140 60                 RTS
107 3141 20B22A     P1CKL3  JSR   ESYMTB      OK, ENTER SYMBOL
108 3144 A000               LDY   #0          INTO TABLE, MARK AS
109 3146 F0F2               BZ    P1CKL2      SCRATCHPAD
110                 *
111 3148 D3C3D2C1   SPOMS   ASC   'SCRATCHPAD OVERFLOW'
112 315B 8D                 DATA  $8D
113                 *
114 315C A980       CKLBL   LDA   #$80
115 315E 8DB831             STA   MDW
116 3161 AD4A2A             LDA   LABEL
117 3164 C9A0               CMP   #' '
118 3166 D002               BNE   CKL2
119 3168 B8                 CLV
120 3169 60                 RTS               NO LABEL
121 316A C9C1       CKL2    CMP   #'A'        1ST CHAR ALPHABETIC?
122 316C 9043               BLS   ILBL
123 316E C9DB               CMP   #'['
124 3170 B03F               BGE   ILBL
125 3172 A200               LDX   #0
```

```
126 3174 BD4B2A      CKL3    LDA   LABEL+1,X
127 3177 C9A0                CMP   #' '        END OF LABEL?
128 3179 F021                BEQ   EOCK
129 317B C9AB                CMP   #'+'
130 317D F032                BEQ   ILBL
131 317F C9AD                CMP   #'-'
132 3181 F02E                BEQ   ILBL
133 3183 C9AA                CMP   #'*'
134 3185 F02A                BEQ   ILBL
135 3187 C9AF                CMP   #'/'
136 3189 F026                BEQ   ILBL
137 318B C9A8                CMP   #'('
138 318D F022                BEQ   ILBL
139 318F C9A9                CMP   #')'
140 3191 F01E                BEQ   ILBL
141 3193 C9AC                CMP   #','
142 3195 F01A                BEQ   ILBL
143 3197 E8                  INX
144 3198 E005                CPX   #5
145 319A D0D8                BNE   CKL3
146 319C A94A8566    EOCK    CDB   UPTR1,LABEL
147 31A4 208D2B              JSR   SRCSTB
148 31A7 A001                LDY   #1
149 31A9 B1A3                LDA   *SYMPTR,Y
150 31AB 8DB831              STA   MDW
151 31AE A901                LDA   #1
152 31B0 60                  RTS                RETURN WITH CARRY 1/0
153 31B1 A980        ILBL    LDA   #$80         BUM LABEL, SET ZERO
154 31B3 186980              ADD   #$80         AND OVERFLOW
155 31B6 18                  CLC
156 31B7 60                  RTS
157 31B8 00          MDW     ***
158 31B9 D0C1C7C5    HDNG    ASC   'PAGE '
159 31BE             DPGNO   RES   3
160 31C1 A0A0A0A0            ASC   '       '
161 31C8             TITLE   RES   42
162 31F2 CFD0C5D2    ONPD    ASC   'OPERAND NOT PREVIOUSLY DEFINED'
163 3210 8D                  DATA  $8D
164 3211 D4C5D2CD    TEMS    ASC   'TERMINAL PASS 1 ERROR, '
165 3228 C3C1CECE            ASC   'CANNOT CONTINUE'
166 3237 8D                  DATA  $8D
167                  *
168                  * ASSEMBLER PASS2. THIS PASS DEFINES ALL NON
169                  * SCRATCHPAD SYMBOLS.
170                  *
171 3238 A900        PASS2   CLA
172 323A 85B2                STA   SPDFLG       CLEAR SCRATCHPAD FLAG
173 323C 85B6                STA   ILC          SET L.C. TO $300
174 323E A903                LDA   #3           AS DEFAULT LOCATION
175 3240 85B7                STA   ILC+1
176 3242 201227      P2B     JSR   FORMAT       GET SOURCE RECORD
177 3245 A934856A            CDB   UPTR3,LNO-2 PTR FOR LINE # CONVERT
178 324D E6AED002            IDB   SRCLNO
179 3253 A5AE                LDA   SRCLNO
180 3255 A6AF                LDX   SRCLNO+1
181 3257 20E724              JSR   CNVLNO
182 325A A900                LDA   #0
183 325C 85B3                STA   LENGTH       CLEAR LENGTH
184 325E 85B4                STA   LENGTH+1
```

```
185 3260 A5B6               LDA   ILC        SET NOMINAL LABEL
186 3262 857C               STA   UT1        TO L.C.
187 3264 A5B7               LDA   ILC+1
188 3266 857D               STA   UT1+1
189 3268 AD4A2A             LDA   LABEL      CHECK FOR COMMENT
190 326B C9AA               CMP   #'*'
191 326D F0D3               BEQ   P2B
192 326F A590               LDA   CLASS      CHECK CLASS
193 3271 3065               BMI   P2JUNK
194 3273 D00D               BNZ   P2C        NOT A PSEUDO-OP
195 3275 A9A48566           CDB   UPTR1,P2PSTB SET PTR TO PSEUDO
196 327D A58F               LDA   SKEL       OP TABLE
197 327F 4C8F32             JMP   P2D
198 3282 A9BE8566   P2C     CDB   UPTR1,P2JTBL SET PTR TO INST.
199 328A A590               LDA   CLASS      JUMP TABLE
200 328C 1869FF             DEA
201 328F 0A         P2D     ASLA
202 3290 A8                 TAY
203 3291 C8                 INY
204 3292 B166               LDA   *UPTR1,Y
205 3294 48                 PHA              SAVE ADDRESS ON STACK
206 3295 88                 DEY
207 3296 B166               LDA   *UPTR1,Y
208 3298 48                 PHA
209 3299 A9578566           CDB   UPTR1,OPND SET PTR TO OPND FIELD
210 32A1 A5B2               LDA   SPDFLG
211 32A3 60                 RTS              JUMP TO PROCESSOR
212                 *
213                 * PSEUDO-OP JUMP TABLE
214                 *
215 32A4 EF320533   P2PSTB  DBL   P2ORG-1,P2SPD-1
216 32A8 0B331933           DBL   P2END-1,P2TITL-1
217 32AC 30334433           DBL   P2EQU-1,P2REM-1
218 32B0 5B337033           DBL   P2RES-1,P2DTA-1
219 32B4 7E338633           DBL   P2DBL-1,P2ASC-1
220 32B8 A6331333           DBL   P2AST-1,P2ESP-1
221 32BC AF33               DBL   P2$-1
222                 *
223                 * INSTRUCTION JUMP TABLE
224                 *
225 32BE CD33F933   P2JTBL  DBL   P2C1-1,P2C2-1
226 32C2 01340934           DBL   P2C3-1,P2C4-1
227 32C6 F9330F34           DBL   P2C5-1,P2C6-1
228 32CA 1A341A34           DBL   P2C7-1,P2C8-1
229 32CE 0F340F34           DBL   P2C9-1,P2CA-1
230 32D2 F933F933           DBL   P2CB-1,P2CC-1
231 32D6 2234               DBL   P2CD-1
232                 *
233 32D8 A5B2       P2JUNK  LDA   SPDFLG
234 32DA D027               BNZ   JP2B
235 32DC A904               LDA   #4
236 32DE 85B3               STA   LENGTH
237 32E0 A5B3       P2SUM   LDA   LENGTH
238 32E2 1865B6             ADD   ILC
239 32E5 85B6               STA   ILC
240 32E7 A5B4               LDA   LENGTH+1
241 32E9 65B7               ADC   ILC+1
242 32EB 85B7               STA   ILC+1
243 32ED 4C4232             JMP   P2B
```

```
244                     *
245 32F0 A900           P2ORG   LDA   #0          CLEAR SPD FLAG
246 32F2 85B2                   STA   SPDFLG
247 32F4 205A2C                 JSR   OPDSCN
248 32F7 2960                   AND   #$60
249 32F9 F04D                   BZ    P2ERR
250 32FB A582                   LDA   UT4         SET NEW ORIGIN
251 32FD 85B6                   STA   ILC
252 32FF A583                   LDA   UT4+1
253 3301 85B7                   STA   ILC+1
254 3303 4C4232         JP2B    JMP   P2B
255                     *
256 3306 A901           P2SPD   LDA   #1
257 3308 85B2                   STA   SPDFLG
258 330A D0F7                   BNZ   JP2B
259                     *
260 330C A5B5           P2END   LDA   P1ERFL
261 330E D001                   BNZ   P2END2
262 3310 60                     RTS
263 3311 4C8E30         P2END2  JMP   P1END2
264                     *
265 3314 A900           P2ESP   LDA   #0
266 3316 85B2                   STA   SPDFLG
267 3318 F0E9                   BZ    JP2B
268                     *
269 331A A5B1           P2TITL  LDA   TLFLAG
270 331C D0E5                   BNZ   JP2B
271 331E A229                   LDX   #41
272 3320 A98D                   LDA   #$8D
273 3322 D003                   BNZ   P2TTL2
274 3324 BD572A         P2TTL1  LDA   OPND,X
275 3327 9DC831         P2TTL2  STA   TITLE,X
276 332A CA                     DEX
277 332B 10F7                   BPL   P2TTL1
278 332D 86B1                   STX   TLFLAG
279 332F D0D2                   BNZ   JP2B
280                     *
281 3331 D0D0           P2EQU   BNZ   JP2B
282 3333 205A2C                 JSR   OPDSCN
283 3336 3010                   BMI   P2ERR
284 3338 A582                   LDA   UT4
285 333A 857C                   STA   UT1
286 333C A583                   LDA   UT4+1
287 333E 857D                   STA   UT1+1
288 3340 20B933                 JSR   P2CKL
289 3343 B003                   BCS   P2ERR
290 3345 4C4232         P2REM   JMP   P2B
291 3348 A936           P2ERR   LDA   #<LNO
292 334A A22A                   LDX   #>LNO
293 334C 20D420                 JSR   DISP
294 334F A9F2                   LDA   #<ONPD
295 3351 A231                   LDX   #>ONPD
296 3353 20D420                 JSR   DISP
297 3356 A901                   LDA   #1
298 3358 85B5                   STA   P1ERFL
299 335A D0E9                   BNZ   P2REM
300                     *
301 335C D0E7           P2RES   BNZ   P2REM
302 335E 205A2C                 JSR   OPDSCN
```

```
303 3361 30E5               BMI   P2ERR
304 3363 A582               LDA   UT4
305 3365 85B3               STA   LENGTH
306 3367 A583               LDA   UT4+1
307 3369 85B4               STA   LENGTH+1
308 336B 20B933             JSR   P2CKL
309 336E 4CE032             JMP   P2SUM
310                 *
311 3371 D0D2       P2DTA   BNZ   P2REM
312 3373 E680               INC   UT3
313 3375 A580       P2DTA2  LDA   UT3
314 3377 85B3               STA   LENGTH
315 3379 20B933             JSR   P2CKL
316 337C 4CE032             JMP   P2SUM
317                 *
318 337F D0C4       P2DBL   BNZ   P2REM
319 3381 E680               INC   UT3
320 3383 0680               ASL   UT3
321 3385 D0EE               BNZ   P2DTA2
322                 *
323 3387 D0BC       P2ASC   BNZ   P2REM
324 3389 20B933             JSR   P2CKL
325 338C A9578566           CDB   UPTR1,OPND
326 3394 205A2C             JSR   OPDSCN
327 3397 1003               BPL   P2ASC3
328 3399 4CD832     P2ASC2  JMP   P2JUNK
329 339C 2902       P2ASC3  AND   #2
330 339E F0F9               BZ    P2ASC2
331 33A0 A5B8               LDA   ALNTH
332 33A2 85B3       P2ASC4  STA   LENGTH
333 33A4 4CE032             JMP   P2SUM
334                 *
335 33A7 D09C       P2AST   BNZ   P2REM
336 33A9 20B933             JSR   P2CKL
337 33AC A901               LDA   #1
338 33AE D0F2               BNZ   P2ASC4
339                 *
340 33B0 D0F5       P2$     BNZ   P2AST
341 33B2 20B933             JSR   P2CKL
342 33B5 A902               LDA   #2
343 33B7 D0E9               BNZ   P2ASC4
344 33B9 205C31     P2CKL   JSR   CKLBL
345 33BC D001               BNZ   P2CKL1
346 33BE 60                 RTS
347 33BF 9009       P2CKL1  BCC   P2CKL2
348 33C1 A001               LDY   #1
349 33C3 B1A3               LDA   *SYMPTR,Y
350 33C5 297F               AND   #$7F
351 33C7 91A3               STA   *SYMPTR,Y
352 33C9 60                 RTS
353 33CA 20B22A     P2CKL2  JSR   ESYMTB
354 33CD 60                 RTS
355                 *
356                 * INSTRUCTION PROCESSORS
357                 *
358 33CE D0E0       P2C1    BNZ   P2$
359 33D0 A902               LDA   #2
360 33D2 85B3               STA   LENGTH
361 33D4 AD572A     P2C1S   LDA   OPND      CHK FOR IND. OR IMM.
```

```
362 33D7 C9A8              CMP   #'('
363 33D9 F011              BEQ   P2C1B
364 33DB C9AA              CMP   #'*'
365 33DD F00D              BEQ   P2C1B
366 33DF C9A3       P2C10  CMP   #'#'
367 33E1 F009              BEQ   P2C1B
368 33E3 205A2C     P2C1A  JSR   OPDSCN
369 33E6 2940              AND   #$40
370 33E8 D002              BNZ   P2C1B
371 33EA E6B3              INC   LENGTH
372 33EC A5B6       P2C1B  LDA   ILC
373 33EE 857C              STA   UT1
374 33F0 A5B7              LDA   ILC+1
375 33F2 857D              STA   UT1+1
376 33F4 20B933            JSR   P2CKL
377 33F7 4CE032     P2C1C  JMP   P2SUM
378                 *
379 33FA D0D2       P2C2   BNZ   P2C1
380 33FC A902       P2C2A  LDA   #2
381 33FE 85B3              STA   LENGTH
382 3400 D0E1              BNZ   P2C1A
383                 *
384 3402 D0CA       P2C3   BNZ   P2C1
385 3404 A901              LDA   #1
386 3406 85B3       P2C3A  STA   LENGTH
387 3408 D0E2              BNZ   P2C1B
388                 *
389 340A D0C2       P2C4   BNZ   P2C1
390 340C A902              LDA   #2
391 340E D0F6              BNZ   P2C3A
392                 *
393                 P2C5   EQU   P2C2
394                 *
395 3410 D0BC       P2C6   BNZ   P2C1
396 3412 A902              LDA   #2
397 3414 85B3              STA   LENGTH
398 3416 AD572A            LDA   OPND
399 3419 D0C4              BNZ   P2C10
400                 *
401 341B D0B1       P2C7   BNZ   P2C1
402 341D A903              LDA   #3
403 341F 85B3              STA   LENGTH
404 3421 D0C9              BNZ   P2C1B
405                 *
406                 P2C8   EQU   P2C7
407                 *
408                 P2C9   EQU   P2C6
409                 *
410                 P2CA   EQU   P2C6
411                 *
412                 P2CB   EQU   P2C2
413                 *
414                 P2CC   EQU   P2C2
415                 *
416 3423 D0EB       P2CD   BNZ   P2C6
417 3425 A903              LDA   #3
418 3427 85B3              STA   LENGTH
419 3429 A58F              LDA   SKEL      SCAN FOR PSEUDO-OP
420 342B F017              BZ    P2CDB     DEA
```

```
421 342D C901               CMP   #1
422 342F F013               BEQ   P2CDB     INA
423 3431 C904               CMP   #4
424 3433 900D               BLS   P2CDA     CLA, CMA
425 3435 F010               BEQ   P2CD5     TCA
426 3437 C905               CMP   #5
427 3439 F012               BEQ   P2CDC     CDB
428 343B C906               CMP   #6
429 343D F01F               BEQ   P2CDD     IDB
430 343F 4CD433             JMP   P2C1S
431 3442 C6B3       P2CDA   DEC   LENGTH
432 3444 4CEC33     P2CDB   JMP   P2C1B
433 3447 A905       P2CD5   LDA   #5
434 3449 85B3               STA   LENGTH
435 344B D0F7               BNZ   P2CDB
436 344D A908       P2CDC   LDA   #8
437 344F 85B3       P2CDC1  STA   LENGTH
438 3451 205A2C             JSR   OPDSCN
439 3454 2940               AND   #$40
440 3456 D0EC               BNZ   P2CDB
441 3458 E6B3               INC   LENGTH
442 345A E6B3               INC   LENGTH
443 345C D0E6               BNZ   P2CDB
444 345E A906       P2CDD   LDA   #6
445 3460 D0ED               BNZ   P2CDC1
446                 *
447                 * ASSEMBLER PASS 3
448                 *
449 3462 A903       PASS3   LDA   #3        SET DEFAULT
450 3464 85B7               STA   ILC+1     LOCATION COUNTER
451 3466 8DE93E             STA   RCDORG+1
452 3469 A900               CLA
453 346B 85B6               STA   ILC
454 346D 8DE83E             STA   RCDORG
455 3470 85B0               STA   SPLC      AND SP LOC. CTR.
456 3472 85B2               STA   SPDFLG    CLEAR S.P. FLAG
457 3474 8D373D             STA   BPGNO
458 3477 A901               LDA   #1
459 3479 8D383D             STA   LPPCNT
460 347C A5BC               LDA   PAUSCT
461 347E 8D393D             STA   PSCT
462 3481 20CF3E             JSR   WLDR      WRITE OBJECT LEADER
463                                         START PAGINATION
464 3484 A978               LDA   #<BLKLIN
465 3486 A23F               LDX   #>BLKLIN
466 3488 20AC3C             JSR   LPNT
467 348B 20B620     P3A     JSR   DTCC      WANT TO STOP??
468 348E 9003               BCC   P3A0
469 3490 4C2520             JMP   MON1      YES
470 3493 201227     P3A0    JSR   FORMAT    GET SOURCE LINE
471 3496 A934856A           CDB   UPTR3,LNO-2 SET CONVERT PTR.
472 349E E6AED002           IDB   SRCLNO    BUMP LINE NUMBER
473 34A4 A5AE               LDA   SRCLNO
474 34A6 A6AF               LDX   SRCLNO+1
475 34A8 20E724             JSR   CNVLNO    CONVERT LINE #
476 34AB A900               CLA
477 34AD A026               LDY   #ERCDLN-OCODE-1
478 34AF 99B13D     P3A2    STA   OCODE,Y   CLEAR OBJECT AREA
479 34B2 88                 DEY
```

```
480 34B3 10FA               BPL   P3A2
481 34B5 A007               LDY   #7
482 34B7 A9A0               LDA   #' '
483 34B9 99DC3D      P3A3   STA   ECA,Y      CLEAR ERROR FLAGS
484 34BC 88                 DEY
485 34BD 10FA               BPL   P3A3
486 34BF A9B18576           CDB   OBJPTR,OCODE SET OBJ. PTR.
487 34C7 AD4A2A             LDA   LABEL      COMMENT?
488 34CA C9AA               CMP   #'*'
489 34CC D003               BNE   P3A4       YES, SKIP THE FOLDEROL
490 34CE 4C9D3A             JMP   P3PRT
491 34D1 205C31      P3A4   JSR   CKLBL      CHECK LABEL VALIDITY
492 34D4 700C               BVS   BMLBL
493 34D6 2EB831             ROL   MDW
494 34D9 B00C               BCS   P3A5
495 34DB A9CD               LDA   #'M'       MARK MULTIPLE
496 34DD 206036             JSR   STERCH
497 34E0 D005               BNZ   P3A5
498 34E2 A9D3        BMLBL  LDA   #'S'       MARK BAD SYMBOL
499 34E4 206036             JSR   STERCH
500 34E7 A590        P3A5   LDA   CLASS
501 34E9 1008               BPL   P3A6
502 34EB A9CF               LDA   #'O'       MARK BAD INST.
503 34ED 206036             JSR   STERCH
504 34F0 4C2835             JMP   P3JUNK
505 34F3 D00D        P3A6   BNZ   P3A7       SKIP IF INST.
506 34F5 A9308566           CDB   UPTR1,P3PSTB
507 34FD A58F               LDA   SKEL       SELECT PSEUDO-OP TABLE
508 34FF 4C1335             JMP   P3A8
509 3502 A94A8566    P3A7   CDB   UPTR1,P3JTBL
510 350A A900               LDA   #0
511 350C 85B2               STA   SPDFLG
512 350E A590               LDA   CLASS      SELECT INST. TABLE
513 3510 1869FF             DEA
514 3513 0A          P3A8   ASLA
515 3514 A8                 TAY
516 3515 C8                 INY
517 3516 B166               LDA   *UPTR1,Y
518 3518 48                 PHA              GET ADDRESS FROM TABLE
519 3519 88                 DEY
520 351A B166               LDA   *UPTR1,Y
521 351C 48                 PHA
522 351D A9578566           CDB   UPTR1,OPND SET PTR. FOR SCAN
523 3525 A5B2               LDA   SPDFLG
524 3527 60                 RTS              JUMP TO SERVICE ROUTINE
525 3528 A904        P3JUNK LDA   #4
526 352A 8DD43D             STA   P3LNTH
527 352D 4C9D3A             JMP   P3PRT
528                  *
529                  * PSEUDO-OP TABLE
530                  *
531 3530 63357E35    P3PSTB DBL   P3ORG-1,P3SPD-1
532 3534 A9358A34           DBL   P3END-1,P3TITL-1
533 3538 A135B235           DBL   P3EQU-1,P3REM-1
534 353C BF35D535           DBL   P3RES-1,P3DTA-1
535 3540 FA353636           DBL   P3DBL-1,P3ASC-1
536 3544 4C369735           DBL   P3AST-1,P3ESP-1
537 3548 5336               DBL   P3$-1
538                  *
```

```
                        * INSTRUCTION TABLE
                        *
354A 72361937   P3JTBL  DBL   P3C1-1,P3C2-1
354E 56376337           DBL   P3C3-1,P3C4-1
3552 A637D937           DBL   P3C5-1,P3C6-1
3556 13384B38           DBL   P3C7-1,P3C8-1
355A 5238A138           DBL   P3C9-1,P3CA-1
355E A538E538           DBL   P3CB-1,P3CC-1
3562 E938               DBL   P3CD-1
                        *
                        * PSEUDO-OP PROCESSORS
                        *
3564 20283E     P3ORG   JSR   DPOBJ
3567 A900               CLA
3569 85B2               STA   SPDFLG
356B 205A2C             JSR   OPDSCN
356E A582               LDA   UT4
3570 85B6               STA   ILC
3572 8DE83E             STA   RCDORG
3575 A583               LDA   UT4+1
3577 85B7               STA   ILC+1
3579 8DE93E             STA   RCDORG+1
357C 4C9D3A     P3ORG3  JMP   P3PRT
                        *
357F 20283E     P3SPD   JSR   DPOBJ     DUMP BUFFER
3582 A901               LDA   #1
3584 85B2               STA   SPDFLG
3586 205A2C             JSR   OPDSCN
3589 A582               LDA   UT4
358B 85B0               STA   SPLC
358D 8DE83E             STA   RCDORG
3590 A900               CLA
3592 8DE93E             STA   RCDORG+1
3595 4C9D3A             JMP   P3PRT
                        *
3598 A900       P3ESP   CLA
359A 85B2               STA   SPDFLG    CLEAR SPD FLAG
359C 20283E             JSR   DPOBJ
359F 4C9D3A             JMP   P3PRT
                        *
                        P3TITL  EQU   P3A       DEFAULT
                        *
35A2 205A2C     P3EQU   JSR   OPDSCN    SCAN OPERAND
35A5 10D5               BPL   P3ORG3
35A7 4C5836             JMP   UERR
                        *
35AA CECF3D     P3END   DEC   ENDFLG    SET FLAG
35AD 20283E             JSR   DPOBJ     CLEAR OBJECT BUFFER
35B0 4C9D3A             JMP   P3PRT
                        *
35B3 A9A0       P3REM   LDA   #' '
35B5 8D522A             STA   INST      CLEAR INST. FIELD
35B8 8D532A             STA   INST+1
35BB 8D542A             STA   INST+2
35BE D0BC               BNZ   P3ORG3
                        *
35C0 205A2C     P3RES   JSR   OPDSCN    GET RES VALUE
35C3 A582               LDA   UT4
35C5 8DD63D             STA   RSLNTH
```

```
598 35C8 A583              LDA   UT4+1
599 35CA 8DD73D            STA   RSLNTH+1
600 35CD 20283E            JSR   DPOBJ      CLEAR OBJECT BUFFER
601 35D0 20723E            JSR   PCHRES     OUTPUT "RES" TYPE RECORD
602 35D3 4C9D3A            JMP   P3PRT
603                  *
604 35D6 E680        P3DTA INC   UT3
605 35D8 A580              LDA   UT3         GET FIELD COUNT
606 35DA 8DD43D            STA   P3LNTH
607 35DD 205A2C      P3DTA2 JSR   OPDSCN
608 35E0 3076              BMI   UERR
609 35E2 2951              AND   #$51
610 35E4 D003              BNZ   P3DTA3
611 35E6 206A36            JSR   CTERR
612 35E9 A582        P3DTA3 LDA   UT4
613 35EB A000              LDY   #0
614 35ED 9176              STA   *OBJPTR,Y
615 35EF E676D002          IDB   OBJPTR
616 35F5 C680              DEC   UT3
617 35F7 D0E4              BNZ   P3DTA2
618 35F9 F02E              BZ    P3DBLA
619                  *
620 35FB E680        P3DBL INC   UT3
621 35FD A580              LDA   UT3         GET FIELD COUNT
622 35FF 0A                ASLA              TIMES 2
623 3600 8DD43D            STA   P3LNTH
624 3603 205A2C      P3DBL2 JSR   OPDSCN     GET DBL VALUE
625 3606 3050              BMI   UERR
626 3608 2960              AND   #$60
627 360A D003              BNZ   P3DBLB
628 360C 206A36            JSR   CTERR
629 360F A000        P3DBLB LDY   #0
630 3611 A582              LDA   UT4
631 3613 9176              STA   *OBJPTR,Y
632 3615 E676D002          IDB   OBJPTR
633 361B A583              LDA   UT4+1
634 361D 9176              STA   *OBJPTR,Y
635 361F E676D002          IDB   OBJPTR
636 3625 C680              DEC   UT3
637 3627 D0DA              BNZ   P3DBL2
638 3629 ADD03D      P3DBLA LDA   P3T
639 362C D003              BNZ   P3DBL4
640 362E 4C9D3A      P3DBL3 JMP   P3PRT
641 3631 ADD13D      P3DBL4 LDA   P3T+1
642 3634 4CDE36            JMP   P3C1E
643                  *
644 3637 205A2C      P3ASC  JSR   OPDSCN
645 363A A5B8              LDA   ALNTH
646 363C 8DD43D            STA   P3LNTH
647 363F A000              LDY   #0
648 3641 B1AC        P3ASC2 LDA   *ASCPTR,Y TRANFER ASCII STRING
649 3643 9176              STA   *OBJPTR,Y TO OBJECT BUFFER
650 3645 C8                INY
651 3646 C6B8              DEC   ALNTH
652 3648 D0F7              BNZ   P3ASC2
653 364A 4C9D3A            JMP   P3PRT
654                  *
655 364D A901        P3AST  LDA   #1
656 364F 8DD43D      P3AST2 STA   P3LNTH
```

```
657 3652 D0DA              BNZ   P3DBL3
658                 *
659 3654 A902        P3$   LDA   #2
660 3656 D0F7              BNZ   P3AST2
661                 *
662 3658 A9D5        UERR  LDA   #'U'
663 365A 206036            JSR   STERCH
664 365D 4C9D3A            JMP   P3PRT
665 3660 ACD33D      STERCH LDY   ERCNT
666 3663 99DC3D            STA   ECA,Y
667 3666 EED33D            INC   ERCNT
668 3669 60                RTS
669 366A EED03D      CTERR INC   P3T
670 366D A584              LDA   ECODE
671 366F 8DD13D            STA   P3T+1
672 3672 60                RTS
673                 *
674                 * INSTRUCTION PROCESSORS
675                 *
676                                      MEMORY REFERENCE CLASS 1
677 3673 A902        P3C1  LDA   #2
678 3675 8DD43D      P3C1N STA   P3LNTH    SET LENGTH
679 3678 AD572A            LDA   OPND
680 367B C9A3              CMP   #'#'      CHECK FOR IMMEDIATE
681 367D F065              BEQ   P3C1IM
682 367F C9AA              CMP   #'*'      OR INDIRECT
683 3681 F078              BEQ   P3C1IN
684 3683 C9A8              CMP   #'('
685 3685 F074              BEQ   P3C1IN
686 3687 205A2C            JSR   OPDSCN    GET ADDRESS
687 368A 304D              BMI   P3C1C
688 368C A000              LDY   #0
689 368E 2940              AND   #$40      SCRATCHPAD ADDRESS?
690 3690 F024              BZ    P3C1B     NO
691 3692 A204              LDX   #4
692 3694 A58E              LDA   OPSTAT
693 3696 2904              AND   #4
694 3698 F00A              BZ    P3C1A
695 369A 9008              BCC   P3C1A
696 369C A214              LDX   #$14
697 369E B166              LDA   *UPTR1,Y  CHECK VALID INDEX CHAR.
698 36A0 C9D8              CMP   #'X'
699 36A2 D035              BNE   P3C1C
700 36A4 8A          P3C1A TXA
701 36A5 058F              ORA   SKEL      MERGE ADDRESSING MODE
702                                        WITH INST. SKELETON
703 36A7 9176              STA   *OBJPTR,Y
704 36A9 A582              LDA   UT4
705 36AB C8                INY
706 36AC 9176              STA   *OBJPTR,Y
707 36AE A583              LDA   UT4+1
708 36B0 C8                INY
709 36B1 9176              STA   *OBJPTR,Y
710 36B3 4C9D3A            JMP   P3PRT
711 36B6 EED43D      P3C1B INC   P3LNTH
712 36B9 A58E              LDA   OPSTAT
713 36BB 2920              AND   #$20
714 36BD F01D              BZ    P3C1D
715 36BF A20C              LDX   #$C
```

```
716 36C1 A58E            LDA   OPSTAT
717 36C3 2904            AND   #4
718 36C5 F0DD            BZ    P3C1A
719 36C7 A21C            LDX   #$1C
720 36C9 B166            LDA   *UPTR1,Y  CHECK VALID INDEX CHAR.
721 36CB C9D8            CMP   #'X'
722 36CD F0D5            BEQ   P3C1A
723 36CF A218            LDX   #$18
724 36D1 C9D9            CMP   #'Y'
725 36D3 F0CF            BEQ   P3C1A
726 36D5 A9C9            LDA   #'I'
727 36D7 D005            BNE   P3C1E
728 36D9 EED43D    P3C1C INC   P3LNTH
729 36DC A9D5      P3C1D LDA   #'U'      MARK UNDEFINED SYMBOL
730 36DE 206036    P3C1E JSR   STERCH
731 36E1 4C9D3A          JMP   P3PRT
732 36E4 A58F      P3C1IM LDA  SKEL      CHECK FOR STA #
733 36E6 C981            CMP   #$81
734 36E8 F0F2            BEQ   P3C1D
735 36EA E666D002        IDB   UPTR1     SKIP # CHARACTER
736 36F0 205A2C          JSR   OPDSCN
737 36F3 2910            AND   #$10      VALID IMMEDIATE OPERAND?
738 36F5 F0E5            BZ    P3C1D
739 36F7 A208            LDX   #$8
740 36F9 D0A9            BNZ   P3C1A
741 36FB E666D002  P3C1IN IDB  UPTR1     SKIP * OR (
742 3701 205A2C          JSR   OPDSCN
743 3704 2940            AND   #$40      VALID ADDRESS?
744 3706 F0D4            BZ    P3C1D     NO
745 3708 A000            LDY   #0
746 370A A200            LDX   #0
747 370C B166            LDA   *UPTR1,Y
748 370E C9D8            CMP   #'X'      VALID INDEX CHAR.?
749 3710 F092      P3C1J BEQ   P3C1A
750 3712 A210            LDX   #$10
751 3714 C9D9            CMP   #'Y'
752 3716 F08C            BEQ   P3C1A
753 3718 D0C2            BNE   P3C1D
754                *
755                * SHIFTS, INC/DEC
756                *
757 371A A902      P3C2  LDA   #2
758 371C 8DD43D          STA   P3LNTH
759 371F 205A2C          JSR   OPDSCN    GET OPERAND
760 3722 30B5            BMI   P3C1C
761 3724 A000            LDY   #0
762 3726 2940            AND   #$40      VALID SCRATCHPAD ADDRESS?
763 3728 F012            BZ    P3C2B
764 372A A200            LDX   #0
765 372C A58E            LDA   OPSTAT
766 372E 2904            AND   #4        END IN COMMA?
767 3730 F0DE      P3C2A BZ    P3C1J
768 3732 A210            LDX   #$10
769 3734 B166            LDA   *UPTR1,Y
770 3736 C9D8            CMP   #'X'
771 3738 D0A2            BNE   P3C1D
772 373A F0F4            BEQ   P3C2A
773 373C EED43D    P3C2B INC   P3LNTH
774 373F A58E            LDA   OPSTAT
```

```
775 3741 2920              AND   #$20       VALID ADDRESS AT ALL?
776 3743 F097              BZ    P3C1D      NO, TILT
777 3745 A208              LDX   #8
778 3747 A58E              LDA   OPSTAT
779 3749 2904              AND   #4         CHECK FOR COMMA
780 374B F0E3              BZ    P3C2A
781 374D A218              LDX   #$18
782 374F B166              LDA   *UPTR1,Y
783 3751 C9D8              CMP   #'X'
784 3753 D087       P3C2J  BNE   P3C1D
785 3755 F0D9              BEQ   P3C2A
786                 *
787                 * SINGLE BYTE, NO OPERAND
788                 *
789 3757 A901       P3C3   LDA   #1
790 3759 8DD43D            STA   P3LNTH
791 375C A58F              LDA   SKEL       MOVE SKELETON TO
792 375E 8DB13D            STA   OCODE      OBJECT BUFFER
793 3761 4C9D3A            JMP   P3PRT
794                 *
795                 * RELATIVE JUMPS
796                 *
797 3764 A902       P3C4   LDA   #2
798 3766 8DD43D            STA   P3LNTH     SET LENGTH
799 3769 A58F              LDA   SKEL
800 376B 8DB13D            STA   OCODE      OP CODE TO OBJ BFR.
801 376E 205A2C            JSR   OPDSCN
802 3771 30E0              BMI   P3C2J
803 3773 A582              LDA   UT4
804 3775 A683              LDX   UT4+1      GET TARGET ADDRESS
805 3777 38E902            SUB   #2
806 377A B001              BCS   P3C4A
807 377C CA                DEX
808 377D 38E5B6     P3C4A  SUB   ILC        CAN WE REACH IT?
809 3780 48                PHA
810 3781 8A                TXA
811 3782 E5B7              SBC   ILC+1
812 3784 F008              BZ    P3C4B
813 3786 C9FF              CMP   #$FF
814 3788 F004              BEQ   P3C4B
815 378A 68         P3C4E  PLA              NO
816 378B 4CA237            JMP   AERR
817 378E 8582       P3C4B  STA   UT4
818 3790 68                PLA
819 3791 48                PHA
820 3792 4582              EOR   UT4
821 3794 30F4              BMI   P3C4E
822 3796 68                PLA
823 3797 8DB23D            STA   OCODE+1    OK
824 379A A58F              LDA   SKEL
825 379C 8DB13D            STA   OCODE
826 379F 4C9D3A            JMP   P3PRT
827 37A2 A9C1       AERR   LDA   #'A'
828 37A4 4CDE36            JMP   P3C1E
829                 *
830 37A7 A902       P3C5   LDA   #2
831 37A9 8DD43D            STA   P3LNTH     SET LENGTH
832 37AC 205A2C            JSR   OPDSCN
833 37AF 1006              BPL   P3C5A
```

```
834 37B1 EED43D      P3C5E   INC   P3LNTH
835 37B4 4C5836      P3C5F   JMP   UERR
836 37B7 2940        P3C5A   AND   #$40
837 37B9 F012                BZ    P3C5C
838 37BB A924                LDA   #$24
839 37BD 8DB13D      P3C5B   STA   OCODE
840 37C0 A582                LDA   UT4
841 37C2 8DB23D              STA   OCODE+1
842 37C5 A583                LDA   UT4+1
843 37C7 8DB33D              STA   OCODE+2
844 37CA 4C9D3A              JMP   P3PRT
845 37CD A58E        P3C5C   LDA   OPSTAT
846 37CF 2920                AND   #$20
847 37D1 F0DE                BZ    P3C5E
848 37D3 EED43D              INC   P3LNTH
849 37D6 A92C                LDA   #$2C
850 37D8 D0E3                BNZ   P3C5B
851                  *
852                  * INDEX COMPARES
853                  *
854 37DA A902        P3C6    LDA   #2
855 37DC 8DD43D              STA   P3LNTH
856 37DF AD572A              LDA   OPND
857 37E2 C9A3                CMP   #'#'       CHECK IMMEDIATE
858 37E4 F01D                BEQ   P3C6IM
859 37E6 205A2C              JSR   OPDSCN
860 37E9 30C6        P3C6E   BMI   P3C5E
861 37EB A58F                LDA   SKEL
862 37ED 186904              ADD   #4
863 37F0 AA                  TAX
864 37F1 A58E                LDA   OPSTAT
865 37F3 2940                AND   #$40
866 37F5 D009                BNZ   P3C6A
867 37F7 A58F                LDA   SKEL
868 37F9 18690C              ADD   #$C
869 37FC AA                  TAX
870 37FD EED43D              INC   P3LNTH
871 3800 8A          P3C6A   TXA
872 3801 D0BA                BNZ   P3C5B
873 3803 E666D002    P3C6IM  IDB   UPTR1
874 3809 205A2C              JSR   OPDSCN
875 380C 2910                AND   #$10
876 380E F0A1                BZ    P3C5E
877 3810 A58F                LDA   SKEL
878 3812 D0A9                BNZ   P3C5B
879                  *
880                  * UNCONDITIONAL JUMP
881                  *
882 3814 A903        P3C7    LDA   #3
883 3816 8DD43D              STA   P3LNTH
884 3819 AD572A              LDA   OPND
885 381C C9AA                CMP   #'*'       CHECK FOR INDIRECT
886 381E F01F                BEQ   P3C7IN
887 3820 C9A8                CMP   #'('
888 3822 F01B                BEQ   P3C7IN
889 3824 205A2C      P3C7A   JSR   OPDSCN
890 3827 308B                BMI   P3C5F
891 3829 2960                AND   #$60
892 382B F087                BZ    P3C5F
```

```
893 382D A58F        P3C7B    LDA   SKEL
894 382F 8DB13D               STA   OCODE
895 3832 A582                 LDA   UT4
896 3834 8DB23D               STA   OCODE+1
897 3837 A583                 LDA   UT4+1
898 3839 8DB33D               STA   OCODE+2
899 383C 4C9D3A               JMP   P3PRT
900 383F A96C        P3C7IN   LDA   #$6C
901 3841 858F                 STA   SKEL
902 3843 E666D002             IDB   UPTR1
903 3849 4C2438               JMP   P3C7A
904                  *
905                  * SUBROUTINE CALL
906                  *
907 384C A903        P3C8     LDA   #3
908 384E 8DD43D               STA   P3LNTH
909 3851 D0D1                 BNZ   P3C7A
910                  *
911                  * LOAD X
912                  *
913 3853 A9D9        P3C9     LDA   #'Y'
914 3855 8DD23D      P3C9A    STA   CMPND
915 3858 A902                 LDA   #2
916 385A 8DD43D               STA   P3LNTH
917 385D AD572A               LDA   OPND
918 3860 C9A3                 CMP   #'#'
919 3862 F02C                 BEQ   P3C9IM
920 3864 205A2C               JSR   OPDSCN
921 3867 2940                 AND   #$40
922 3869 F018                 BZ    P3C9C
923 386B A204                 LDX   #4
924 386D A58E        P3C9B    LDA   OPSTAT
925 386F 2904                 AND   #4
926 3871 F00D                 BZ    P3C9R
927 3873 A000                 LDY   #0
928 3875 B166                 LDA   *UPTR1,Y
929 3877 CDD23D               CMP   CMPND
930 387A D004                 BNE   P3C9R
931 387C 8A                   TXA
932 387D 0910                 ORA   #$10
933 387F AA                   TAX
934 3880 4CA436      P3C9R    JMP   P3C1A
935 3883 EED43D      P3C9C    INC   P3LNTH
936 3886 A20C                 LDX   #$C
937 3888 A58E                 LDA   OPSTAT
938 388A 2920                 AND   #$20
939 388C D0DF                 BNZ   P3C9B
940 388E F00F                 BZ    P3C9E
941 3890 E666D002    P3C9IM   IDB   UPTR1
942 3896 205A2C               JSR   OPDSCN
943 3899 A200                 LDX   #0
944 389B 2910                 AND   #$10
945 389D D0E1                 BNZ   P3C9R
946 389F 4C5836      P3C9E    JMP   UERR
947                  *
948                  * LOAD Y
949                  *
950 38A2 A9D8        P3CA     LDA   #'X'
951 38A4 D0AF                 BNZ   P3C9A
```

```
952                     *
953                     * STORE X
954                     *
955 38A6 A9D9           P3CB    LDA   #'Y'
956 38A8 8DD23D         P3CBA   STA   CMPND
957 38AB A902                   LDA   #2
958 38AD 8DD43D                 STA   P3LNTH
959 38B0 205A2C                 JSR   OPDSCN
960 38B3 1006                   BPL   P3CBB
961 38B5 EED43D                 INC   P3LNTH
962 38B8 4CB137         P3CBER  JMP   P3C5E
963 38BB 2940           P3CBB   AND   #$40
964 38BD F01A                   BZ    P3CBD
965 38BF A200                   LDX   #0
966 38C1 A58E                   LDA   OPSTAT
967 38C3 2904                   AND   #4
968 38C5 F00F                   BZ    P3CBC
969 38C7 248E                   BIT   OPSTAT
970 38C9 50ED                   BVC   P3CBER
971 38CB A000                   LDY   #0
972 38CD B166                   LDA   *UPTR1,Y
973 38CF CDD23D                 CMP   CMPND
974 38D2 D0E4                   BNE   P3CBER
975 38D4 A210                   LDX   #$10
976 38D6 4CA436         P3CBC   JMP   P3C1A
977 38D9 EED43D         P3CBD   INC   P3LNTH
978 38DC A58E                   LDA   OPSTAT
979 38DE 2920                   AND   #$20
980 38E0 F0D6                   BZ    P3CBER
981 38E2 A208                   LDX   #8
982 38E4 D0F0                   BNZ   P3CBC
983                     *
984                     * STORE Y
985                     *
986 38E6 A9D8           P3CC    LDA   #'X'
987 38E8 D0BE                   BNZ   P3CBA
988                     *
989                     * PSEUDO INSTRUCTIONS
990                     *
991 38EA A58F           P3CD    LDA   SKEL
992 38EC F036                   BZ    DEA
993 38EE C901                   CMP   #1
994 38F0 F049                   BEQ   INA
995 38F2 C902                   CMP   #2
996 38F4 F04E                   BEQ   CLA
997 38F6 C903                   CMP   #3
998 38F8 F058                   BEQ   CMA
999 38FA C904                   CMP   #4
000 38FC D003                   BNE   P3CD1
001 38FE 4C6039                 JMP   TCA
002 3901 C905           P3CD1   CHP   #5
003 3903 F078                   BEQ   CDB
004 3905 C906                   CMP   #6
005 3907 D003                   BNE   P3CD2
006 3909 4C343A                 JMP   IDB
007 390C C961           P3CD2   CMP   #$61
008 390E F010                   BEQ   ADDI
009 3910 A938                   LDA   #$38      MUST BE SUBTRACT
010 3912 8DB13D         P3CDA   STA   OCODE
```

```
011 3915 E676D002           IDB   OBJPTR
012 391B A903               LDA   #3
013 391D 4C7536             JMP   P3C1N
014 3920 A918       ADDI    LDA   #$18
015 3922 D0EE               BNZ   P3CDA
016 3924 A938       DEA     LDA   #$38
017 3926 8DB13D             STA   OCODE
018 3929 A9E9               LDA   #$E9
019 392B 8DB23D     DEA2    STA   OCODE+1
020 392E A901               LDA   #1
021 3930 8DB33D             STA   OCODE+2
022 3933 A903               LDA   #3
023 3935 8DD43D     DEA3    STA   P3LNTH
024 3938 4C9D3A             JMP   P3PRT
025 393B A918       INA     LDA   #$18
026 393D 8DB13D             STA   OCODE
027 3940 A969               LDA   #$69
028 3942 D0E7               BNZ   DEA2
029 3944 A9A9       CLA     LDA   #$A9
030 3946 8DB13D             STA   OCODE
031 3949 A900               LDA   #0
032 394B 8DB23D             STA   OCODE+1
033 394E A902               LDA   #2
034 3950 D0E3               BNZ   DEA3
035 3952 A949       CMA     LDA   #$49
036 3954 8DB13D             STA   OCODE
037 3957 A9FF               LDA   #$FF
038 3959 8DB23D             STA   OCODE+1
039 395C A902               LDA   #2
040 395E D0D5               BNZ   DEA3
041 3960 A918       TCA     LDA   #$18
042 3962 8DB13D             STA   OCODE
043 3965 A949               LDA   #$49
044 3967 8DB23D             STA   OCODE+1
045 396A A9FF               LDA   #$FF
046 396C 8DB33D             STA   OCODE+2
047 396F A969               LDA   #$69
048 3971 8DB43D             STA   OCODE+3
049 3974 A901               LDA   #1
050 3976 8DB53D             STA   OCODE+4
051 3979 A905               LDA   #5
052 397B D0B8               BNZ   DEA3
053 397D A908       CDB     LDA   #8
054 397F 8DD43D             STA   P3LNTH
055 3982 205A2C             JSR   OPDSCN
056 3985 3043               BMI   CDUERR
057 3987 2904               AND   #4
058 3989 F03F               BZ    CDUERR
059 398B A58E               LDA   OPSTAT
060 398D 2940               AND   #$40
061 398F F050               BZ    CDBMM
062 3991 A582               LDA   UT4
063 3993 C9FF               CMP   #$FF
064 3995 F03C               BEQ   CDPERR
065 3997 859D               STA   IDIFF
066 3999 205A2C             JSR   OPDSCN
067 399C 302C               BMI   CDUERR
068 399E 2960               AND   #$60
069 39A0 F028               BZ    CDUERR
```

```
070 39A2 A9A9              LDA   #$A9
071 39A4 8DB13D            STA   OCODE
072 39A7 8DB53D            STA   OCODE+4
073 39AA A985              LDA   #$85
074 39AC 8DB33D            STA   OCODE+2
075 39AF 8DB73D            STA   OCODE+6
076 39B2 A59D              LDA   IDIFF
077 39B4 8DB43D            STA   OCODE+3
078 39B7 186901            INA
079 39BA 8DB83D            STA   OCODE+7
080 39BD A582              LDA   UT4
081 39BF 8DB23D            STA   OCODE+1
082 39C2 A583              LDA   UT4+1
083 39C4 8DB63D            STA   OCODE+5
084 39C7 4C9D3A            JMP   P3PRT
085 39CA EED43D    CDUERR  INC   P3LNTH
086 39CD EED43D            INC   P3LNTH
087 39D0 4C5836            JMP   UERR
088 39D3 EED43D    CDPERR  INC   P3LNTH
089 39D6 EED43D            INC   P3LNTH
090 39D9 A9C2              LDA   #'B'
091 39DB 206036            JSR   STERCH
092 39DE 4C9D3A            JMP   P3PRT
093 39E1 A58E      CDBMM   LDA   OPSTAT
094 39E3 2920              AND   #$20
095 39E5 F0E3              BZ    CDUERR
096 39E7 A582              LDA   UT4
097 39E9 859D              STA   IDIFF
098 39EB A583              LDA   UT4+1
099 39ED 859E              STA   IDIFF+1
100 39EF 205A2C            JSR   OPDSCN
101 39F2 30D6              BMI   CDUERR
102 39F4 2960              AND   #$60
103 39F6 F0D2      CDBMMJ  BZ    CDUERR
104 39F8 A90A              LDA   #10
105 39FA 8DD43D            STA   P3LNTH
106 39FD A9A9              LDA   #$A9
107 39FF 8DB13D            STA   OCODE
108 3A02 8DB63D            STA   OCODE+5
109 3A05 A98D              LDA   #$8D
110 3A07 8DB33D            STA   OCODE+2
111 3A0A 8DB83D            STA   OCODE+7
112 3A0D A59D              LDA   IDIFF
113 3A0F 8DB43D            STA   OCODE+3
114 3A12 A59E              LDA   IDIFF+1
115 3A14 8DB53D            STA   OCODE+4
116 3A17 E69DD002          IDB   IDIFF
117 3A1D A59D              LDA   IDIFF
118 3A1F 8DB93D            STA   OCODE+8
119 3A22 A59E              LDA   IDIFF+1
120 3A24 8DBA3D            STA   OCODE+9
121 3A27 A582              LDA   UT4
122 3A29 8DB23D            STA   OCODE+1
123 3A2C A583              LDA   UT4+1
124 3A2E 8DB73D            STA   OCODE+6
125 3A31 4C9D3A            JMP   P3PRT
126 3A34 A906      IDB     LDA   #6
127 3A36 8DD43D            STA   P3LNTH
128 3A39 205A2C            JSR   OPDSCN
```

```
129 3A3C 308C              BMI   CDUERR
130 3A3E 2940              AND   #$40
131 3A40 F024              BZ    IDBMM
132 3A42 A582              LDA   UT4
133 3A44 C9FF              CMP   #$FF
134 3A46 F08B              BEQ   CDPERR
135 3A48 8DB23D            STA   OCODE+1
136 3A4B 186901            INA
137 3A4E 8DB63D            STA   OCODE+5
138 3A51 A9E6              LDA   #$E6
139 3A53 8DB13D            STA   OCODE
140 3A56 8DB53D            STA   OCODE+4
141 3A59 A9D0              LDA   #$D0
142 3A5B 8DB33D            STA   OCODE+2
143 3A5E A902              LDA   #2
144 3A60 8DB43D            STA   OCODE+3
145 3A63 4C9D3A            JMP   P3PRT
146 3A66 A908      IDBMM   LDA   #8
147 3A68 8DD43D            STA   P3LNTH
148 3A6B A58E              LDA   OPSTAT
149 3A6D 2920              AND   #$20
150 3A6F F085              BZ    CDBMMJ
151 3A71 A582              LDA   UT4
152 3A73 8DB23D            STA   OCODE+1
153 3A76 A583              LDA   UT4+1
154 3A78 8DB33D            STA   OCODE+2
155 3A7B E682D002          IDB   UT4
156 3A81 A582              LDA   UT4
157 3A83 8DB73D            STA   OCODE+6
158 3A86 A583              LDA   UT4+1
159 3A88 8DB83D            STA   OCODE+7
160 3A8B A9D0              LDA   #$D0
161 3A8D 8DB43D            STA   OCODE+3
162 3A90 A903              LDA   #3
163 3A92 8DB53D            STA   OCODE+4
164 3A95 A9EE              LDA   #$EE
165 3A97 8DB13D            STA   OCODE
166 3A9A 8DB63D            STA   OCODE+5
167 3A9D 20E53D    P3PRT   JSR   PCHOBJ
168 3AA0 A5B2              LDA   SPDFLG
169 3AA2 F010              BZ    P3PRTB
170 3AA4 A93B856A          CDB   UPTR3,LOC+1
171 3AAC A5B0              LDA   SPLC
172 3AAE 209826            JSR   CNV2HX
173 3AB1 4CC63A            JMP   P3PRTC
174 3AB4 A93A856A  P3PRTB  CDB   UPTR3,LOC
175 3ABC A5B7              LDA   ILC+1
176 3ABE 209826            JSR   CNV2HX
177 3AC1 A5B6              LDA   ILC
178 3AC3 209826            JSR   CNV2HX
179 3AC6 A93F856A  P3PRTC  CDB   UPTR3,CONT
180 3ACE ADD43D            LDA   P3LNTH
181 3AD1 F021              BZ    P3NXT
182 3AD3 C905              CMP   #5
183 3AD5 9002              BLS   P3PRTZ
184 3AD7 A904              LDA   #4
185 3AD9 859D      P3PRTZ  STA   IDIFF
186 3ADB A9B18585          CDB   TVAL,OCODE
187 3AE3 A000      P3PRTD  LDY   #0
```

```
188 3AE5 B185              LDA   *TVAL,Y
189 3AE7 E685D002          IDB   TVAL
190 3AED 209826            JSR   CNV2HX
191 3AF0 C69D              DEC   IDIFF
192 3AF2 D0EF              BNZ   P3PRTD
193 3AF4 ADD33D     P3NXT  LDA   ERCNT
194 3AF7 F010              BZ    P3PRTE
195 3AF9 A936              LDA   #<LNO
196 3AFB A22A              LDX   #>LNO
197 3AFD 20D420            JSR   DISP
198 3B00 A9D8              LDA   #<ERCDLN
199 3B02 A23D              LDX   #>ERCDLN
200 3B04 20D420            JSR   DISP
201 3B07 A22A              LDX   #>LNO
202 3B09 E691D002   P3PRTE IDB   FIRST
203 3B0F D004              BNZ   MPTRA
204 3B11 A901              LDA   #1
205 3B13 85A9              STA   LFLAG
206 3B15 E693D002   MPTRA  IDB   SECOND
207 3B1B D004              BNZ   LDA
208 3B1D A900              CLA
209 3B1F 85A9              STA   LFLAG
210 3B21 A936       LDA    LDA   #<LNO
211 3B23 A22A              LDX   #>LNO
212 3B25 20AC3C            JSR   LPNT
213 3B28 ADD33D            LDA   ERCNT
214 3B2B F007              BZ    P3PRTF
215 3B2D A9D8              LDA   #<ERCDLN
216 3B2F A23D              LDX   #>ERCDLN
217 3B31 20AC3C            JSR   LPNT
218 3B34 A5B2       P3PRTF LDA   SPDFLG
219 3B36 F016              BZ    P3PRTG
220 3B38 A5B0              LDA   SPLC
221 3B3A 186DD43D          ADD   P3LNTH
222 3B3E B006              BCS   SPVXFR
223 3B40 186DD63D          ADD   RSLNTH
224 3B44 9003              BCC   PCHLC1
225 3B46 4C8A3D     SPVXFR JMP   SPOV3
226 3B49 85B0       PCHLC1 STA   SPLC
227 3B4B 4C6A3B            JMP   P3PRTX
228 3B4E A5B6       P3PRTG LDA   ILC
229 3B50 186DD43D          ADD   P3LNTH
230 3B54 9002              BCC   P3PRTH
231 3B56 E6B7              INC   ILC+1
232 3B58 186DD63D   P3PRTH ADD   RSLNTH
233 3B5C 9002              BCC   P3PRTI
234 3B5E E6B7              INC   ILC+1
235 3B60 85B6       P3PRTI STA   ILC
236 3B62 A5B7              LDA   ILC+1
237 3B64 186DD73D          ADD   RSLNTH+1
238 3B68 85B7              STA   ILC+1
239 3B6A EECF3D     P3PRTX INC   ENDFLG
240 3B6D F003              BZ    P3PRTJ
241 3B6F 4C8B34            JMP   P3A
242 3B72 20D93E     P3PRTJ JSR   WTRLR
243 3B75 A5A9              LDA   LFLAG
244 3B77 D001              BNZ   PSYMTB
245 3B79 60                RTS
246 3B7A A59F       PSYMTB LDA   NSYMS
```

```
247 3B7C 49FF               CMA
248 3B7E 85A1               STA   SYMCNT
249 3B80 A5A0               LDA   NSYMS+1
250 3B82 49FF               CMA
251 3B84 85A2               STA   SYMCNT+1
252 3B86 A5A5               LDA   SYMTAB
253 3B88 85A3               STA   SYMPTR
254 3B8A A5A6               LDA   SYMTAB+1
255 3B8C 85A4               STA   SYMPTR+1
256 3B8E A000               LDY   #0
257 3B90 A5BB               LDA   PW
258 3B92 38E90D     PSYM1   SUB   #13         FIND SYMBOL WIDTH
259 3B95 3003               BMI   PSYM1A
260 3B97 C8                 INY
261 3B98 D0F8               BNZ   PSYM1
262 3B9A 8CAA3C     PSYM1A  STY   SYMWD       SYMBOLS/LINE
263 3B9D E6A1D002           IDB   SYMCNT      ANY SYMBOLS?
264 3BA3 D003               BNZ   PSYM1B
265 3BA5 4C6C3C             JMP   EOSYM
266 3BA8 A5BA       PSYM1B  LDA   LPP
267 3BAA F015               BZ    PSYM1C
268 3BAC AD383D             LDA   LPPCNT
269 3BAF C905               CMP   #5
270 3BB1 B00E               BGE   PSYM1C
271 3BB3 A978       PIAS    LDA   #<BLKLIN
272 3BB5 A23F               LDX   #>BLKLIN
273 3BB7 20AC3C             JSR   LPNT
274 3BBA AD383D             LDA   LPPCNT
275 3BBD C5BA               CMP   LPP
276 3BBF D0F2               BNE   PIAS
277 3BC1 A978       PSYM1C  LDA   #<BLKLIN
278 3BC3 A23F               LDX   #>BLKLIN
279 3BC5 20AC3C             JSR   LPNT
280 3BC8 A97B               LDA   #<ASTLIN
281 3BCA A23F               LDX   #>ASTLIN
282 3BCC 20AC3C             JSR   LPNT
283 3BCF A999               LDA   #<STLINE
284 3BD1 A23F               LDX   #>STLINE
285 3BD3 20AC3C             JSR   LPNT
286 3BD6 A97B               LDA   #<ASTLIN
287 3BD8 A23F               LDX   #>ASTLIN
288 3BDA 20AC3C             JSR   LPNT
289 3BDD A978               LDA   #<BLKLIN
290 3BDF A23F               LDX   #>BLKLIN
291 3BE1 20AC3C             JSR   LPNT
292 3BE4 ADAA3C     PSYM2   LDA   SYMWD
293 3BE7 8DAB3C             STA   WCNT
294 3BEA A9368566           CDB   UPTR1,LNO SET OUTPUT BUFFER PTR.
295 3BF2 A200       PSYM2A  LDX   #0
296 3BF4 A000               LDY   #0
297 3BF6 A1A3       PSYM3   LDA   *SYMPTR,X
298 3BF8 E6A3D002           IDB   SYMPTR
299 3BFE 9166               STA   *UPTR1,Y
300 3C00 C8                 INY
301 3C01 C006               CPY   #6
302 3C03 90F1               BLS   PSYM3
303 3C05 A9A0               LDA   #' '
304 3C07 9166               STA   *UPTR1,Y  BLANK BETW. SYMBOL & VALUE
305 3C09 C8                 INY
```

```
306 3C0A A1A3               LDA   *SYMPTR,X GET LOW VALUE
307 3C0C 48                 PHA               SAVE ON STACK
308 3C0D E6A3D002           IDB   SYMPTR
309 3C13 A1A3               LDA   *SYMPTR,X
310 3C15 E6A3D002           IDB   SYMPTR
311 3C1B 208F3C             JSR   HXCNV
312 3C1E 9166               STA   *UPTR1,Y
313 3C20 C8                 INY
314 3C21 8A                 TXA
315 3C22 9166               STA   *UPTR1,Y
316 3C24 C8                 INY
317 3C25 68                 PLA
318 3C26 208F3C             JSR   HXCNV
319 3C29 9166               STA   *UPTR1,Y
320 3C2B C8                 INY
321 3C2C 8A                 TXA
322 3C2D 9166               STA   *UPTR1,Y
323 3C2F C8                 INY
324 3C30 A9A0               LDA   #' '        BLANKS BETW. SYMBOLS
325 3C32 9166               STA   *UPTR1,Y
326 3C34 C8                 INY
327 3C35 9166               STA   *UPTR1,Y
328 3C37 E6A1D002           IDB   SYMCNT      FINISHED??
329 3C3D F022               BZ    PSYM6
330 3C3F A566               LDA   UPTR1
331 3C41 18690D             ADD   #13
332 3C44 8566               STA   UPTR1
333 3C46 9002               BCC   PSYM4
334 3C48 E667               INC   UPTR1+1
335 3C4A CEAB3C     PSYM4   DEC   WCNT        LINE FILLED??
336 3C4D D0A3               BNZ   PSYM2A
337 3C4F A98D       PSYM5   LDA   #$8D
338 3C51 9166               STA   *UPTR1,Y
339 3C53 A936               LDA   #<LNO
340 3C55 A22A               LDX   #>LNO
341 3C57 857C               STA   UT1
342 3C59 20AC3C             JSR   LPNT
343 3C5C A57C               LDA   UT1
344 3C5E 4CE43B             JMP   PSYM2
345 3C61 A98D       PSYM6   LDA   #$8D
346 3C63 9166               STA   *UPTR1,Y
347 3C65 A936               LDA   #<LNO
348 3C67 A22A               LDX   #>LNO
349 3C69 20AC3C             JSR   LPNT
350 3C6C AD383D     EOSYM   LDA   LPPCNT
351 3C6F C5BA               CMP   LPP
352 3C71 F019               BEQ   EOSYM3
353 3C73 EE383D             INC   LPPCNT
354 3C76 EE383D             INC   LPPCNT
355 3C79 20233D     EOSYM2  JSR   PRBLN
356 3C7C CE383D             DEC   LPPCNT
357 3C7F D0F8               BNZ   EOSYM2
358 3C81 A5BD               LDA   DSHFLG
359 3C83 F007               BZ    EOSYM3
360 3C85 A93A               LDA   #<DASHES
361 3C87 A23D               LDX   #>DASHES
362 3C89 202B3D             JSR   MPRT
363 3C8C 4C2520     EOSYM3  JMP   MON1        FINISHED
364 3C8F 48         HXCNV   PHA               SAVE INPUT
```

```
365 3C90 209D3C             JSR   NXCNV
366 3C93 AA                 TAX
367 3C94 68                 PLA
368 3C95 6A                 RORA
369 3C96 6A                 RORA
370 3C97 6A                 RORA
371 3C98 6A                 RORA
372 3C99 209D3C             JSR   NXCNV
373 3C9C 60                 RTS
374 3C9D 290F       NXCNV   AND   #$F
375 3C9F C90A               CMP   #$A
376 3CA1 9003               BLS   NXCNV2
377 3CA3 186907             ADD   #7
378 3CA6 1869B0     NXCNV2  ADD   #$B0
379 3CA9 60                 RTS
380 3CAA 00         SYMWD   ***
381 3CAB 00         WCNT    ***
382 3CAC 202B3D     LPNT    JSR   MPRT
383 3CAF A5BA               LDA   LPP
384 3CB1 D01A               BNZ   LPNT2
385 3CB3 A5A9       LPNT1   LDA   LFLAG
386 3CB5 F004               BZ    LPRTS
387 3CB7 A5BC               LDA   PAUSCT
388 3CB9 D001               BNZ   LPNT1A
389 3CBB 60         LPRTS   RTS
390 3CBC CE393D     LPNT1A  DEC   PSCT
391 3CBF D0FA               BNZ   LPRTS
392 3CC1 8D393D             STA   PSCT
393 3CC4 20B620             JSR   DTCC
394 3CC7 20B920             JSR   SB
395 3CCA 4CBB3C             JMP   LPRTS
396 3CCD CE383D     LPNT2   DEC   LPPCNT
397 3CD0 D0E9               BNZ   LPRTS
398 3CD2 8D383D             STA   LPPCNT
399 3CD5 20233D             JSR   PRBLN
400 3CD8 20233D             JSR   PRBLN
401 3CDB A5BD               LDA   DSHFLG
402 3CDD F00A               BZ    LPNT3
403 3CDF A93A               LDA   #<DASHES
404 3CE1 A23D               LDX   #>DASHES
405 3CE3 202B3D             JSR   MPRT
406 3CE6 4CEC3C             JMP   LPNT4
407 3CE9 20233D     LPNT3   JSR   PRBLN
408 3CEC 20233D     LPNT4   JSR   PRBLN
409 3CEF EE373D             INC   BPGNO
410 3CF2 AD373D             LDA   BPGNO
411 3CF5 A2B0               LDX   #$B0
412 3CF7 38E964     LPCNV1  SUB   #100
413 3CFA 9003               BCC   LPCNV2
414 3CFC E8                 INX
415 3CFD D0F8               BNZ   LPCNV1
416 3CFF 186964     LPCNV2  ADD   #100
417 3D02 8EBE31             STX   DPGNO
418 3D05 A2B0               LDX   #$B0
419 3D07 38E90A     LPCNV3  SUB   #10
420 3D0A 9003               BCC   LPCNV4
421 3D0C E8                 INX
422 3D0D D0F8               BNZ   LPCNV3
423 3D0F 1869BA     LPCNV4  ADD   #$BA
```

```
424 3D12 8EBF31            STX   DPGNO+1
425 3D15 8DC031            STA   DPGNO+2
426 3D18 A9B9              LDA   #<HDNG
427 3D1A A231              LDX   #>HDNG
428 3D1C 202B3D            JSR   MPRT
429 3D1F 20233D            JSR   PRBLN
430 3D22 60                RTS
431 3D23 A978       PRBLN  LDA   #<BLKLIN
432 3D25 A23F              LDX   #>BLKLIN
433 3D27 202B3D            JSR   MPRT
434 3D2A 60                RTS
435 3D2B 48         MPRT   PHA
436 3D2C A5A9       MPRTB  LDA   LFLAG
437 3D2E F005              BZ    MPRT2
438 3D30 68                PLA
439 3D31 48                PHA
440 3D32 20B020            JSR   PRT
441 3D35 68         MPRT2  PLA
442 3D36 60                RTS
443 3D37 00         BPGNO  ***
444 3D38 00         LPPCNT ***
445 3D39 00         PSCT   ***
446 3D3A ADADADAD   DASHES ASC   '----------'
447 3D4E ADADADAD          ASC   '----------'
448 3D62 ADADADAD          ASC   '----------'
449 3D76 ADADADAD          ASC   '----------'
450 3D89 8D                DATA  $8D
451 3D8A A936       SPOV3  LDA   #<LNO
452 3D8C A22A              LDX   #>LNO
453 3D8E 20D420            JSR   DISP
454 3D91 A99D              LDA   #<SPOVMS
455 3D93 A23D              LDX   #>SPOVMS
456 3D95 20AC3C            JSR   LPNT
457 3D98 A900              LDA   #0
458 3D9A 4C493B            JMP   PCHLC1
459 3D9D D3C3D2C1   SPOVMS ASC   'SCRATCHPAD OVERFLOW'
460 3DB0 8D                DATA  $8D
461 3DB1            OCODE  RES   30
462 3DCF 00         ENDFLG ***
463 3DD0 0000       P3T    DBL   0
464 3DD2 00         CMPND  ***
465 3DD3 00         ERCNT  ***
466 3DD4 0000       P3LNTH DBL   0
467 3DD6 0000       RSLNTH DBL   0
468 3DD8 A0A0A0A0   ERCDLN ASC   '    '
469 3DDC            ECA    RES   8
470 3DE4 8D                DATA  $8D
471                 *
472 3DE5 A5AA       PCHOBJ LDA   OBFLAG
473 3DE7 D001              BNZ   PCHOB2
474 3DE9 60         PCHOB1 RTS
475 3DEA ADD43D     PCHOB2 LDA   P3LNTH
476 3DED F0FA              BZ    PCHOB1
477 3DEF 8DE43E            STA   OCNT
478 3DF2 186DE53E          ADD   OBJCNT
479 3DF6 1006              BPL   PCHOB3
480 3DF8 20283E            JSR   DPOBJ
481 3DFB 4CEA3D            JMP   PCHOB2
482 3DFE 8DE53E     PCHOB3 STA   OBJCNT
```

```
483 3E01 8DEB3E            STA   RCDLNT
484 3E04 A9B18576          CDB   OBJPTR,OCODE
485 3E0C A000              LDY   #0
486 3E0E A200              LDX   #0
487 3E10 B176      PCHOB4  LDA   *OBJPTR,Y MOVE OBJECT CODE
488 3E12 8164              STA   *OBFPTR,X FROM PASS 3 BUFFER TO
489 3E14 186DE63E          ADD   CKSM
490 3E18 8DE63E            STA   CKSM
491 3E1B E664D002          IDB   OBFPTR    OBJECT OUTPUT BUFFER
492 3E21 C8                INY
493 3E22 CEE43E            DEC   OCNT
494 3E25 D0E9              BNZ   PCHOB4
495 3E27 60                RTS
496                *
497 3E28 ACE53E    DPOBJ   LDY   OBJCNT    DUMP OBJECT BUFFER
498 3E2B F031              BZ    DPOBJ2
499 3E2D ADE63E            LDA   CKSM
500 3E30 186DE73E          ADD   RM0
501 3E34 186DE83E          ADD   RCDORG
502 3E38 186DE93E          ADD   RCDORG+1
503 3E3C 186DEB3E          ADD   RCDLNT
504 3E40 99EC3E            STA   OBFR,Y
505 3E43 98                TYA
506 3E44 186906            ADD   #6
507 3E47 A8                TAY
508 3E48 A9E7              LDA   #<RM0
509 3E4A A23E              LDX   #>RM0
510 3E4C 20BF20            JSR   WOBJR     WRITE OBJECT RECORD
511 3E4F ADE53E            LDA   OBJCNT
512 3E52 186DE83E          ADD   RCDORG
513 3E56 8DE83E            STA   RCDORG
514 3E59 9003              BCC   DPOBJ2
515 3E5B EEE93E            INC   RCDORG+1
516 3E5E A900      DPOBJ2  CLA
517 3E60 8DE53E            STA   OBJCNT
518 3E63 8DE63E            STA   CKSM
519 3E66 8DEB3E            STA   RCDLNT
520 3E69 A9EC8564          CDB   OBFPTR,OBFR
521 3E71 60                RTS
522                *
523 3E72 A5AA      PCHRES  LDA   OBFLAG
524 3E74 F058              BZ    PCHRS5
525 3E76 20283E            JSR   DPOBJ
526                                FIRST CLEAR OUT
527                                NORMAL OBJECT BUFFER
528 3E79 A5B2              LDA   SPDFLG
529 3E7B F00C              BZ    PCHRS2
530 3E7D A5B0              LDA   SPLC
531 3E7F 8D703F            STA   T1ORG
532 3E82 A900              CLA
533 3E84 8D713F            STA   T1ORG+1
534 3E87 F00A              BZ    PCHRS3
535 3E89 A5B6      PCHRS2  LDA   ILC
536 3E8B 8D703F            STA   T1ORG
537 3E8E A5B7              LDA   ILC+1
538 3E90 8D713F            STA   T1ORG+1
539 3E93 ADD63D    PCHRS3  LDA   RSLNTH
540 3E96 8D753F            STA   T1LNTH
541 3E99 ADD73D            LDA   RSLNTH+1
```

```
542 3E9C 8D763F             STA   T1LNTH+1
543 3E9F A900               CLA
544 3EA1 8D733F             STA   FLFLAG
545 3EA4 A007               LDY   #T1CKSM-RM1-1
546 3EA6 A900               CLA
547 3EA8 18796F3F   PCHRS4  ADD   RM1,Y
548 3EAC 88                 DEY
549 3EAD 10F9               BPL   PCHRS4
550 3EAF A009               LDY   #T1CKSM-RM1+1
551 3EB1 8D773F             STA   T1CKSM
552 3EB4 A96F               LDA   #<RM1
553 3EB6 A23F               LDX   #>RM1
554 3EB8 20BF20             JSR   WOBJR
555 3EBB ADE83E             LDA   RCDORG
556 3EBE 186DD63D           ADD   RSLNTH
557 3EC2 8DE83E             STA   RCDORG
558 3EC5 ADE93E             LDA   RCDORG+1
559 3EC8 6DD73D             ADC   RSLNTH+1
560 3ECB 8DE93E             STA   RCDORG+1
561 3ECE 60         PCHRS5  RTS
562                 *
563 3ECF A5AA       WLDR    LDA   OBFLAG
564 3ED1 F0FB               BZ    PCHRS5
565 3ED3 20BC20             JSR   WOBL
566 3ED6 4C5E3E             JMP   DPOBJ2
567                 *
568 3ED9 A5AA       WTRLR   LDA   OBFLAG
569 3EDB F0F1               BZ    PCHRS5
570 3EDD 20283E             JSR   DPOBJ
571 3EE0 20C220             JSR   WOBTR     WRITE OBJECT TRAILER
572 3EE3 60                 RTS
573 3EE4 00         OCNT    ***
574 3EE5 00         OBJCNT  ***
575                 *
576                 * TYPE 0 RECORD
577                 *
578 3EE6 00         CKSM    ***
579 3EE7 FF         RM0     DATA  $FF
580 3EE8 0000       RCDORG  DBL   0
581 3EEA 00         TYPE    DATA  0
582 3EEB 00         RCDLNT  DATA  0
583 3EEC            OBFR    RES   131
584                 *
585                 * TYPE 1 (RES) RECORD
586                 *
587 3F6F FF         RM1     DATA  $FF
588 3F70 0000       T1ORG   DBL   0
589 3F72 01         TYPE1   DATA  1
590 3F73 00         FLFLAG  ***
591 3F74 00         FLCHR   ***
592 3F75 0000       T1LNTH  DBL   0
593 3F77 00         T1CKSM  ***
594 3F78 A0A0       BLKLIN  ASC   '  '
595 3F7A 8D                 DATA  $8D
596 3F7B AAAAAAAA   ASTLIN  ASC   '******************************'
597 3F98 8D                 DATA  $8D
598 3F99 AAA0A0D3   STLINE  ASC   '*  S Y M B O L   T A B L E  *'
599 3FB6 8D                 DATA  $8D
600                         SPD   $60
```

```
601 0060 2520        MONPTR   DBL   MON1
602 0062 4F20        ERPTR    DBL   ERR
603 0064 0000        OBFPTR   DBL   0
604 0066 0000        UPTR1    DBL   0
605 0068 0000        UPTR2    DBL   0
606 006A 0000        UPTR3    DBL   0
607 006C 0000        UPTR4    DBL   0
608 006E 0000        UPTR5    DBL   0
609 0070 0000        UPTR6    DBL   0
610 0072 0000        UPTR7    DBL   0
611 0074 0000        UPTR8    DBL   0
612 0076 0000        OBJPTR   DBL   0
613 0078 0000        LNCNT    DBL   0
614 007A C520        NXTCHR   DBL   BFR
615 007C 0000        UT1      DBL   0
616 007E 0000        UT2      DBL   0
617 0080 0000        UT3      DBL   0
618 0082 0000        UT4      DBL   0
619 0084 00          ECODE    ***
620 0085 0000        TVAL     DBL   0
621 0087 00          FLO2     ***
622 0088 00          FHI1     ***
623 0089 00          FHI2     ***
624 008A 00          LMASK    ***
625 008B 00          HMASK    ***
626 008C 00          OPR8R    ***
627 008D 00          TMN8R    ***
628 008E 00          OPSTAT   ***
629 008F 00          SKEL     ***
630 0090 00          CLASS    ***
631 0091 0000        FIRST    DBL   0
632 0093 0000        SECOND   DBL   0
633 0095 0000        BADR1    DBL   0
634 0097 0000        EADR1    DBL   0
635 0099 0000        BADR2    DBL   0
636 009B 0000        EADR2    DBL   0
637 009D 0000        IDIFF    DBL   0
638 009F 0000        NSYMS    DBL   0
639 00A1 0000        SYMCNT   DBL   0
640 00A3 0000        SYMPTR   DBL   0
641 00A5 0000        SYMTAB   DBL   0
642 00A7 00          SCOUNT   ***
643 00A8 00          PAGE     ***
644 00A9 00          LFLAG    ***
645 00AA 00          OBFLAG   ***
646 00AB 00          NTERMS   ***
647 00AC 0000        ASCPTR   DBL   0
648 00AE 0000        SRCLNO   DBL   0
649 00B0 00          SPLC     ***
650 00B1 00          TLFLAG   ***
651 00B2 00          SPDFLG   ***
652 00B3 0000        LENGTH   DBL   0
653 00B5 00          P1ERFL   ***
654 00B6 0000        ILC      DBL   0
655 00B8 00          ALNTH    ***
656 00B9 00          IMVAL    ***
657 00BA 00          LPP      ***             LINES/PRINTED PAGE
658 00BB 00          PW       ***             PRINTER WIDTH
659 00BC 00          PAUSCT   ***             LINES BETW. PAUSES ON CRT
```

```
660 00BD 00          DSHFLG  ***              PAGE SEPARATOR FLAG
661 00BE                     END
```

****
* S Y M B O L T A B L E *
****

ACNT	2411	ADD	2296	ADD0	22B3	ADD01	22C0
ADD1	22DD	ADD10	2360	ADD2	22E8	ADD3	2304
ADD4	230E	ADD4A	2313	ADD5	231D	ADD6	2323
ADD7	2331	ADD8	233B	ADD9	2343	ADD9A	2350
ADD9B	2354	ADDF	236C	ADDF1	237B	ADDF2	2382
ADDF3	2385	ADDF4	238A	ADDI	3920	ADERR	230B
AERR	37A2	ALNTH	00B8	APTR	2412	ASCPTR	00AC
ASMBL	2EFE	ASMERR	2F2A	ASTLIN	3F7B	BADR1	0095
BADR2	0099	BCCFPE	21D3	BFA	240E	BFE	24E1
BFM	2689	BFOMS	2415	BFR	20C5	BLKLIN	3F78
BMLBL	34E2	BPGNO	3D37	CBITS	2C59	CDB	397D
CDBMM	39E1	CDBMMJ	39F6	CDPERR	39D3	CDUERR	39CA
CHFND	2C8E	CHTBL	2CA4	CKL2	316A	CKL3	3174
CKLBL	315C	CKSM	3EE6	CL1	20C9	CLA	3944
CLASS	0090	CLEAR	200C	CLERR	20D1	CLFLAG	2410
CLY	277B	CMA	3952	CMP88	25A0	CMPND	3DD2
CMT	2A61	CNCT	20B3	CNV2	24F9	CNV2HX	2698
CNV3	2505	CNV4	2509	CNV5	251D	CNVLNO	24E7
CNVN	26A7	CNVN2	26B0	CONT	2A3F	CSPACE	260C
CTABLE	206F	CTERR	366A	DASHES	3D3A	DEA	3924
DEA2	392B	DEA3	3935	DELETE	2391	DISP	20D4
DL2	23F3	DL3	240B	DLMTR	2C1F	DPGNO	31BE
DPOBJ	3E28	DPOBJ2	3E5E	DSHFLG	00BD	DSP1	20DD
DSPRTN	20E9	DTAB	2536	DTCC	20B6	EADR1	0097
EADR2	009B	ECA	3DDC	ECODE	0084	ENDFLG	3DCF
ENTSM2	2B54	ENTSM3	2B6C	ENTSYM	2B50	EOB	20C7
EOCK	319C	EOCMT	2AB1	EODEC	2D4C	EOFSYM	2E7B
EOIDC	21A9	EOIDC1	21B2	EOL	25BA	EONUM	2E30
EONUM2	2E38	EONUM3	2E3D	EOS	2C41	EOSC1	2EB4
EOSC2	2EB6	EOSC2A	2EC1	EOSC3	2ED5	EOSC3A	2EE4
EOSC4	2EEA	EOSC5	2EF4	EOSC6	2EFA	EOSCAN	2EA4
EOSYM	3C6C	EOSYM2	3C79	EOSYM3	3C8C	EOT2	2CF3
EOT3	2D0A	EOTERM	2CD8	ERCDLN	3DD8	ERCNT	3DD3
ERPTR	0062	ERR	204F	EST0	2ACE	EST1	2AD6
EST2	2ADA	EST3	2AE7	EST4	2AEB	ESYM	2AFA
ESYM0	2AF8	ESYM2	2B1B	ESYM3	2B22	ESYM3A	2B35
ESYM4	2B37	ESYM5	2B45	ESYMTB	2AB2	FASC	2D81
FASC1	2D97	FASC2	2DA4	FASC3	2DAA	FASC3A	2DBA
FASC4	2DC0	FASC4A	2DDB	FASC5	2DE9	FBA	2233
FBA2	2250	FBA3	225E	FBA4	2266	FBA5	226C
FBA6	227C	FBA7	2280	FBIN	2DF4	FDCML	2D0D
FHEX	212B	FHEX2	213C	FHEX3	2141	FHI1	0088
FHI2	0089	FHXDC	2E50	FIRST	0091	FJTBL	2CBE
FKCH	20A4	FLCHR	3F74	FLFLAG	3F73	FLO2	0087
FM1	2716	FMTEND	27AC	FNOFLO	2E41	FNUM1	2DFE
FNUM1A	2E00	FNUM3	2E08	FNUM4	2E1D	FNUM5	2E21
FOCT	2E44	FORMAT	2712	FPERR	2221	FPRM	21B9
FPRM2	21D7	FS1	26C4	FS2	26D6	FS3	26DE
FS3A	26E8	FS4	26EF	FS4B	26FA	FS4C	2704
FS5	2709	FSCAN	26BC	FSYM	2E60	FSYM2	2E6D
GL1	2593	GL2	25AD	GLCHR	25D4	GLINE	258E
HDNG	31B9	HMASK	008B	HSH	2E83	HSH1	2E89

HSL	2E9C	HXCNV	3C8F	IDB	3A34	IDBMM	3A66
IDC	2144	IDC1	214C	IDC2	2171	IDIFF	009D
ILBL	31B1	ILC	00B6	ILOP	2801	IMVAL	00B9
INA	393B	INITLZ	209B	INST	2A52	JM2	216E
JP1B	3080	JP2B	3303	JTABLE	207D	KBFR	253E
LABEL	2A4A	LDA	3B21	LENGTH	00B3	LFLAG	00A9
LIST	2425	LIST1	2430	LIST3	245C	LMASK	008A
LNCNT	0078	LNO	2A36	LOC	2A3A	LPCNV1	3CF7
LPCNV2	3CFF	LPCNV3	3D07	LPCNV4	3D0F	LPNT	3CAC
LPNT1	3CB3	LPNT1A	3CBC	LPNT2	3CCD	LPNT3	3CE9
LPNT4	3CEC	LPP	00BA	LPPCNT	3D38	LPRTS	3CBB
MDW	31B8	MEM1	2498	MEM2	24A2	MEM3	24BA
MEM4	24C1	MEMORY	245F	MERR	249F	MLIMIT	24CE
MLTRL	2ECD	MON	2000	MON1	2025	MON1A	202B
MON2	203B	MON3	2047	MON4	2057	MONPTR	0060
MPRT	3D2B	MPRT2	3D35	MPRTB	3D2C	MPTRA	3B15
MSPACE	2413	NEMSG	2FC3	NF0	2D68	NF1	2D70
NF2	2D7C	NOEND	2FB1	NOFLO	2D4F	NOFLO1	2D51
NOFLO2	2D57	NOFND	2BD3	NOPROG	2FA7	NP2	2FAB
NPMSG	2FB8	NSYMS	009F	NTERMS	00AB	NXCNV	3C9D
NXCNV2	3CA6	NXTCHR	007A	NXTSYM	2C12	OBFLAG	00AA
OBFPTR	0064	OBFR	3EEC	OBJCNT	3EE5	OBJPTR	0076
OCNT	3EE4	OCODE	3DB1	ONPD	31F2	OPDSCN	2C5A
OPND	2A57	OPR8R	008C	OPSC2	2C6E	OPSC3	2C70
OPSTAT	008E	OPT2	276A	OPTBL	2806	OPTST	2756
OUT	20A1	P1$	3127	P1ASC	30FE	P1ASC2	3110
P1ASC3	3113	P1ASC4	3119	P1AST	311E	P1B	2FE1
P1C	3010	P1CKL	3130	P1CKL1	3136	P1CKL2	313A
P1CKL3	3141	P1D	3016	P1DBL	30F6	P1DTA	30E8
P1DTA2	30EC	P1END	3089	P1END2	308E	P1EQU	30AF
P1ERFL	00B5	P1ERR	30C6	P1ESP	3083	P1JTBL	302F
P1JUNK	3049	P1ORG	306B	P1REM	30C3	P1RES	30D7
P1SPD	3071	P1SUM	304D	P1TTL	3098	P1TTL1	30A2
P1TTL2	30A5	P2$	33B0	P2ASC	3387	P2ASC2	3399
P2ASC3	339C	P2ASC4	33A2	P2AST	33A7	P2B	3242
P2C	3282	P2C1	33CE	P2C10	33DF	P2C1A	33E3
P2C1B	33EC	P2C1C	33F7	P2C1S	33D4	P2C2	33FA
P2C2A	33FC	P2C3	3402	P2C3A	3406	P2C4	340A
P2C5	33FA	P2C6	3410	P2C7	341B	P2C8	341B
P2C9	3410	P2CA	3410	P2CB	33FA	P2CC	33FA
P2CD	3423	P2CD5	3447	P2CDA	3442	P2CDB	3444
P2CDC	344D	P2CDC1	344F	P2CDD	345E	P2CKL	33B9
P2CKL1	33BF	P2CKL2	33CA	P2D	328F	P2DBL	337F
P2DTA	3371	P2DTA2	3375	P2END	330C	P2END2	3311
P2EQU	3331	P2ERR	3348	P2ESP	3314	P2JTBL	32BE
P2JUNK	32D8	P2ORG	32F0	P2PSTB	32A4	P2REM	3345
P2RES	335C	P2SPD	3306	P2SUM	32E0	P2TITL	331A
P2TTL1	3324	P2TTL2	3327	P3$	3654	P3A	348B
P3A0	3493	P3A2	34AF	P3A3	34B9	P3A4	34D1
P3A5	34E7	P3A6	34F3	P3A7	3502	P3A8	3513
P3ASC	3637	P3ASC2	3641	P3AST	364D	P3AST2	364F
P3C1	3673	P3C1A	36A4	P3C1B	36B6	P3C1C	36D9
P3C1D	36DC	P3C1E	36DE	P3C1IM	36E4	P3C1IN	36FB
P3C1J	3710	P3C1N	3675	P3C2	371A	P3C2A	3730
P3C2B	373C	P3C2J	3753	P3C3	3757	P3C4	3764
P3C4A	377D	P3C4B	378E	P3C4E	378A	P3C5	37A7
P3C5A	37B7	P3C5B	37BD	P3C5C	37CD	P3C5E	37B1
P3C5F	37B4	P3C6	37DA	P3C6A	3800	P3C6E	37E9
P3C6IM	3803	P3C7	3814	P3C7A	3824	P3C7B	382D

P3C7IN	383F	P3C8	384C	P3C9	3853	P3C9A	3855
P3C9B	386D	P3C9C	3883	P3C9E	389F	P3C9IM	3890
P3C9R	3880	P3CA	38A2	P3CB	38A6	P3CBA	38A8
P3CBB	38BB	P3CBC	38D6	P3CBD	38D9	P3CBER	38B8
P3CC	38E6	P3CD	38EA	P3CD1	3901	P3CD2	390C
P3CDA	3912	P3DBL	35FB	P3DBL2	3603	P3DBL3	362E
P3DBL4	3631	P3DBLA	3629	P3DBLB	360F	P3DTA	35D6
P3DTA2	35DD	P3DTA3	35E9	P3END	35AA	P3EQU	35A2
P3ESP	3598	P3JTBL	354A	P3JUNK	3528	P3LNTH	3DD4
P3NXT	3AF4	P3ORG	3564	P3ORG3	357C	P3PRT	3A9D
P3PRTB	3AB4	P3PRTC	3AC6	P3PRTD	3AE3	P3PRTE	3B09
P3PRTF	3B34	P3PRTG	3B4E	P3PRTH	3B58	P3PRTI	3B60
P3PRTJ	3B72	P3PRTX	3B6A	P3PRTZ	3AD9	P3PSTB	3530
P3REM	35B3	P3RES	35C0	P3SPD	357F	P3T	3DD0
P3TITL	348B	PAGE	00A8	PASS1	2FCC	PASS2	3238
PASS3	3462	PAUSCT	00BC	PCHLC1	3B49	PCHOB1	3DE9
PCHOB2	3DEA	PCHOB3	3DFE	PCHOB4	3E10	PCHOBJ	3DE5
PCHRES	3E72	PCHRS2	3E89	PCHRS3	3E93	PCHRS4	3EA8
PCHRS5	3ECE	PIAS	3BB3	PRBLN	3D23	PROMPT	2099
PRT	20B0	PSCT	3D39	PSYM1	3B92	PSYM1A	3B9A
PSYM1B	3BA8	PSYM1C	3BC1	PSYM2	3BE4	PSYM2A	3BF2
PSYM3	3BF6	PSYM4	3C4A	PSYM5	3C4F	PSYM6	3C61
PSYMTB	3B7A	PW	00BB	RCDLNT	3EEB	RCDORG	3EE8
RDBYTE	20AA	READ	2620	READ2	2639	READ3	2641
READ4	2662	READ5	2671	RERR	2695	RHDR	20A7
RKB	20EA	RKB2	20F8	RKB3	2105	RKB4	210D
RKB5	2115	RKBRTN	211B	RKBT	2128	RKBX	2129
RKBY	212A	RM0	3EE7	RM1	3F6F	RP	2C43
RSLNTH	3DD6	SB	20B9	SCOUNT	00A7	SECOND	0093
SKEL	008F	SPDFLG	00B2	SPLC	00B0	SPOMS	3148
SPOV3	3D8A	SPOVMS	3D9D	SPVXFR	3B46	SR1	27C9
SR2	27D3	SR3	27DB	SRCLNO	00AE	SRCOP	27B0
SRCOP1	27BE	SRCOP2	27EE	SRCOP3	27F3	SRCOP4	27FE
SRCST2	2B9D	SRCST3	2BAF	SRCST4	2BC9	SRCST5	2BDA
SRCST6	2BE9	SRCSTB	2B8D	SRST3A	2BBB	STERCH	3660
STL2	2F3F	STL2A	2F4D	STL4	2F5D	STLINE	3F99
STLIST	2F2D	STOFLO	2B6D	STOFM	2B77	STSPTR	2F96
SUBTRA	2534	SVOP	2C9A	SVPTR	2BD8	SYMBOL	2C53
SYMCNT	00A1	SYMPTR	00A3	SYMTAB	00A5	SYMWD	3CAA
T1CKSM	3F77	T1LNTH	3F75	T1ORG	3F70	TCA	3960
TCD	2224	TCDR	2232	TCRLF	209E	TEMS	3211
TITLE	31C8	TL2	25DC	TLFLAG	00B1	TLM	25DF
TMN8R	008D	TSHF	2DEF	TSRC	2C78	TSRC1	2C85
TSRC2	2C86	TULONG	25D5	TVAL	0085	TYPE	3EEA
TYPE1	3F72	UERR	3658	UNDFT	2E7E	UNDFTP	2DEC
UPDPTR	2C47	UPDRTN	2C52	UPTR1	0066	UPTR2	0068
UPTR3	006A	UPTR4	006C	UPTR5	006E	UPTR6	0070
UPTR7	0072	UPTR8	0074	UT1	007C	UT2	007E
UT3	0080	UT4	0082	WCNT	3CAB	WLDR	3ECF
WOBJR	20BF	WOBL	20BC	WOBTR	20C2	WRITE	25EE
WRTAPE	20AD	WTRLR	3ED9	XCL	2792	XMTC2	2785
XMTC3	2790	XMTCMT	277D	XMTOP	273D		

APPENDIX B

ASSEMBLY LISTING OF APPLE II INPUT/OUTPUT SUBROUTINES FOR THE ASSEMBLER

```
002                                                  COMMAND TABLE PATCH TO ALLOW
003                                                  TRANSFER TO DEBUG
004                             ORG   $2078
005 2078 A4                     DATA  '$'
006                             ORG   $208F
007 208F FF0F                   DBL   $FFF
008
009                                                  JUMP TABLE PATCH
010
011                             ORG   $209B
012 209B 4C0040       J1        JMP   INITLZ
013 209E 4C3C40       J2        JMP   TCRLF
014 20A1 4C4340       J3        JMP   OUT
015 20A4 4C5940       J4        JMP   FKCH
016 20A7 4C9A43       J5        JMP   RHDR
017 20AA 4CCF43       J6        JMP   RDBYTE
018 20AD 4CB342       J7        JMP   WRTAPE
019 20B0 4C0341       J8        JMP   PRDVC1
020 20B3 4C8D41       J9        JMP   CNCT
021 20B6 4C5643       J10       JMP   DTCC
022 20B9 4C6643       J11       JMP   SB
023 20BC 4C9044       J12       JMP   CSTLDR
024 20BF 4C9C44       J13       JMP   CSTWRT
025 20C2 4CB844       J14       JMP   CSTTRL
026 20C5 EF44                   DBL   EOIO+1         BUFFER ORIGIN
027
028                             ORG   $4000
029                                                  INITIALIZE APPLE
030 4000 8D54C0       INITLZ    STA   $C054          SET PRIMARY SCREEN
031 4003 8D51C0                 STA   $C051          SET TEXT MODE
032 4006 A90085E3               CDB   IOPTR1,SCREEN
033 400E A204                   LDX   #4
034 4010 A9A0                   LDA   #' '
035 4012 A000                   LDY   #0
036 4014 84BA                   STY   LPP
037 4016 84BD                   STY   DSHFLG
038 4018 84EC                   STY   LPTR           INITIALIZE LINE POINTER
039 401A 91E3         INT2      STA   *IOPTR1,Y      CLEAR SCREEN
040 401C C8                     INY
041 401D D0FB                   BNZ   INT2
042 401F E6E4                   INC   IOPTR1+1
043 4021 CA                     DEX
044 4022 D0F6                   BNZ   INT2
045 4024 A9FF                   LDA   #$FF
046 4026 8D02C1                 STA   TTYOUT
047 4029 A9038DB1               CDB   J8+1,PRDVC1    SET TO SCREEN
048 4033 A928                   LDA   #40
049 4035 85BB                   STA   PW
050 4037 A90C                   LDA   #12
051 4039 85BC                   STA   PAUSCT
052 403B 60                     RTS
053
054                                                  SCROLL SCREEN 2 LINES
055 403C 20B840       TCRLF     JSR   SCRADV
056 403F 20B840                 JSR   SCRADV
057 4042 60                     RTS
058
059                                                  OUTPUT CHAR IN A TO SCREEN
060 4043 84E0         OUT       STY   IOT1
```

```
061 4045 48              PHA
062 4046 A4EC            LDY   LPTR
063 4048 995007          STA   NTLINE,Y  CHARACTER TO SCREEN
064 404B C8              INY
065 404C C028            CPY   #40       CHECK END OF LINE
066 404E D002            BNE   OUT2
067 4050 A080            LDY   #$80
068 4052 84EC      OUT2  STY   LPTR      SAVE POINTER
069 4054 18              CLC
070 4055 A4E0      OUT3  LDY   IOT1
071 4057 68              PLA
072 4058 60              RTS
073
074                                    GET CHARACTER FROM KEYBOARD
075                                    ECHO TO SCREEN
076 4059 84EB      FKCH  STY   KBY       SAVE Y
077 405B A9A0            LDA   #' '      SET UP CURSOR
078 405D A4EC            LDY   LPTR
079 405F 995007          STA   NTLINE,Y
080 4062 A9C0      FKCH0 LDA   #$C0      CURSOR TIME CONSTANT
081 4064 85E0            STA   IOT1
082 4066 85E1            STA   IOT2      SET CURSOR TIMEOUT
083 4068 B95007          LDA   NTLINE,Y  INVERT CURSOR
084 406B 490E            EOR   #$E
085 406D 995007    FKCH1 STA   NTLINE,Y
086 4070 AD00C0    FKCH2 LDA   KB
087 4073 300A            BMI   FKCH3
088 4075 E6E0            INC   IOT1
089 4077 D0F7            BNZ   FKCH2
090 4079 E6E1            INC   IOT2
091 407B D0F3            BNZ   FKCH2
092 407D F0E3            BZ    FKCH0
093 407F 8D10C0    FKCH3 STA   KBS       CLEAR STROBE
094 4082 85E1            STA   IOT2
095 4084 C988            CMP   #$88      BACKSPACE?
096 4086 F017            BEQ   FKCHB
097 4088 C98D            CMP   #$8D
098 408A D002            BNE   FKCH3A
099 408C A9A0            LDA   #' '
100 408E 995007    FKCH3A STA  NTLINE,Y
101 4091 C8              INY
102 4092 C028            CPY   #40       END OF LINE
103 4094 D002            BNE   FKCH4
104 4096 A080            LDY   #$80      YES, SET TO BOTTOM LINE
105 4098 84EC      FKCH4 STY   LPTR      SAVE POINTER
106 409A A4EB            LDY   KBY       RESTORE Y
107 409C A5E1            LDA   IOT2
108 409E 60              RTS             AND RETURN
109 409F A9A0      FKCHB LDA   #' '
110 40A1 995007          STA   NTLINE,Y  CLEAR CURSOR
111 40A4 88              DEY             BACK UP POINTER
112 40A5 C0FF            CPY   #-1       OVER START OF LINE?
113 40A7 D003            BNE   FKCHB1
114 40A9 C8              INY             YES, RESET
115 40AA F0EC            BZ    FKCH4
116 40AC C07F      FKCHB1 CPY  #$7F      FROM 2ND LINE TO 1ST?
117 40AE D002            BNE   FKCHB2
118 40B0 A027            LDY   #39       YES, RESET POINTER
119 40B2 995007    FKCHB2 STA  NTLINE,Y
```

```
120 40B5 4C9840             JMP   FKCH4
121
122                                       SCROLL SCREEN ONE LINE
123 40B8 A900       SCRADV  LDA   #0
124 40BA 48                 PHA
125 40BB 20EC40             JSR   LNADR
126 40BE A5E3       SCRAD2  LDA   IOPTR1
127 40C0 85E5               STA   IOPTR2
128 40C2 A5E4               LDA   IOPTR1+1
129 40C4 85E6               STA   IOPTR2+1
130 40C6 68                 PLA
131 40C7 186901             INA
132 40CA C918               CMP   #24
133 40CC F00F               BEQ   SCRTN
134 40CE 48                 PHA
135 40CF 20EC40             JSR   LNADR
136 40D2 A027               LDY   #39
137 40D4 B1E3       SCRAD3  LDA   *IOPTR1,Y
138 40D6 91E5               STA   *IOPTR2,Y
139 40D8 88                 DEY
140 40D9 10F9               BPL   SCRAD3
141 40DB 30E1               BMI   SCRAD2
142 40DD A027       SCRTN   LDY   #39
143 40DF A9A0               LDA   #' '
144 40E1 99D007     SCRAD4  STA   BLINE,Y   BLANK BOTTOM LINE
145 40E4 88                 DEY
146 40E5 10FA               BPL   SCRAD4
147 40E7 A900               LDA   #0
148 40E9 85EC               STA   LPTR
149 40EB 60                 RTS
150
151                                       COMPUTE BEGINNING ADDRESS
152                                       OF LINE WHOSE NUMBER,
153                                       0-23 IS IN A ON ENTRY
154 40EC 48         LNADR   PHA
155 40ED 2907               AND   #7
156 40EF 0908               ORA   #8
157 40F1 85E4               STA   IOPTR1+1
158 40F3 68                 PLA
159 40F4 2918               AND   #$18
160 40F6 85E3               STA   IOPTR1
161 40F8 0A                 ASLA
162 40F9 0A                 ASLA
163 40FA 05E3               ORA   IOPTR1
164 40FC 0A                 ASLA
165 40FD 66E4               ROR   IOPTR1+1
166 40FF 6A                 RORA
167 4100 85E3               STA   IOPTR1
168 4102 60                 RTS
169                 SCREEN  EQU   $400
170                 NTLINE  EQU   $750
171                 BLINE   EQU   $7D0
172                 KB      EQU   $C000
173                 KBS     EQU   $C010
174                 *
175                 * PRINT LINE WHOSE ADDRESS IS IN A-X ON ENTRY
176                 * ON ASSIGNED OUTPUT DEVICE.
177                 *
178                                       APPLE SCREEN OUTPUT
```

```
179 4103 85E7        PRDVC1  STA   IOPTR3
180 4105 86E8                STX   IOPTR3+1  SAVE ADDRESS
181 4107 84E2                STY   IOT3
182 4109 20B840              JSR   SCRADV     ROLL SCREEN
183 410C A000                LDY   #0         CLEAR INDEX
184 410E B1E7        PRDV1A  LDA   *IOPTR3,Y
185 4110 C98D                CMP   #$8D
186 4112 F052                BEQ   BEQEOP
187 4114 204340              JSR   OUT
188 4117 C8                  INY
189 4118 D0F4                BNZ   PRDV1A
190                                           LINE PRINTER OUTPUT
191 411A 85E7        PRDVC2  STA   IOPTR3
192 411C 86E8                STX   IOPTR3+1
193 411E 84E2                STY   IOT3
194 4120 A000                LDY   #0
195 4122 A200                LDX   #0
196 4124 B1E7        PRDV2A  LDA   *IOPTR3,Y
197 4126 9DB0C0              STA   PRTR,X
198 4129 48                  PHA
199 412A BDC1C3      PRDV2B  LDA   PRFLAG,X
200 412D 30FB                BMI   PRDV2B
201 412F 68                  PLA
202 4130 C8                  INY
203 4131 C98D                CMP   #$8D
204 4133 D0EF                BNE   PRDV2A
205 4135 A98A                LDA   #$8A
206 4137 9DB0C0              STA   PRTR,X
207 413A BDC1C3      PRDV2C  LDA   PRFLAG,X
208 413D 30FB                BMI   PRDV2C
209 413F 1032                BPL   EOPRT
210 4141 0000        TC      DBL   0
211                                           TELETYPE OUTPUT
212 4143 85E7        PRDVC3  STA   IOPTR3
213 4145 86E8                STX   IOPTR3+1
214 4147 84E2                STY   IOT3
215 4149 A048                LDY   #72
216 414B 8C7641              STY   CHCNT
217 414E A000                LDY   #0
218 4150 8C02C1              STY   TTYOUT
219 4153 A98D                LDA   #$8D
220 4155 207741              JSR   OUTTTY
221 4158 A98A                LDA   #$8A
222 415A 207741              JSR   OUTTTY
223 415D A9FF                LDA   #$FF
224 415F 207741              JSR   OUTTTY
225 4162 B1E7        PRDV3A  LDA   *IOPTR3,Y
226 4164 C98D                CMP   #$8D
227 4166 F00B        BEQEOP  BEQ   EOPRT
228 4168 207741              JSR   OUTTTY
229 416B CE7641              DEC   CHCNT
230 416E F003                BZ    EOPRT
231 4170 C8                  INY
232 4171 D0EF                BNZ   PRDV3A
233 4173 A4E2        EOPRT   LDY   IOT3
234 4175 60                  RTS
235                  PRTR    EQU   $C0B0
236                  PRFLAG  EQU   $C3C1     PRINTER READY FLAG
237 4176 00          CHCNT   ***
```

```
                                                    $27 IF READY
                                                    $FE IF NOT READY
                        TTYST   EQU   $C101
                        TTYOUT  EQU   $C102
                        TTYDTA  EQU   $C100
                        MON     EQU   $60
                        ERR     EQU   $62
4177 8D02C1             OUTTTY  STA   TTYOUT
417A AD01C1             OTTY2   LDA   TTYST
417D 6A                         RORA
417E 90FA                       BCC   OTTY2
4180 60                         RTS
4181 A98A               PR2LF   LDA   #$8A       SUPPLY LINE FEED
4183 9DB0C0                     STA   PRTR,X     TO PRINTER
4186 BDC1C3             PR2LF2  LDA   PRFLAG,X
4189 30FB                       BMI   PR2LF2
418B 10E6                       BPL   EOPRT
                        *
                        * DEVICE CONNECT
                        *
418D 205940             CNCT    JSR   FKCH       GET L, R OR W
4190 AA                         TAX               SAVE
4191 205940                     JSR   FKCH       GET DEVICE #
4194 48                         PHA               SAVE ON STACK
4195 8A                         TXA               SAVE ON STACK
4196 48                         PHA
4197 205940                     JSR   FKCH       GET PERIOD
419A C9AE                       CMP   #'.'
419C D017                       BNE   CERR       TILT
419E 68                         PLA               GET DEVICE CODE
419F C9CF                       CMP   #'O'       OBJECT DEVICE?
41A1 D003                       BNE   CNCT1
41A3 4C6842                     JMP   CNOBJ
41A6 C9CC               CNCT1   CMP   #'L'
41A8 F00E                       BEQ   CNLIST
41AA C9D2                       CMP   #'R'       TAPE READ DEVICE
41AC F060                       BEQ   CNREAD
41AE C9D7                       CMP   #'W'       TAPE WRITE DEVICE
41B0 D003                       BNE   CERR
41B2 4C4542                     JMP   CNWRT
41B5 6C6200             CERR    JMP   *ERR
                                                    SET LISTING DEVICE
41B8 68                 CNLIST  PLA
41B9 C9B1                       CMP   #'1'
41BB D01B                       BNE   CNL2
41BD A9038DB1                   CDB   J8+1,PRDVC1 APPLE SCREEN
41C7 A928                       LDA   #40        SET SCREEN WIDTH
41C9 85BB               CNL1    STA   PW
41CB A900                       LDA   #0
41CD 85BD                       STA   DSHFLG
41CF 85BA                       STA   LPP
41D1 A90C                       LDA   #12
41D3 85BC                       STA   PAUSCT
41D5 6C6000                     JMP   *MON
41D8 C9B2               CNL2    CMP   #'2'
41DA D01D                       BNE   CNL3
41DC A91A8DB1                   CDB   J8+1,PRDVC2 LINE PRINTER
41E6 A950                       LDA   #80        SET PRINTER WIDTH
41E8 85BB               CNL2A   STA   PW
```

```
297 41EA A93C               LDA   #60
298 41EC 85BA               STA   LPP
299 41EE A900               LDA   #0
300 41F0 85BD               STA   DSHFLG
301 41F2 A900               LDA   #0
302 41F4 85BC               STA   PAUSCT
303 41F6 6C6000             JMP   *MON
304 41F9 C9B3       CNL3    CMP   #'3'
305 41FB D0B8               BNE   CERR
306 41FD A9438DB1           CDB   J8+1,PRDVC3 TTY
307 4207 A948               LDA   #72       SET TTY WIDTH
308 4209 D0DD               BNZ   CNL2A
309 420B 6C6000             JMP   *MON      RETURN TO MONITOR
310                                         SET TAPE READ DEVICE
311 420E 68         CNREAD  PLA
312 420F C9B1               CMP   #'1'
313 4211 D017               BNE   CNR2
314 4213 A99A8DA8           CDB   J5+1,RHDR
315 421D A9CF8DAB           CDB   J6+1,RDBYTE
316 4227 6C6000             JMP   *MON
317 422A C9B2       CNR2    CMP   #'2'
318 422C D087       CNR3    BNE   CERR
319 422E A9018DA8           CDB   J5+1,RTHDR TTY TAPE INPUT
320 4238 A9068DAB           CDB   J6+1,RTBYTE
321 4242 6C6000             JMP   *MON
322                                         SET TAPE WRITE DEVICE
323 4245 68         CNWRT   PLA
324 4246 C9B1               CMP   #'1'
325 4248 D00D               BNE   CNW2
326 424A A9B38DAE           CDB   J7+1,WRTAPE CASSETTE OUTPUT
327 4254 6C6000             JMP   *MON
328 4257 C9B2       CNW2    CMP   #'2'
329 4259 D0D1               BNE   CNR3
330 425B A9288DAE           CDB   J7+1,WRTTY TTY OUTPUT
331 4265 6C6000             JMP   *MON
332                                         CONNECT OBJECT DEVICE
333 4268 68         CNOBJ   PLA
334 4269 C9B1               CMP   #'1'
335 426B F025               BEQ   CNCSTE
336 426D C9B2               CMP   #'2'
337 426F D0BB               BNE   CNR3
338 4271 A95C8DBD           CDB   J12+1,TTYLDR
339 427B A96A8DC0           CDB   J13+1,TTYPCH
340 4285 A98C8DC3           CDB   J14+1,TTYTR
341 428F 6C6000             JMP   *MON
342 4292 A9908DBD   CNCSTE  CDB   J12+1,CSTLDR
343 429C A99C8DC0           CDB   J13+1,CSTWRT
344 42A6 A9B88DC3           CDB   J14+1,CSTTRL
345 42B0 6C6000             JMP   *MON
346                 *
347 42B3 85E7       WRTAPE  STA   IOPTR3
348 42B5 86E8               STX   IOPTR3+1
349 42B7 A9A885E9           CDB   IOPTR4,-7000
350 42BF 20B840             JSR   SCRADV
351 42C2 A200               LDX   #0
352 42C4 BD4743     WRTP1   LDA   TONM,X
353 42C7 E8                 INX
354 42C8 C98D               CMP   #$8D
355 42CA F006               BEQ   WRTP2
```

```
356 42CC 204340            JSR   OUT
357 42CF 4CC442            JMP   WRTP1
358 42D2 206643    WRTP2   JSR   SB
359 42D5 38        WRCD    SEC
360 42D6 201743            JSR   WRBIT
361 42D9 208143            JSR   ICNT
362 42DC B005              BCS   WZEROS
363 42DE A22C              LDX   #44
364 42E0 4CD542            JMP   WRCD
365 42E3 A000      WZEROS  LDY   #0
366 42E5 18                CLC
367 42E6 A225              LDX   #37
368 42E8 201743            JSR   WRBIT
369 42EB A222      WDATA   LDX   #34
370 42ED A59D              LDA   IDIFF
371 42EF 85E9              STA   IOPTR4
372 42F1 A59E              LDA   IDIFF+1
373 42F3 85EA              STA   IOPTR4+1
374 42F5 206E43    WDATA1  JSR   FETCH
375 42F8 A008              LDY   #8
376 42FA EA                NOP
377 42FB EA                NOP
378 42FC EA                NOP
379 42FD 2A        WDATA2  ROLA
380 42FE 201743            JSR   WRBIT
381 4301 A22C              LDX   #44
382 4303 88                DEY
383 4304 D0F7              BNZ   WDATA2
384 4306 208143            JSR   ICNT
385 4309 B008              BCS   WRCEND
386 430B A218              LDX   #24
387 430D EA                NOP
388 430E EA                NOP
389 430F EA                NOP
390 4310 4CF542            JMP   WDATA1
391 4313 88        WRCEND  DEY
392 4314 D0FD              BNZ   WRCEND
393 4316 60                RTS
394                *
395                * WRITE BIT
396                *
397 4317 B015      WRBIT   BCS   WRONE
398 4319 CA        WR1     DEX
399 431A D0FD              BNZ   WR1
400 431C EA                NOP
401 431D EA                NOP
402 431E AE20C0            LDX   TPOUT
403 4321 A230              LDX   #48
404 4323 CA        WR3     DEX
405 4324 D0FD              BNZ   WR3
406 4326 EA                NOP
407 4327 4C2A43            JMP   WR4
408 432A AE20C0    WR4     LDX   TPOUT
409 432D 60                RTS
410 432E CA        WRONE   DEX
411 432F D0FD              BNZ   WRONE
412 4331 EA                NOP
413 4332 EA                NOP
414 4333 4C3643            JMP   WRONE1
```

```
415 4336 A231        WRONE1  LDX   #49
416 4338 CA          WRONE2  DEX
417 4339 D0FD                BNZ   WRONE2
418 433B AE20C0              LDX   TPOUT
419 433E A263                LDX   #99
420 4340 CA          WRONE3  DEX
421 4341 D0FD                BNZ   WRONE3
422 4343 AE20C0              LDX   TPOUT
423 4346 60                  RTS
424                  *
425 4347 D4C1D0C5    TONM    ASC   'TAPE ON, SPACE'
426 4355 8D                  DATA  $8D
427                  *
428                  * DETECT CONTROL-
429                  *
430 4356 AD00C0      DTCC    LDA   KB
431 4359 3002                BMI   DTCC2
432 435B 18          DTCC1   CLC
433 435C 60                  RTS
434 435D 8D10C0      DTCC2   STA   KBS       CLEAR STROBE
435 4360 C983                CMP   #$83
436 4362 D0F7                BNE   DTCC1
437 4364 38                  SEC
438 4365 60                  RTS
439                  *
440                  * WAIT FOR SPACE BAR
441                  *
442 4366 205940      SB      JSR   FKCH
443 4369 C9A0                CMP   #' '
444 436B D0F9                BNE   SB
445 436D 60                  RTS
446                  *
447                  * FETCH MEMORY WORD - CONSTANT 35 US
448                  * TIME INCLUDING CALLING JSR
449                  *
450 436E B1E7        FETCH   LDA   *IOPTR3,Y
451 4370 E6E7                INC   IOPTR3
452 4372 D006                BNZ   F1
453 4374 E6E8                INC   IOPTR3+1
454 4376 EA                  NOP
455 4377 EA                  NOP
456 4378 EA          FRTN    NOP
457 4379 60                  RTS
458 437A 18          F1      CLC
459 437B 4C7E43              JMP   F2
460 437E 4C7843      F2      JMP   FRTN
461                  *
462                  * INCREMENT COUNT IN IOPTR4
463                  * CONSTANT 35 US TIME INCLUDING CALLING
464                  * JSR. CARRY RETURNS ON IF RESULT ZERO
465                  *
466 4381 18          ICNT    CLC
467 4382 E6E9                INC   IOPTR4
468 4384 D009                BNZ   IC1
469 4386 E6EA                INC   IOPTR4+1
470 4388 F00C                BZ    ICEND
471 438A 9000                BCC   IC2
472 438C EA          IC2     NOP
473 438D EA                  NOP
```

```
474 438E 60                 RTS
475 438F 9000       IC1     BCC   IC3
476 4391 9000       IC3     BCC   IC4
477 4393 48         IC4     PHA
478 4394 68                 PLA
479 4395 60                 RTS
480 4396 EA         ICEND   NOP
481 4397 EA                 NOP
482 4398 38                 SEC
483 4399 60                 RTS
484                 TPOUT   EQU   $C020
485                 TPIN    EQU   $C060
486                 *
487                 * PASS TAPE HEADER
488                 *
489 439A 20B840     RHDR    JSR   SCRADV
490 439D A200               LDX   #0
491 439F BD4743     RHF     LDA   TONM,X
492 43A2 E8                 INX
493 43A3 C98D               CMP   #$8D
494 43A5 F006               BEQ   RHW
495 43A7 204340             JSR   OUT
496 43AA 4C9F43             JMP   RHF
497 43AD 206643     RHW     JSR   SB          WAIT FOR SPACE
498 43B0 A000       RH0     LDY   #0
499 43B2 20EA43     RH1     JSR   RTRANS      GET TRANSITION
500 43B5 70FB               BVS   RH1
501 43B7 E01B               CPX   #TTIME
502 43B9 90F5               BLS   RH0
503 43BB C8                 INY
504 43BC C050               CPY   #80
505 43BE 90F2               BLS   RH1
506 43C0 20EA43     RH2     JSR   RTRANS
507 43C3 E01B               CPX   #TTIME
508 43C5 B0F9               BGE   RH2
509 43C7 20EA43             JSR   RTRANS      GET TWO SHORT
510 43CA E01B               CPX   #TTIME
511 43CC B0F2               BGE   RH2
512 43CE 60                 RTS
513                 *
514                 * READ TAPE BYTE
515                 *
516 43CF A008       RDBYTE  LDY   #8
517 43D1 A900               LDA   #0
518 43D3 48                 PHA
519 43D4 20E743     RDB2    JSR   RDBIT
520 43D7 700B               BVS   RDB4
521 43D9 68                 PLA
522 43DA E01B               CPX   #TTIME
523 43DC 2A                 ROLA
524 43DD 48                 PHA
525 43DE 88                 DEY
526 43DF D0F3               BNZ   RDB2
527 43E1 68                 PLA
528 43E2 18                 CLC
529 43E3 60         RDB3    RTS
530 43E4 68         RDB4    PLA
531 43E5 38                 SEC
532 43E6 60                 RTS
```

```
                        *
                        * READ BIT FROM TAPE
                        *
43E7 20EA43     RDBIT   JSR   RTRANS
43EA A200       RTRANS  LDX   #0
43EC E8         RT2     INX
43ED F00D               BZ    RT3       SKIP OUT ON BLANK TAPE
43EF AD60C0             LDA   TPIN
43F2 45ED               EOR   B4
43F4 10F6               BPL   RT2
43F6 45ED               EOR   B4
43F8 85ED               STA   B4
43FA B8                 CLV
43FB 60                 RTS
43FC A980       RT3     LDA   #$80
43FE 6980               ADC   #$80      SET OVERFLOW
4400 60                 RTS
                TTIME   EQU   27
                        *
                        * TELETYPE PAPER TAPE I/O
                        *
4401 A900       RTHDR   LDA   #0
4403 85E0               STA   IOT1
4405 60                 RTS
4406 AD01C1     RTBYTE  LDA   TTYST
4409 10FB               BPL   RTBYTE
440B AD00C1             LDA   TTYDTA
440E C98A               CMP   #$8A
4410 F0F4               BEQ   RTBYTE
4412 C900               CMP   #0
4414 D006               BNE   RTB2
4416 E6E0               INC   IOT1
4418 D0EC               BNZ   RTBYTE
441A 38                 SEC
441B 60                 RTS
441C 48         RTB2    PHA
441D A9F6               LDA   #-10
441F 85E0               STA   IOT1
4421 68                 PLA
4422 0980               ORA   #$80      FORCE HIGH BIT ON !!
4424 85E1               STA   IOT2
4426 18                 CLC
4427 60                 RTS
4428 85E7       WRTTY   STA   IOPTR3
442A 86E8               STX   IOPTR3+1
442C A59D               LDA   IDIFF
442E 85E9               STA   IOPTR4
4430 A59E               LDA   IDIFF+1
4432 85EA               STA   IOPTR4+1
4434 A03C               LDY   #60       HEADER LENGTH
4436 A900       WRT1    LDA   #0
4438 207741             JSR   OUTTTY
443B 88                 DEY
443C D0F8               BNZ   WRT1
443E B1E7       WRT2    LDA   *IOPTR3,Y
4440 207741             JSR   OUTTTY
4443 E6E7D002           IDB   IOPTR3
4449 E6E9D002           IDB   IOPTR4
444F D0ED               BNZ   WRT2
```

```
592 4451 A03C              LDY   #60
593 4453 A900      WRT3    LDA   #0
594 4455 207741            JSR   OUTTTY
595 4458 88                DEY
596 4459 D0F8              BNZ   WRT3
597 445B 60                RTS
598                *
599                * TTY OBJECT DRIVERS
600                *
601 445C 20DB44    TTYLDR  JSR   OBMSG
602 445F A03C      TTYLD0  LDY   #60
603 4461 A900      TTYLD1  LDA   #0
604 4463 207741            JSR   OUTTTY
605 4466 88                DEY
606 4467 D0F8              BNZ   TTYLD1
607 4469 60                RTS
608 446A 85E7      TTYPCH  STA   IOPTR3
609 446C 86E8              STX   IOPTR3+1
610 446E 84E0              STY   IOT1
611 4470 A000              LDY   #0
612 4472 B1E7      TTPCH1  LDA   *IOPTR3,Y
613 4474 E6E7D002          IDB   IOPTR3
614 447A 207741            JSR   OUTTTY
615 447D C6E0              DEC   IOT1
616 447F D0F1              BNZ   TTPCH1
617 4481 A900              LDA   #0
618 4483 207741            JSR   OUTTTY
619 4486 A900              LDA   #0
620 4488 207741            JSR   OUTTTY
621 448B 60                RTS
622 448C A046      TTYTR   LDY   #70        7" TRAILER
623 448E D0D1              BNZ   TTYLD1
624                *
625                * CASSETTE OBJECT DRIVERS
626                *
627 4490 20DB44    CSTLDR  JSR   OBMSG
628 4493 A9A885E9          CDB   IOPTR4,-7000 SET LONG LEADER
629 449B 60                RTS             FOR FIRST RECORD
630 449C 85E7      CSTWRT  STA   IOPTR3
631 449E 86E8              STX   IOPTR3+1
632 44A0 98                TYA
633 44A1 49FF1869          TCA
634 44A6 859D              STA   IDIFF
635 44A8 A9FF              LDA   #$FF
636 44AA 859E              STA   IDIFF+1
637 44AC 20D542            JSR   WRCD
638 44AF A93885E9          CDB   IOPTR4,-200 SET SHORT LEADER FOR
639 44B7 60                RTS             SUCCEEDING RECORDS
640 44B8 A9C0      CSTTRL  LDA   #<EOF      WRITE END OF FILE
641 44BA A244              LDX   #>EOF
642 44BC A004              LDY   #4
643 44BE D0DC              BNZ   CSTWRT
644 44C0 FF0000FF  EOF     DATA  $FF,0,0,$FF
645 44C4 CFC2CAC5  RDYMSG  ASC   'OBJECT DEVICE ON, SPACE'
646 44DB 20B840    OBMSG   JSR   SCRADV     ROLL SCREEN
647 44DE A000              LDY   #0
648 44E0 B9C444    OBMSG1  LDA   RDYMSG,Y
649 44E3 204340            JSR   OUT
650 44E6 C8                INY
```

```
651 44E7 C017                CPY   #23
652 44E9 D0F5                BNE   OBMSG1
653 44EB 206643              JSR   SB          WAIT FOR SPACE
654 44EE 60          EOIO    RTS
655                          SPD   $E0
656                  IDIFF   EQU   $9D
657                  LPP     EQU   $BA          LINES/PAGE
658                  PW      EQU   LPP+1        PRINTER WIDTH
659                  PAUSCT  EQU   PW+1
660                  DSHFLG  EQU   PAUSCT+1
661 00E0 00          IOT1    ***
662 00E1 00          IOT2    ***
663 00E2 00          IOT3    ***
664 00E3 0000        IOPTR1  DBL   0
665 00E5 0000        IOPTR2  DBL   0
666 00E7 0000        IOPTR3  DBL   0
667 00E9 0000        IOPTR4  DBL   0
668 00EB 00          KBY     ***
669 00EC 00          LPTR    ***
670 00ED 00          B4      ***
671 00EE                     END

****************************
* S Y M B O L   T A B L E *
****************************

B4      00ED   BEQEOP  4166   BLINE   07D0   CERR    41B5
CHCNT   4176   CNCSTE  4292   CNCT    418D   CNCT1   41A6
CNL1    41C9   CNL2    41D8   CNL2A   41E8   CNL3    41F9
CNLIST  41B8   CNOBJ   4268   CNR2    422A   CNR3    422C
CNREAD  420E   CNW2    4257   CNWRT   4245   CSTLDR  4490
CSTTRL  44B8   CSTWRT  449C   DSHFLG  00BD   DTCC    4356
DTCC1   435B   DTCC2   435D   EOF     44C0   EOIO    44EE
EOPRT   4173   ERR     0062   F1      437A   F2      437E
FETCH   436E   FKCH    4059   FKCH0   4062   FKCH1   406D
FKCH2   4070   FKCH3   407F   FKCH3A  408E   FKCH4   4098
FKCHB   409F   FKCHB1  40AC   FKCHB2  40B2   FRTN    4378
IC1     438F   IC2     438C   IC3     4391   IC4     4393
ICEND   4396   ICNT    4381   IDIFF   009D   INITLZ  4000
INT2    401A   IOPTR1  00E3   IOPTR2  00E5   IOPTR3  00E7
IOPTR4  00E9   IOT1    00E0   IOT2    00E1   IOT3    00E2
J1      209B   J10     20B6   J11     20B9   J12     20BC
J13     20BF   J14     20C2   J2      209E   J3      20A1
J4      20A4   J5      20A7   J6      20AA   J7      20AD
J8      20B0   J9      20B3   KB      C000   KBS     C010
KBY     00EB   LNADR   40EC   LPP     00BA   LPTR    00EC
MON     0060   NTLINE  0750   OBMSG   44DB   OBMSG1  44E0
OTTY2   417A   OUT     4043   OUT2    4052   OUT3    4055
OUTTTY  4177   PAUSCT  00BC   PR2LF   4181   PR2LF2  4186
PRDV1A  410E   PRDV2A  4124   PRDV2B  412A   PRDV2C  413A
PRDV3A  4162   PRDVC1  4103   PRDVC2  411A   PRDVC3  4143
PRFLAG  C3C1   PRTR    C0B0   PW      00BB   RDB2    43D4
RDB3    43E3   RDB4    43E4   RDBIT   43E7   RDBYTE  43CF
RDYMSG  44C4   RH0     43B0   RH1     43B2   RH2     43C0
RHDR    439A   RHF     439F   RHW     43AD   RT2     43EC
RT3     43FC   RTB2    441C   RTBYTE  4406   RTHDR   4401
RTRANS  43EA   SB      4366   SCRAD2  40BE   SCRAD3  40D4
SCRAD4  40E1   SCRADV  40B8   SCREEN  0400   SCRTN   40DD
TC      4141   TCRLF   403C   TONM    4347   TPIN    C060
```

TPOUT	C020	TTIME	001B	TTPCH1	4472	TTYDTA	C100
TTYLD0	445F	TTYLD1	4461	TTYLDR	445C	TTYOUT	C102
TTYPCH	446A	TTYST	C101	TTYTR	448C	WDATA	42EB
WDATA1	42F5	WDATA2	42FD	WR1	4319	WR3	4323
WR4	432A	WRBIT	4317	WRCD	42D5	WRCEND	4313
WRONE	432E	WRONE1	4336	WRONE2	4338	WRONE3	4340
WRT1	4436	WRT2	443E	WRT3	4453	WRTAPE	42B3
WRTP1	42C4	WRTP2	42D2	WRTTY	4428	WZEROS	42E3

APPENDIX C

USING THE ASSEMBLER WITH OTHER 6502 BASED COMPUTERS

The assembler whose listing appears in appendix A is entirely free of specific input/output references. All input/output is done through a jump table which begins at location $209B. To use the assembler with a specific peripheral configuration this jump table must be filled in with jump instructions leading to a set of user supplied input/output routines. There are 14 routines to be supplied. Not all of them may be appropriate to your system. If the routine is unnecessary its position in the jump table must be filled with:

```
RTS
DBL  0
```

Each of these routines must be in subroutine form and must end with an RTS instruction. Before undertaking the task of creating your custom I/O package study the Apple II I/O package in appendix B. The input/output routines and their functions are:

INITLZ $209B

This routine is a catch-all initialization routine which will be called whenever the assembler is entered at $2000. It should be used as an opportunity to initialize peripheral interfaces, clear terminal screens, etc. All registers may be used.

TCRLF $209E

TCRLF should advance the terminal one line. For teletype-like devices this means a carriage return/line feed sequence, plus whatever padding characters are need for delay or other control purposes. For CRT terminals this routine should scroll the screen up one

line. The A and Y registers may be used.

OUT $20A1

OUT should output the character in the A register to the system terminal device. The registers must return unchanged.

FKCH $20A4

FKCH reads one character from the terminal keyboard and returns it in the A register. Only the A register should be changed. Note that if the typed character is a backspace ($88) the FKCH routine is responsible for erasing the character so deleted from the screen, but the backspace must still be returned to the assembler so that the proper deletion may be made in the source line buffer. Only the A register may be changed.

RHDR $20A7

RHDR scans the source input medium, returning control to the assembler when the blank space or coded record information has been passed over, and data are about to be read. This is necessary for cassette input but can be dispensed with for media like paper tape in which timing is not critical. In this latter case it will probably be easier to build a fixed count into the paper tape input routine which will allow it to scan over blank input medium for some reasonable length. This kind of arrangement is shown in the Apple II I/O package in appendix B, line 545.

RDBYTE $20AA

This routine gets one byte from the source input medium and returns control immediately to the assembler. Carry must return off unless RDBYTE detects an end-of-file condition. In this case carry must return on. The RDBYTE and RTBYTE routines in the Apple II I/O are examples of this process for cassette and teletype.

WRTAPE $20AD

WRTAPE writes a source block onto the system source medium. The address of the first character of the block to be written is in A and X on entry, low order in A. The number of bytes to be written is in scratchpad locations IDIFF and IDIFF+1 ($9D, $9E). This is negative twos complement double precision count. WRTAPE must supply both leader and trailer if appropriate.

PRT $20B0

PRT prints the character string whose beginning address is in A and X on entry on the system listing device. The PRT routine has responsibility for detecting lines which are too long for the device. The end of the string is signaled by the appearance of carriage return. PRT must supply carriage return and line feed as required by the device.

CNCT $20B3

CNCT is entered whenever a U command is typed. CNCT may be used for any necessary purpose, but was intended as a means of switching peripheral device assignments. The ability to do this is becomes valuable in a multiperipheral system when peripherals are down and being able to switch to an auxiliary

device means the difference between working and waiting until the peripheral is fixed. There are a number of ways to accomplish peripheral switches. The Apple II I/O in appendix B shows one of them.

The device connect routines which deal with assembly listing output must supply four pieces of information to the assembler. In scratchpad locations $BA through $BD the following data must be supplied:

$BA LPP

$BA contains the number of lines per printed page. If this location contains zero there will be no pagination. This would be the case when assembling to a CRT as listing device.

$BB PW

$BB contains the printer width in characters. This must be properly filled in. The assembler uses this number to determine how many symbols can be printed across the page when printing the symbol table.

$BC PAUSCT

This is the count of the number of lines to be advanced before pausing when assembling to a screen. In the Apple II I/O this is set at 12. This location should be set to zero for hard copy devices.

$BD DSHFLG

If $BD is not zero a line of dashes will be printed as a page separator. This is a handy option when printing assembly listings on devices like teletypes whose paper comes from a continuous unperforated roll.

DTCC $20B6

This routine tests whether a key has been struck on the system terminal device. If no key has been struck it returns immediately to the assembler with carry off. If a key has been struck the character must be examined. If the character typed was not control-C, $83, control returns to the assembler with carry off. If the key struck was control-C, return with carry on. In no case should DTCC wait in a loop for a key to be struck.

SB $20B9

SB should wait in an I/O test loop until a key is struck. If the key struck was blank, $A0, return control to the assembler. If the key was not blank ignore it and re-enter the wait loop. Don't return until you get a blank.

WOBL $20BC

WOBL writes whatever prefix information is required for object records. For paper tape devices this amounts to nothing more than a 10 or 20 centimeters of blank tape. Other media like cassette tape may requiring that sychronizing information be written. The CSTLDR routine in the Apple II I/O will show how this is done.

WOBJR $20BF

WOBJR writes a binary object record onto the object output medium. The address of the record to be written is in A and X, low order in A. The count, a positive single precision number is in Y on entry. All formatting has already been done by the assembler. WOBJR should simply transfer the block beginning at the

given A & X address to the output medium. The CSTWRT routine in the Apple II I/O will give you the idea of how this is done.

WOBTR $20C2

WOBTR writes whatever record suffix information may be required for object records, e.g., an end of file mark. CSTTRL in the Apple I/O performs this function.

The double word which immediately follows the jump to your WOBTR routine specifies the beginning of the assembler source buffer. You may locate the source buffer anywhere you please, of course, but maximum utilization of memory is achieved by beginning it at the first available word after the I/O package. There is a very easy way to do this as part of the I/O package. Create a label like EOIO for the last word of your I/O package. Then make the line which follows the JMP WOBTR read:

```
DBL  EOIO+1
```

This will insure that no space is wasted between the assembler and the source buffer.

APPENDIX D

ASSEMBLY LISTING OF THE DEBUG PROGRAM

```
002 1000                        ORG  $1000
003 1000 A220        DBG        LDX  #DBS
004 1002 9A                     TXS             LOAD STACK
005 1003 20B410                 JSR  CLSCR      YES
006 1006 AD51C0      DBG2       LDA  $C051      SET MAIN SCREEN
007 1009 AD52C0                 LDA  $C052
008 100C AD54C0                 LDA  $C054
009 100F 204C11                 JSR  URD        UPDATE REGISTER DISPLAY
010 1012 205818      DBG2A      JSR  UPDW       AND WINDOW DISPLAY
011 1015 20C511                 JSR  UDSD       STACK DISPLAY
012 1018 208612                 JSR  URTS       AND "RTS" DISPLAY
013 101B ADFA14      DBG3       LDA  BPFLAG     EXIT ON BREAKPOINT?
014 101E F003                   BZ   RCL
015 1020 4C4816                 JMP  BPFAIL     YES DISASTER
016 1023 A017        RCL        LDY  #23        CLEAR COMMAND LINE
017 1025 A9A0                   LDA  #' '
018 1027 99CF07      RCL2       STA  CMLINE-1,Y
019 102A 88                     DEY
020 102B D0FA                   BNZ  RCL2
021 102D A9A3                   LDA  #'#'       "#" TO COMMAND LINE
022 102F 8DD007                 STA  CMLINE
023 1032 C8                     INY
024 1033 8CDF12                 STY  CLPTR
025 1036 20E312      RCL3       JSR  RKB        GET CHARACTER
026 1039 B024                   BCS  RCL6       CHECK FOR PERIOD
027 103B C017                   CPY  #23        OVERFLOW LINE?
028 103D D0F7                   BNE  RCL3
029 103F A9BF        RCL3A      LDA  #'?'       FILL WITH "?"
030 1041 A017                   LDY  #23
031 1043 99CF07      RCL4       STA  CMLINE-1,Y
032 1046 88                     DEY
033 1047 D0FA                   BNZ  RCL4
034 1049 8CA614      RCL4A      STY  CFLASH     DELAY
035 104C 8CA714                 STY  CFLASH+1
036 104F EEA614      RCL5       INC  CFLASH
037 1052 D0FB                   BNZ  RCL5
038 1054 EEA714                 INC  CFLASH+1
039 1057 D0F6                   BNZ  RCL5
040 1059 A220                   LDX  #DBS
041 105B 9A                     TXS
042 105C 4C2310                 JMP  RCL
043 105F ADD107      RCL6       LDA  CMLINE+1   CHECK FOR LEGAL COMMAND
044 1062 A00F                   LDY  #JTBL-CTBL
045 1064 D98610      RCL7       CMP  CTBL-1,Y
046 1067 F005                   BEQ  RCL8
047 1069 88                     DEY
048 106A D0F8                   BNZ  RCL7
049 106C F0D1                   BZ   RCL3A      COMMAND NOT IN LIST
050 106E 98          RCL8       TYA             COMPUTE JUMP TABLE
051 106F 0A                     ASLA            DISPLACEMENT
052 1070 A8                     TAY
053 1071 B99510                 LDA  JTBL-1,Y
054 1074 48                     PHA             GET ADDRESS AND SAVE
055 1075 B99410                 LDA  JTBL-2,Y   ON STACK
056 1078 48                     PHA
057 1079 A002                   LDY  #2         SET C.L. POINTER
058 107B 8CDF12                 STY  CLPTR
059 107E A900                   CLA
060 1080 8DA814                 STA  RESULT     CLEAR RESULT AREA
061 1083 8DA914                 STA  RESULT+1
```

```
062 1086 60                    RTS
063 1087 CDD2C2CA    CTBL      ASC  'MRBJFSWLVD'
064 1091 00000000              DATA 0,0,0,0,0 EXPANSION COMMANDS
065 1096 9313        JTBL      DBL  MEMORY-1
066 1098 4414                  DBL  REG-1
067 109A 2A15                  DBL  BKPT-1
068 109C FA14                  DBL  JUMP-1
069 109E AC1C                  DBL  FILL-1
070 10A0 D61C                  DBL  SEARCH-1
071 10A2 0717                  DBL  WINDOW-1
072 10A4 D41A                  DBL  LOAD-1
073 10A6 6216                  DBL  VIEW-1
074 10A8 411D                  DBL  DUMP-1
075 10AA                       RES  10          SPACE FOR JUMP ADDRESSES
076 10B4                                        CLEAR DEBUG SCREEN
077 10B4 A9008DCB    CLSCR     CDB  CLADR,SCREEN
078 10BE A9008D21              CDB  DCNT,-1024
079 10C8 A9A0                  LDA  #' '
080 10CA 8DCA10      CLS2      STA  CLS2
081 10CD             CLADR     EQU  CLS2+1
082 10CD EECB10D0              IDB  CLADR
083 10D5 EE2111D0              IDB  DCNT
084 10DD D0EB                  BNZ  CLS2
085 10DF A028                  LDY  #40         TRANSMIT REGISTER BANNER
086 10E1 B92211      SFMT      LDA  BANNER-1,Y
087 10E4 99A707                STA  SBLINE-1,Y
088 10E7 88                    DEY
089 10E8 D0F7                  BNZ  SFMT
090 10EA A9D3                  LDA  #'S'
091 10EC 8D7407                STA  NTL+36
092 10EF A9C3                  LDA  #'C'
093 10F1 8D7507                STA  NTL+37
094 10F4 A9D2                  LDA  #'R'
095 10F6 8D7607                STA  NTL+38
096 10F9 60                    RTS
097 10FA                                        ADVANCE REGISTER FILE
098 10FA A206        ADVR      LDX  #6
099 10FC A000                  LDY  #0
100 10FE B91511      ADVR2     LDA  RVCTR+6,Y
101 1101 991B11                STA  RVCTR+12,Y
102 1104 B90F11                LDA  RVCTR,Y
103 1107 991511                STA  RVCTR+6,Y
104 110A C8                    INY
105 110B CA                    DEX
106 110C D0F0                  BNZ  ADVR2
107 110E 60                    RTS
108 110F                                        REGISTER FILE
109 110F
110 110F 00          A         ***              PSEUDO A
111 1110 00          X         ***              PSEUDO X
112 1111 00          Y         ***              PSEUDO Y
113 1112 FF          S         DATA $FF         PSEUDO STACK
114 1113 00          D         ***              STACK DIFFERENCE
115 1114 04          PSW       DATA 4           PSEUDO PSW, INT'S OFF
116 1115 0000                  $$$
117 1117 0000                  $$$
118 1119 0000                  $$$
119 111B 0000                  $$$
120 111D 0000                  $$$
121 111F 0000                  $$$
```

```
1121              RVCTR   EQU  A
1121 0000         DCNT    $$$
1123 C1A0A0D8     BANNER  ASC  'A  X  Y  S  D  NVBDIZC'
1139 A0A0D3D4             ASC  '  STACK  BKPT  RTS '
114C
114C                                   SCREEN DEFINITIONS
114C
114C              SCREEN  EQU  $400
114C              SBLINE  EQU  SCREEN+$3A8
114C              RL1     EQU  SCREEN+$50
114C              NTL     EQU  SCREEN+$350
114C              FLN     EQU  SCREEN+$3D0
114C
114C                                   UPDATE REGISTER DISPLAY
114C
114C A903         URD     LDA  #3
114E 8D2412               STA  MCNT
1151 A9508DBA             CDB  DSPADR,RL1
115B A90F8DAE             CDB  RADR,RVCTR
1165 A905         URD0    LDA  #5
1167 8D2512               STA  RCNT
116A 20AD11       URD1    JSR  FCH        GET REG VALUE
116D 20C212               JSR  CNV2HX
1170 20B911               JSR  STCH
1173 8A                   TXA
1174 20B911               JSR  STCH
1177 A9A0                 LDA  #' '
1179 20B911               JSR  STCH
117C CE2512               DEC  RCNT
117F D0E9                 BNZ  URD1
1181 A907                 LDA  #7
1183 8D2512               STA  RCNT
1186 A003                 LDY  #3
1188 20AD11               JSR  FCH
118B 8D2712               STA  URDT
118E A958         URD2    LDA  #$58
1190 2E2712               ROL  URDT
1193 2A                   ROLA
1194 88                   DEY
1195 F0F7                 BZ   URD2
1197 20B911               JSR  STCH
119A CE2512               DEC  RCNT
119D D0EF                 BNZ  URD2
119F A950                 LDA  #$50
11A1 8DBA11               STA  DSPADR
11A4 EEBB11               INC  DSPADR+1
11A7 CE2412               DEC  MCNT
11AA D0B9                 BNZ  URD0
11AC 60                   RTS
11AD                                   FETCH CHARACTER
11AD ADAD11       FCH     LDA  FCH
11B0              RADR    EQU  FCH+1
11B0 EEAE11D0             IDB  RADR
11B8 60                   RTS
11B9                                   STORE CHARACTER
11B9 8DB911       STCH    STA  STCH
11BC              DSPADR  EQU  STCH+1
11BC EEBA11D0             IDB  DSPADR
11C4 60                   RTS
11C5
```

```
182 11C5                                        UPDATE STACK DISPLAY
183 11C5
184 11C5 A9688DBA   UDSD    CDB  DSPADR,RL1+24
185 11CF AD2612             LDA  TUSTK
186 11D2 8DFD11             STA  STKADR
187 11D5 A908               LDA  #8
188 11D7 8D2412             STA  MCNT
189 11DA A9B1       UDSD1   LDA  #'1'
190 11DC 20B911             JSR  STCH
191 11DF ADFD11             LDA  STKADR
192 11E2 20C212             JSR  CNV2HX
193 11E5 20B911             JSR  STCH
194 11E8 8A                 TXA
195 11E9 20B911             JSR  STCH
196 11EC A0A0               LDY  #' '
197 11EE ADFD11             LDA  STKADR
198 11F1 CD1211             CMP  USTK
199 11F4 D002               BNE  UDSD2
200 11F6 A0BE               LDY  #'>'       SET ">" TO MARK
201 11F8                                    STACK POSITION
202 11F8 98         UDSD2   TYA
203 11F9 20B911             JSR  STCH
204 11FC AD0001     UDSD3   LDA  $100
205 11FF            STKADR  EQU  UDSD3+1
206 11FF 20C212             JSR  CNV2HX
207 1202 20B911             JSR  STCH
208 1205 8A                 TXA
209 1206 20B911             JSR  STCH
210 1209 CEFD11             DEC  STKADR
211 120C ADBA11             LDA  DSPADR
212 120F 2980               AND  #$80
213 1211 0968               ORA  #$68
214 1213 186980             ADD  #$80
215 1216 8DBA11             STA  DSPADR
216 1219 9003               BCC  UDSD4
217 121B EEBB11             INC  DSPADR+1
218 121E CE2412     UDSD4   DEC  MCNT
219 1221 D0B7               BNZ  UDSD1
220 1223 60                 RTS
221 1224 00         MCNT    ***
222 1225 00         RCNT    ***
223 1226            USTK    EQU  S
224 1226 FF         TUSTK   DATA $FF
225 1227 00         URDT    ***
226 1228
227 1228                                    UPDATE BREAKPOINT HISTORY
228 1228
229 1228 A96E8D44   UBPD    CDB  UBA1,NTL+30
230 1232 A9EE8D47           CDB  UBA2,FLN+30
231 123C A907               LDA  #7
232 123E 8D2412             STA  MCNT
233 1241 A004       UBP0    LDY  #4
234 1243 B94312     UBP1    LDA  UBP1,Y
235 1246            UBA1    EQU  UBP1+1
236 1246 994612     UBP2    STA  UBP2,Y
237 1249            UBA2    EQU  UBP2+1
238 1249 88                 DEY
239 124A D0F7               BNZ  UBP1
240 124C AD4412             LDA  UBA1
241 124F 38E980             SUB  #$80
```

```
242 1252 8D4412             STA  UBA1
243 1255 B003               BCS  UBP3
244 1257 CE4512             DEC  UBA1+1
245 125A AD4712      UBP3   LDA  UBA2
246 125D 38E980             SUB  #$80
247 1260 8D4712             STA  UBA2
248 1263 B003               BCS  UBP4
249 1265 CE4812             DEC  UBA2+1
250 1268 CE2412      UBP4   DEC  MCNT
251 126B D0D4               BNZ  UBP0
252 126D AD7B15             LDA  BPA+1
253 1270 20C212             JSR  CNV2HX
254 1273 8D6F04             STA  RL1+31
255 1276 8E7004             STX  RL1+32
256 1279 AD7A15             LDA  BPA
257 127C 20C212             JSR  CNV2HX
258 127F 8D7104             STA  RL1+33
259 1282 8E7204             STX  RL1+34
260 1285 60                 RTS
261 1286
262 1286                                 UPDATE "RTS" DISPLAY
263 1286
264 1286 AD1211      URTS   LDA  USTK
265 1289 186901             INA
266 128C 8D9612             STA  FST1+1
267 128F 186901             INA
268 1292 8D9C12             STA  FST2+1
269 1295 AD0001      FST1   LDA  $100
270 1298 8D2111             STA  DCNT
271 129B AD0001      FST2   LDA  $100
272 129E 8D2211             STA  DCNT+1
273 12A1 EE2111D0           IDB  DCNT
274 12A9 AD2211             LDA  DCNT+1
275 12AC 20C212             JSR  CNV2HX
276 12AF 8D7404             STA  RL1+36
277 12B2 8E7504             STX  RL1+37
278 12B5 AD2111             LDA  DCNT
279 12B8 20C212             JSR  CNV2HX
280 12BB 8D7604             STA  RL1+38
281 12BE 8E7704             STX  RL1+39
282 12C1 60                 RTS
283 12C2
284 12C2                                 CONVERT A TO 2 HEX DIGITS,
285 12C2                                 HIGH IN A, LOW IN X
286 12C2
287 12C2 AA          CNV2HX TAX
288 12C3 4A                 LSRA
289 12C4 4A                 LSRA
290 12C5 4A                 LSRA
291 12C6 4A                 LSRA
292 12C7 20D212             JSR  CVHXDG
293 12CA 48                 PHA
294 12CB 8A                 TXA
295 12CC 20D212             JSR  CVHXDG
296 12CF AA                 TAX
297 12D0 68                 PLA
298 12D1 60                 RTS
299 12D2 290F        CVHXDG AND  #$F
300 12D4 C90A               CMP  #10
301 12D6 9003               BLS  CVHX2
```

```
302 12D8 186907          ADD  #7
303 12DB 1869B0   CVHX2  ADD  #$B0
304 12DE 60              RTS
305 12DF          CMLINE EQU  $7D0
306 12DF 00       CLPTR  ***
307 12E0 4C3F10   RTILT  JMP  RCL3A
308 12E3                                KEYBOARD READ
309 12E3 A9A0     RKB    LDA  #' '
310 12E5 C018            CPY  #24
311 12E7 F0F7            BEQ  RTILT
312 12E9 99D007          STA  CMLINE,Y
313 12EC A940     RKB1   LDA  #$40
314 12EE 8DA614          STA  CFLASH
315 12F1 8DA714          STA  CFLASH+1
316 12F4 B9D007          LDA  CMLINE,Y
317 12F7 490E            EOR  #$E
318 12F9 99D007          STA  CMLINE,Y  FLASH CURSOR
319 12FC AD00C0   RKB2   LDA  KB
320 12FF 300C            BMI  RKB4
321 1301 EEA614   RKB3   INC  CFLASH
322 1304 D0FB            BNZ  RKB3
323 1306 EEA714          INC  CFLASH+1
324 1309 D0F1            BNZ  RKB2
325 130B F0DF            BZ   RKB1
326 130D 8D10C0   RKB4   STA  KBS       CLEAR STROBE
327 1310 C988            CMP  #$88
328 1312 F010            BEQ  RKB7      BACK UP CURSOR
329 1314 C9AE            CMP  #'.'      END OF LINE
330 1316 D009            BNE  RKB6
331 1318 38              SEC
332 1319 99D007   RKB5   STA  CMLINE,Y
333 131C C8              INY
334 131D 8CDF12          STY  CLPTR
335 1320 60              RTS
336 1321 18       RKB6   CLC
337 1322 90F5            BCC  RKB5
338 1324 A9A0     RKB7   LDA  #' '
339 1326 99D007          STA  CMLINE,Y
340 1329 88              DEY
341 132A 8CDF12          STY  CLPTR
342 132D C002            CPY  #2
343 132F B0B2            BGE  RKB
344 1331 A002            LDY  #2
345 1333 8CDF12          STY  CLPTR
346 1336 D0AB            BNZ  RKB
347 1338                                GET HEXADECIMAL
348 1338                                NUMBER FROM KEYBOARD
349 1338 A200     FPRM   LDX  #0
350 133A 8EA814          STX  RESULT
351 133D 8EA914          STX  RESULT+1
352 1340 B9D007   FPRM2  LDA  CMLINE,Y
353 1343 C9B0            CMP  #'0'      CHECK VALIDITY
354 1345 9033            BLS  PNG
355 1347 C9BA            CMP  #':'
356 1349 900B            BLS  DOK
357 134B C9C1            CMP  #'A'
358 134D 902B            BLS  PNG
359 134F C9C7            CMP  #'G'
360 1351 B027            BGE  PNG
361 1353 38E907          SUB  #7
```

```
362 1356 290F       DOK     AND  #$F
363 1358 0EA814             ASL  RESULT
364 135B 2EA914             ROL  RESULT+1
365 135E 0EA814             ASL  RESULT
366 1361 2EA914             ROL  RESULT+1
367 1364 0EA814             ASL  RESULT
368 1367 2EA914             ROL  RESULT+1
369 136A 0EA814             ASL  RESULT
370 136D 2EA914             ROL  RESULT+1
371 1370 0DA814             ORA  RESULT
372 1373 8DA814             STA  RESULT
373 1376 C8                 INY
374 1377 E8                 INX
375 1378 D0C6               BNZ  FPRM2
376 137A C9AE       PNG     CMP  #'.'       CHECK FOR TERMINATOR
377 137C F00B               BEQ  POK
378 137E C9A0               CMP  #' '
379 1380 F007               BEQ  POK
380 1382 C9AC               CMP  #','
381 1384 F003               BEQ  POK
382 1386 A900               LDA  #0
383 1388 60                 RTS
384 1389 8DAA14     POK     STA  LAST
385 138C 6A                 RORA
386 138D 6A                 RORA
387 138E C8                 INY
388 138F 8A                 TXA
389 1390 60                 RTS
390 1391 4C3F10     TILT    JMP  RCL3A
391 1394                                    INSPECT/MODIFY MEMORY
392 1394 A002       MEMORY  LDY  #2
393 1396 203813             JSR  FPRM
394 1399 F0F6               BZ   TILT
395 139B 90F4               BCC  TILT
396 139D ADA814             LDA  RESULT
397 13A0 8DD613             STA  MADR
398 13A3 8D0214             STA  RPADR      INITIALIZE ADDRESSES
399 13A6 ADA914             LDA  RESULT+1
400 13A9 8DD713             STA  MADR+1
401 13AC 8D0314             STA  RPADR+1
402 13AF A9A0       M0      LDA  #' '
403 13B1 A000               LDY  #0
404 13B3 99D207     M0A     STA  CMLINE+2,Y
405 13B6 C8                 INY
406 13B7 C015               CPY  #21
407 13B9 D0F8               BNE  M0A
408 13BB A002               LDY  #2
409 13BD ADD713             LDA  MADR+1
410 13C0 20C212             JSR  CNV2HX
411 13C3 8DD207             STA  CMLINE+2
412 13C6 8ED307             STX  CMLINE+3
413 13C9 ADD613             LDA  MADR
414 13CC 20C212             JSR  CNV2HX
415 13CF 8DD407             STA  CMLINE+4
416 13D2 8ED507             STX  CMLINE+5
417 13D5 ADD513     M1      LDA  M1
418 13D8            MADR    EQU  M1+1
419 13D8 48                 PHA
420 13D9 20C212             JSR  CNV2HX
421 13DC 8DD707             STA  CMLINE+7
```

```
422 13DF 8ED807              STX  CMLINE+8
423 13E2 68                  PLA
424 13E3 8DDB07              STA  CMLINE+11
425 13E6 A00D                LDY  #13
426 13E8 20E312      M1A     JSR  RKB
427 13EB C9AE                CMP  #'.'
428 13ED F008                BEQ  M1B
429 13EF C9A0                CMP  #' '
430 13F1 F004                BEQ  M1B
431 13F3 C9AC                CMP  #','
432 13F5 D0F1                BNE  M1A
433 13F7 A00D        M1B     LDY  #13
434 13F9 203813              JSR  FPRM
435 13FC F006                BZ   M2A
436 13FE ADA814              LDA  RESULT
437 1401 8D0114      M2      STA  M2
438 1404             RPADR   EQU  M2+1
439 1404 ADAA14      M2A     LDA  LAST
440 1407 C9AE                CMP  #'.'
441 1409 D003                BNE  M3
442 140B 4C1210              JMP  DBG2A
443 140E C9A0        M3      CMP  #' '
444 1410 D013                BNE  M4
445 1412 EED613D0            IDB  MADR       INCREMENT FOR NEXT
446 141A EE0214D0            IDB  RPADR      ACCESS
447 1422 4CAF13      M3A     JMP  M0
448 1425 C9AC        M4      CMP  #','
449 1427 F003                BEQ  M5
450 1429 4C3F10              JMP  RCL3A
451 142C ADD613      M5      LDA  MADR       DECREMENT FOR NEXT
452 142F 38E901              DEA              ACCESS
453 1432 8DD613              STA  MADR
454 1435 8D0214              STA  RPADR
455 1438 C9FF                CMP  #$FF
456 143A D0E6                BNE  M3A
457 143C CED713              DEC  MADR+1
458 143F CE0314              DEC  RPADR+1
459 1442 4CAF13              JMP  M0
460 1445                                     MODIFY PSEUDO REGISTER
461 1445 A90F8D90    REG     CDB  REGADR,RVCTR
462 144F ADD207              LDA  CMLINE+2
463 1452 A000                LDY  #0
464 1454 8CAB14              STY  STKFLG
465 1457 C9C1                CMP  #'A'
466 1459 F01B                BEQ  RGI
467 145B C8                  INY
468 145C C9D8                CMP  #'X'
469 145E F016                BEQ  RGI
470 1460 C8                  INY
471 1461 C9D9                CMP  #'Y'
472 1463 F011                BEQ  RGI
473 1465 C8                  INY
474 1466 C9D3                CMP  #'S'
475 1468 D005                BNE  RG0
476 146A EEAB14              INC  STKFLG
477 146D D007                BNZ  RGI
478 146F C9D0        RG0     CMP  #'P'
479 1471 F039                BEQ  RP
480 1473 4C3F10      RGE     JMP  RCL3A
481 1476 98          RGI     TYA
```

```
482 1477 186D9014          ADD  REGADR    COMPUTE PSEUDO REGISTER
483 147B 8D9014            STA  REGADR    ADDRESS
484 147E 9003              BCC  RG1
485 1480 EE9114            INC  REGADR+1
486 1483 A003       RG1    LDY  #3
487 1485 203813     RG2    JSR  FPRM
488 1488 90E9              BCC  RGE
489 148A F0E7              BZ   RGE
490 148C ADA814            LDA  RESULT
491 148F 8D8F14     RG3    STA  RG3       OVERLAID ADDRESS
492 1492            REGADR EQU  RG3+1
493 1492 CEAB14            DEC  STKFLG
494 1495 D003              BNZ  RG3A
495 1497 8D2612            STA  TUSTK
496 149A 20C511     RG3A   JSR  UDSD
497 149D 204C11     RG4    JSR  URD
498 14A0 205818            JSR  UPDW
499 14A3 4C2310            JMP  RCL
500 14A6            KB     EQU  $C000
501 14A6            KBS    EQU  $C010
502 14A6 0000       CFLASH $$$
503 14A8 0000       RESULT $$$
504 14AA 00         LAST   ***
505 14AB 00         STKFLG ***
506 14AC                                   SET PSW BITS
507 14AC A003       RP     LDY  #3
508 14AE B9D007     RP0    LDA  CMLINE,Y
509 14B1 C8                INY
510 14B2 C9AE              CMP  #'.'
511 14B4 F0E7              BEQ  RG4
512 14B6 A207              LDX  #7
513 14B8 DDEB14     RP1    CMP  PBTBL-1,X
514 14BB F006              BEQ  RP2
515 14BD CA                DEX
516 14BE 10F8              BPL  RP1
517 14C0 4C3F10            JMP  RCL3A
518 14C3 BDF214     RP2    LDA  BTBL-1,X
519 14C6 8DA814            STA  RESULT
520 14C9 B9D007            LDA  CMLINE,Y
521 14CC C8                INY
522 14CD C9B1              CMP  #'1'
523 14CF F012              BEQ  RP4
524 14D1 C9B0              CMP  #'0'
525 14D3 D09E              BNE  RGE
526 14D5 ADA814            LDA  RESULT
527 14D8 49FF              CMA
528 14DA 2D1411            AND  PSW
529 14DD 8D1411     RP3    STA  PSW
530 14E0 4CAE14            JMP  RP0
531 14E3 ADA814     RP4    LDA  RESULT
532 14E6 0D1411            ORA  PSW
533 14E9 4CDD14            JMP  RP3
534 14EC CED6C2C4   PBTBL  ASC  'NVBDIZC'
535 14F3 80401008   BTBL   DATA $80,$40,$10,8,4,2,1
536 14FA 00         BPFLAG ***
537 14FB                                   JUMP FACILITY
538 14FB
539 14FB 203813     JUMP   JSR  FPRM       GET JUMP ADDRESS
540 14FE F028              BZ   JTILT
541 1500 9026              BCC  JTILT
```

```
542 1502 ADA814             LDA  RESULT
543 1505 8D2615             STA  JADR
544 1508 ADA914             LDA  RESULT+1
545 150B 8D2715             STA  JADR+1
546 150E 20D116      J2     JSR  STSCRN    SET JUMP SCREEN
547 1511 AE1211             LDX  S         LOAD PSEUDO REGISTERS
548 1514 9A                 TXS
549 1515 AD1411             LDA  PSW
550 1518 48                 PHA
551 1519 AD0F11             LDA  A
552 151C AE1011             LDX  X
553 151F AC1111             LDY  Y
554 1522 28                 PLP            SET JUMP FLAGS
555 1523 6C2615      J3     JMP  *JADR
556 1526 0000        JADR   $$$            JUMP ADDRESS
557 1528 4C3F10      JTILT  JMP  RCL3A
558 152B
559 152B                                   BREAKPOINT FACILITY
560 152B
561 152B 203813      BKPT   JSR  FPRM
562 152E B058               BCS  BCNT
563 1530 ADAA14             LDA  LAST
564 1533 C9A0               CMP  #' '
565 1535 D0F1               BNE  JTILT
566 1537 ADA814             LDA  RESULT
567 153A 8D2615             STA  JADR
568 153D ADA914             LDA  RESULT+1
569 1540 8D2715             STA  JADR+1
570 1543 203813      BKPT1  JSR  FPRM
571 1546 90E0               BCC  JTILT
572 1548 ADA814      BKPT1A LDA  RESULT
573 154B 8D7A15             STA  BPA
574 154E 8D7E15             STA  BPA2
575 1551 ADA914             LDA  RESULT+1
576 1554 8D7B15             STA  BPA+1
577 1557 8D7F15             STA  BPA2+1
578 155A A000               LDY  #0        MOVE PROGRAM WORDS
579 155C 207915      BKPT2  JSR  FBW
580 155F 998115             STA  SAVE,Y
581 1562 C8                 INY
582 1563 C003               CPY  #3
583 1565 D0F5               BNE  BKPT2
584 1567 A000               LDY  #0        SET BREAKPOINT
585 1569 B98415      BKPT3  LDA  BPI,Y
586 156C 207D15             JSR  SBW
587 156F C8                 INY
588 1570 C003               CPY  #3
589 1572 D0F5               BNE  BKPT3
590 1574 EEFA14             INC  BPFLAG
591 1577 D095               BNZ  J2        AND JOIN JUMP CODE
592 1579 B97915      FBW    LDA  FBW,Y
593 157C             BPA    EQU  FBW+1
594 157C 60                 RTS
595 157D 997D15      SBW    STA  SBW,Y
596 1580             BPA2   EQU  SBW+1
597 1580 60                 RTS
598 1581             SAVE   RES  3
599 1584 4CAC15      BPI    JMP  BRTN
600 1587 00          PBKF   ***            PREVIOUS BKPT FLAG
601 1588 AD8715      BCNT   LDA  PBKF       WAS THERE A PREVIOUS BKPT?
```

```
602 158B D010              BNZ  BCNT2
603 158D B92F16    BCNT1   LDA  NPB,Y
604 1590 99D007            STA  CMLINE,Y  SET NO PREV. BKPT MESSAGE
605 1593 C8                INY
606 1594 C9D4              CMP  #'T'
607 1596 D0F5              BNE  BCNT1
608 1598 A000              LDY  #0
609 159A 4C4910            JMP  RCL4A
610 159D AD7A15    BCNT2   LDA  BPA       MOVE PREVIOUS BKPT ADDR
611 15A0 8D2615            STA  JADR      TO JUMP ADDRESS
612 15A3 AD7B15            LDA  BPA+1
613 15A6 8D2715            STA  JADR+1
614 15A9 4C4815            JMP  BKPT1A
615 15AC
616 15AC                                  BREAKPOINT RETURN FACILITY
617 15AC
618 15AC 8D2916    BRTN    STA  TA        SAVE REG'S
619 15AF 8E2A16            STX  TX
620 15B2 8C2B16            STY  TY
621 15B5 08                PHP
622 15B6 68                PLA
623 15B7 8D2E16            STA  TPSW
624 15BA BA                TSX
625 15BB 8E2C16            STX  TS
626 15BE 78                DI             KILL INTERRUPTS
627 15BF A220              LDX  #DBS
628 15C1 9A                TXS            LOAD DEBUG STACK
629 15C2 8E8715            STX  PBKF
630 15C5 20FA10            JSR  ADVR      ADVANCE REG FILE
631 15C8 A000              LDY  #0        MOVE IN NEW PSEUDO REG'S
632 15CA B92916    BRTN2   LDA  TA,Y
633 15CD 990F11            STA  A,Y
634 15D0 C8                INY
635 15D1 C006              CPY  #6
636 15D3 D0F5              BNE  BRTN2
637 15D5 AD1211            LDA  S         COMPUTE STACK DIFF.
638 15D8 38ED1811          SUB  S+6
639 15DC 8D1311            STA  D
640 15DF 202812            JSR  UBPD      AND BREAKPOINT DISPLAY
641 15E2 A000              LDY  #0
642 15E4 B93C16    BRTN3   LDA  BPRM,Y    DISPLAY BKPT MESSAGE
643 15E7 99D007            STA  CMLINE,Y
644 15EA C8                INY
645 15EB C9C4              CMP  #'D'
646 15ED D0F5              BNE  BRTN3
647 15EF A900      BRTN4   CLA            CLEAR BKPT FLAG
648 15F1 8DFA14            STA  BPFLAG
649 15F4 8DA614            STA  CFLASH
650 15F7 8DA714            STA  CFLASH+1  SET DELAY
651 15FA 204C11            JSR  URD       UPDATE ALL DISPLAYS
652 15FD 20C511            JSR  UDSD
653 1600 208612            JSR  URTS
654 1603 205818            JSR  UPDW
655 1606 A000              LDY  #0        RESTORE BKPT INST'S
656 1608 B98115    BRTN5   LDA  SAVE,Y
657 160B 207D15            JSR  SBW
658 160E C8                INY
659 160F C003              CPY  #3
660 1611 D0F5              BNE  BRTN5
661 1613 EEA614    BRTN6   INC  CFLASH
```

```
662 1616 D0FB              BNZ  BRTN6
663 1618 EEA714            INC  CFLASH+1
664 161B D0F6              BNZ  BRTN6
665 161D AD51C0            LDA  $C051      TURN ON NORMAL
666 1620 AD52C0            LDA  $C052      DEBUG SCREEN
667 1623 AD54C0            LDA  $C054
668 1626 4C2310            JMP  RCL
669 1629 00          TA    ***
670 162A 00          TX    ***
671 162B 00          TY    ***
672 162C 00          TS    ***
673 162D 00          TD    ***
674 162E 00          TPSW  ***
675 162F CECFA0D0    NPB   ASC  'NO PREV. BKPT'
676 163C C2CBD0D4    BPRM  ASC  'BKPT REACHED'
677 1648             DBS   EQU  $20
678 1648 A000        BPFAIL LDY  #0
679 164A B95816      BPF2  LDA  BPFM,Y
680 164D 99D007            STA  CMLINE,Y
681 1650 C8                INY
682 1651 C9C4              CMP  #'D'
683 1653 D0F5              BNE  BPF2
684 1655 4CEF15            JMP  BRTN4
685 1658 C2CBD0D4    BPFM  ASC  'BKPT FAILED'
686 1663
687 1663                                   SET JUMP SCREEN
688 1663
689 1663 AED207      VIEW  LDX  CMLINE+2
690 1666 A900              LDA  #0
691 1668 E0D4              CPX  #'T'
692 166A D004              BNE  V2
693 166C A2B3              LDX  #'3'
694 166E D025              BNZ  V7
695 1670 E0CC        V2    CPX  #'L'
696 1672 D006              BNE  V4
697 1674 A908              LDA  #8
698 1676 A2B5        V3    LDX  #'5'
699 1678 D01B              BNZ  V7
700 167A E0C8        V4    CPX  #'H'
701 167C D004              BNE  V5
702 167E A918              LDA  #24
703 1680 D0F4              BNZ  V3
704 1682 E0CE        V5    CPX  #'N'
705 1684 D00C              BNE  VE
706 1686 AD51C0            LDA  $C051      SWITCH TO
707 1689 AD52C0            LDA  $C052      NORMAL DEBUG
708 168C AD54C0            LDA  $C054      SCREEN
709 168F 4C2310            JMP  RCL
710 1692 4C3F10      VE    JMP  RCL3A
711 1695 8E2111      V7    STX  DCNT
712 1698 8D2211            STA  DCNT+1
713 169B ADD307            LDA  CMLINE+3
714 169E C9B1              CMP  #'1'
715 16A0 90F0              BLS  VE
716 16A2 CD2111            CMP  DCNT
717 16A5 B0EB              BGE  VE
718 16A7 290F              AND  #$F
719 16A9 38E901            DEA
720 16AC 0A                ASLA
721 16AD 0A                ASLA
```

```
722 16AE 186D2211          ADD  DCNT+1
723 16B2 1869E0     V8     ADD  #<SCRADR
724 16B5 8DD416            STA  JSCR
725 16B8 A916              LDA  #>SCRADR
726 16BA 6900              ADC  #0
727 16BC 8DD516            STA  JSCR+1
728 16BF 20D116            JSR  STSCRN
729 16C2 ADD207            LDA  CMLINE+2
730 16C5 8DF507            STA  FLN+37
731 16C8 ADD307            LDA  CMLINE+3
732 16CB 8DF607            STA  FLN+38
733 16CE 4C2310            JMP  RCL
734 16D1 A203       STSCRN LDX  #3
735 16D3 BDE016     STSC2  LDA  SCRADR,X  OVERLAID ADDRESS
736 16D6            JSCR   EQU  STSC2+1
737 16D6 8DDA16            STA  STSC3+1
738 16D9 AD00C0     STSC3  LDA  $C000
739 16DC CA                DEX
740 16DD 10F4              BPL  STSC2
741 16DF 60                RTS
742 16E0 00515254   SCRADR DATA 0,$51,$52,$54 TEXT 1
743 16E4 00515255          DATA 0,$51,$52,$55 TEXT 2
744 16E8 50525456          DATA $50,$52,$54,$56 LORES 1
745 16EC 50525556          DATA $50,$52,$55,$56 LORES 2
746 16F0 50535456          DATA $50,$53,$54,$56 MIXED LORES 1
747 16F4 50535556          DATA $50,$53,$55,$56 MIXED LORES 2
748 16F8 50525457          DATA $50,$52,$54,$57 HIRES 1
749 16FC 50525557          DATA $50,$52,$55,$57 HIRES 2
750 1700 50535457          DATA $50,$53,$54,$57
751 1704 50535557          DATA $50,$53,$55,$57
752 1708
753 1708                                 WINDOW ASSIGNMENT
754 1708
755 1708 ADD207     WINDOW LDA  CMLINE+2  GET WINDOW #
756 170B C9B1              CMP  #'1'      1 THROUGH F
757 170D 902B              BLS  WTILT
758 170F C9BA              CMP  #':'
759 1711 900B              BLS  WN2
760 1713 C9C1              CMP  #'A'
761 1715 9023              BLS  WTILT
762 1717 C9C7              CMP  #'G'
763 1719 B01F              BGE  WTILT
764 171B 38E907            SUB  #7
765 171E 290F       WN2    AND  #$F
766 1720 38E901            DEA
767 1723 8D5718            STA  WNO       SAVE WINDOW #, 0-14
768 1726 ADD307            LDA  CMLINE+3
769 1729 C9AC              CMP  #','      CHECK FOR COMMA
770 172B D00D              BNE  WTILT
771 172D ADD407            LDA  CMLINE+4
772 1730 A205              LDX  #5
773 1732 DD5118     WN3    CMP  WTBL,X
774 1735 F006              BEQ  WN4
775 1737 CA                DEX
776 1738 10F8              BPL  WN3
777 173A 4C3F10     WTILT  JMP  RCL3A     ILLEGAL TYPE
778 173D 8E4818     WN4    STX  WTYPE
779 1740 E005              CPX  #5        CLEAR?
780 1742 F06A              BEQ  CLEAR     YES, DO IT
781 1744 A9A0              LDA  #' '
```

```
782 1746 A200              LDX  #0
783 1748 9D4B18    WN5     STA  WNAME,X   CLEAR NAME SPACE
784 174B E8                INX
785 174C E006              CPX  #6
786 174E D0F8              BNE  WN5
787 1750 A006              LDY  #6
788 1752 203813            JSR  FPRM      GET WINDOW ADDRESS
789 1755 F0E3              BZ   WTILT
790 1757 B019              BCS  WN7
791 1759 A200              LDX  #0
792 175B B9D007    WN6     LDA  CMLINE,Y  TRANSMIT NAME
793 175E C9AE              CMP  #'.'
794 1760 F010              BEQ  WN7
795 1762 9D4B18            STA  WNAME,X
796 1765 C8                INY
797 1766 E8                INX
798 1767 E006              CPX  #6
799 1769 D0F0              BNE  WN6
800 176B B9D007            LDA  CMLINE,Y  CHECK FOR TERMINAL
801 176E C9AE              CMP  #'.'      COMMA
802 1770 D0C8              BNE  WTILT
803 1772 AD5718    WN7     LDA  WNO
804 1775 0A                ASLA           WINDOW NUMBER
805 1776 0A                ASLA           TIMES 9
806 1777 0A                ASLA
807 1778 186D5718          ADD  WNO
808 177C 1869C1            ADD  #<WDTBL
809 177F 8DB617            STA  WPTR
810 1782 A917              LDA  #>WDTBL
811 1784 6900              ADC  #0
812 1786 8DB717            STA  WPTR+1
813 1789 AD4818            LDA  WTYPE     GET TYPE
814 178C 20B517            JSR  EWTBL     ENTER WINDOW TYPE
815 178F ADA814            LDA  RESULT    AND ADDRESS
816 1792 20B517            JSR  EWTBL
817 1795 ADA914            LDA  RESULT+1
818 1798 20B517            JSR  EWTBL
819 179B A200              LDX  #0        TRANSMIT NAME
820 179D BD4B18    WN8     LDA  WNAME,X
821 17A0 20B517            JSR  EWTBL
822 17A3 E8                INX
823 17A4 E006              CPX  #6
824 17A6 D0F5              BNE  WN8
825 17A8 205818            JSR  UPDW
826 17AB 4C2310            JMP  RCL
827 17AE A9FF      CLEAR   LDA  #$FF
828 17B0 8D4818            STA  WTYPE
829 17B3 D0BD              BNZ  WN7
830 17B5 8DB517    EWTBL   STA  EWTBL
831 17B8           WPTR    EQU  EWTBL+1
832 17B8 EEB617D0          IDB  WPTR
833 17C0 60                RTS
834 17C1 FF        WDTBL   DATA 255
835 17C2                   RES  8
836 17CA FF        W2      DATA 255
837 17CB                   RES  8
838 17D3 FF        W3      DATA 255
839 17D4                   RES  8
840 17DC FF        W4      DATA 255
841 17DD                   RES  8
```

```
842 17E5 FF          W5      DATA 255
843 17E6                     RES  8
844 17EE FF          W6      DATA 255
845 17EF                     RES  8
846 17F7 FF          W7      DATA 255
847 17F8                     RES  8
848 1800 FF          W8      DATA 255
849 1801                     RES  8
850 1809 FF          W9      DATA 255
851 180A                     RES  8
852 1812 FF          WA      DATA 255
853 1813                     RES  8
854 181B FF          WB      DATA 255
855 181C                     RES  8
856 1824 FF          WC      DATA 255
857 1825                     RES  8
858 182D FF          WD      DATA 255
859 182E                     RES  8
860 1836 FF          WE      DATA 255
861 1837                     RES  8
862 183F FF          WF      DATA 255
863 1840                     RES  8
864 1848 00          WTYPE   ***
865 1849 0000        WADR    $$$
866 184B             WNAME   RES  6
867 1851 C1D3C4D0    WTBL    ASC  'ASDPHC'
868 1857 00          WNO     ***
869 1858
870 1858                                      UPDATE WINDOW DISPLAY
871 1858
872 1858 A9C18D6A    UPDW    CDB  WUPTR,WDTBL CREATE TABLE POINTER
873 1862 A900                CLA
874 1864 8D5718              STA  WNO
875 1867 A000        UPDW2   LDY  #0
876 1869 B96918      UPDW2A  LDA  UPDW2A,Y
877 186C             WUPTR   EQU  UPDW2A+1
878 186C 994818              STA  WTYPE,Y   MOVE WINDOW FILE
879 186F C8                  INY
880 1870 C009                CPY  #9
881 1872 D0F5                BNE  UPDW2A
882 1874 AD6A18              LDA  WUPTR     UPDATE POINTER
883 1877 186909              ADD  #9
884 187A 8D6A18              STA  WUPTR
885 187D AD6B18              LDA  WUPTR+1
886 1880 6900                ADC  #0
887 1882 8D6B18              STA  WUPTR+1
888 1885 AD5718      UPDW2B  LDA  WNO       GET WINDOW NUMBER
889 1888 AA                  TAX            COMPUTE SCREEN ADDRESS
890 1889 2907                AND  #7
891 188B 0908                ORA  #8
892 188D 8D2719              STA  WFADR+1
893 1890 8A                  TXA
894 1891 2918                AND  #$18
895 1893 8D2619              STA  WFADR
896 1896 0A                  ASLA
897 1897 0A                  ASLA
898 1898 0D2619              ORA  WFADR
899 189B 0A                  ASLA
900 189C 8D2619              STA  WFADR
901 189F 6E2719              ROR  WFADR+1
```

```
902 18A2 6E2619             ROR  WFADR
903 18A5 AD2619             LDA  WFADR
904 18A8 8DB618             STA  WUPTR2
905 18AB AD2719             LDA  WFADR+1
906 18AE 8DB718             STA  WUPTR2+1
907 18B1 A027               LDY  #39
908 18B3 A9A0               LDA  #' '
909 18B5 99B518     UPDW3   STA  UPDW3,Y   CLEAR LINE
910 18B8            WUPTR2  EQU  UPDW3+1
911 18B8 88                 DEY
912 18B9 10FA               BPL  UPDW3
913 18BB AD4818             LDA  WTYPE
914 18BE 1003               BPL  UPDW3A
915 18C0 4C2E1A             JMP  EOW
916 18C3 AD4A18     UPDW3A  LDA  WADR+1
917 18C6 20C212             JSR  CNV2HX
918 18C9 202519             JSR  STWND
919 18CC 8A                 TXA
920 18CD 202519             JSR  STWND
921 18D0 AD4918             LDA  WADR
922 18D3 20C212             JSR  CNV2HX
923 18D6 202519             JSR  STWND
924 18D9 8A                 TXA
925 18DA 202519             JSR  STWND
926 18DD A9A0               LDA  #' '
927 18DF 202519             JSR  STWND
928 18E2 AD4918             LDA  WADR
929 18E5 8D3219             STA  ACSPTR
930 18E8 AD4A18             LDA  WADR+1
931 18EB 8D3319             STA  ACSPTR+1
932 18EE A200               LDX  #0
933 18F0 BD4B18     UPDW4   LDA  WNAME,X   MOVE NAME TO WINDOW
934 18F3 202519             JSR  STWND
935 18F6 E8                 INX
936 18F7 E006               CPX  #6
937 18F9 D0F5               BNE  UPDW4
938 18FB A9A0               LDA  #' '
939 18FD 202519             JSR  STWND
940 1900 AD4818             LDA  WTYPE     JUMP TO APPROPRIATE
941 1903 F038               BZ   ASCW      PROCESSOR
942 1905 38E901             DEA
943 1908 F043               BZ   SNGLW
944 190A 38E901             DEA
945 190D D003               BNZ  TPTR
946 190F 4C9819             JMP  DBLW
947 1912 38E901     TPTR    DEA
948 1915 D003               BNZ  THEX
949 1917 4C3F1A             JMP  PTRW
950 191A 38E901     THEX    DEA
951 191D D003               BNZ  JMP
952 191F 4C121A             JMP  HEXW
953 1922 4C2219     JMP     JMP  JMP
954 1925 8D2519     STWND   STA  STWND
955 1928            WFADR   EQU  STWND+1
956 1928 EE2619D0           IDB  WFADR
957 1930 60                 RTS
958 1931 AD3119     FWD     LDA  FWD
959 1934            ACSPTR  EQU  FWD+1
960 1934 EE3219D0           IDB  ACSPTR
961 193C 60                 RTS
```

```
962 193D
963 193D                                        ASCII WINDOW
964 193D
965 193D A000       ASCW    LDY  #0
966 193F 203119     ASCW2   JSR  FWD
967 1942 202519             JSR  STWND      MOVE ASCII DATA
968 1945 C8                 INY
969 1946 C01C               CPY  #28
970 1948 D0F5               BNE  ASCW2
971 194A 4C2E1A     ASCW3   JMP  EOW
972 194D
973 194D                                        SINGLE PRECISION DECIMAL
974 194D
975 194D 203119     SNGLW   JSR  FWD        GET NUMBER
976 1950 8D2111             STA  DCNT       SAVE IT
977 1953 A2B0       SNW0    LDX  #'0'
978 1955 38E964     SNW1    SUB  #100       CONVERT TO ABSOLUTE
979 1958 9003               BCC  SWN2       DECIMAL
980 195A E8                 INX
981 195B D0F8               BNZ  SNW1
982 195D 6964       SWN2    ADC  #100
983 195F 48                 PHA
984 1960 8A                 TXA
985 1961 202519             JSR  STWND      SAVE DIGIT
986 1964 68                 PLA
987 1965 A2B0               LDX  #'0'
988 1967 38E90A     SWN3    SUB  #10
989 196A 9003               BCC  SWN4
990 196C E8                 INX
991 196D D0F8               BNZ  SWN3
992 196F 69BA       SWN4    ADC  #$BA
993 1971 48                 PHA
994 1972 8A                 TXA
995 1973 202519             JSR  STWND
996 1976 68                 PLA
997 1977 202519             JSR  STWND
998 197A A9A0               LDA  #' '
999 197C 202519             JSR  STWND      SPACE BETWEEN FIELDS
000 197F AD2111             LDA  DCNT
001 1982 10C6       SWN4A   BPL  ASCW3      FINISHED IF POSITIVE
002 1984 A9AD               LDA  #'-'
003 1986 202519             JSR  STWND      SET MINUS SIGN
004 1989 AD2111             LDA  DCNT
005 198C 1849FF69           TCA             FORCE ARGUMENT POSITIVE
006 1991 A000               LDY  #0
007 1993 8C2111             STY  DCNT
008 1996 10BB       SWN5    BPL  SNW0
009 1998
010 1998                                        DOUBLE PRECISION DECIMAL
011 1998
012 1998 203119     DBLW    JSR  FWD        GET DOUBLE NUMBER
013 199B 8D2111             STA  DCNT
014 199E 8D2916             STA  TA
015 19A1 203119             JSR  FWD
016 19A4 8D2211             STA  DCNT+1
017 19A7 8D2A16             STA  TA+1
018 19AA A905       DBLW0   LDA  #5
019 19AC 8D101A             STA  WCNT
020 19AF A000               LDY  #0         POWER TABLE POINTER
021 19B1 A9B0       DBLW1   LDA  #'0'
```

```
022 19B3 8D111A            STA  DIGIT
023 19B6 AD2111     DBLW2  LDA  DCNT
024 19B9 38F9061A          SUB  PT,Y
025 19BD AA                TAX
026 19BE AD2211            LDA  DCNT+1
027 19C1 F9071A            SBC  PT+1,Y
028 19C4 900B              BCC  DBLW3
029 19C6 8D2211            STA  DCNT+1
030 19C9 8E2111            STX  DCNT
031 19CC EE111A            INC  DIGIT
032 19CF D0E5              BNZ  DBLW2
033 19D1 AD111A     DBLW3  LDA  DIGIT
034 19D4 202519            JSR  STWND
035 19D7 C8                INY             NEXT POWER OF TEN
036 19D8 C8                INY
037 19D9 CE101A            DEC  WCNT
038 19DC D0D3              BNZ  DBLW1
039 19DE AD2A16            LDA  TA+1
040 19E1 109F              BPL  SWN4A
041 19E3 A9AD              LDA  #'-'
042 19E5 202519            JSR  STWND
043 19E8 AD2A16            LDA  TA+1
044 19EB 49FF              CMA
045 19ED 8D2211            STA  DCNT+1
046 19F0 8D2A16            STA  TA+1
047 19F3 AD2916            LDA  TA
048 19F6 1849FF69          TCA
049 19FB 8D2111            STA  DCNT
050 19FE 90AA              BCC  DBLW0
051 1A00 EE2211            INC  DCNT+1
052 1A03 4CAA19            JMP  DBLW0
053 1A06 1027E803   PT     DBL  10000,1000,100,10,1
054 1A10 00         WCNT   ***
055 1A11 00         DIGIT  ***
056 1A12
057 1A12                                   HEX WINDOW
058 1A12
059 1A12 A908       HEXW   LDA  #8
060 1A14 8D2111            STA  DCNT
061 1A17 203119     HEXW1  JSR  FWD
062 1A1A 20C212            JSR  CNV2HX
063 1A1D 202519            JSR  STWND
064 1A20 8A                TXA
065 1A21 202519            JSR  STWND
066 1A24 A9A0              LDA  #' '
067 1A26 202519            JSR  STWND
068 1A29 CE2111            DEC  DCNT
069 1A2C D0E9              BNZ  HEXW1
070 1A2E AD5718     EOW    LDA  WNO
071 1A31 186901            INA
072 1A34 8D5718            STA  WNO
073 1A37 C90F              CMP  #15
074 1A39 D001              BNE  EOL2
075 1A3B 60                RTS
076 1A3C 4C6718     EOL2   JMP  UPDW2
077 1A3F
078 1A3F                                   POINTER WINDOW
079 1A3F
080 1A3F 203119     PTRW   JSR  FWD
081 1A42 8D791A            STA  PTRADR     INITIALIZE ADDRESSES
```

```
082 1A45 8D9D1A             STA  PTRAD2
083 1A48 8DC11A             STA  PTRAD3
084 1A4B 203119             JSR  FWD
085 1A4E 8D7A1A             STA  PTRADR+1
086 1A51 8D9E1A             STA  PTRAD2+1
087 1A54 8DC21A             STA  PTRAD3+1
088 1A57 20C212             JSR  CNV2HX     CONVERT PTR VALUE
089 1A5A 202519             JSR  STWND
090 1A5D 8A                 TXA
091 1A5E 202519             JSR  STWND
092 1A61 AD791A             LDA  PTRADR
093 1A64 20C212             JSR  CNV2HX
094 1A67 202519             JSR  STWND
095 1A6A 8A                 TXA
096 1A6B 202519             JSR  STWND
097 1A6E A9A0               LDA  #' '
098 1A70 202519             JSR  STWND
099 1A73 A9BE               LDA  #'>'
100 1A75 202519             JSR  STWND
101 1A78                                    ACCESS DIRECT THROUGH PTR
102 1A78 AD781A     PTRW1   LDA  PTRW1
103 1A7B            PTRADR  EQU  PTRW1+1
104 1A7B 20C212             JSR  CNV2HX     CONVERT DIRECT OBJECT
105 1A7E 202519             JSR  STWND
106 1A81 8A                 TXA
107 1A82 202519             JSR  STWND
108 1A85 A9A0               LDA  #' '
109 1A87 202519             JSR  STWND
110 1A8A A9AC               LDA  #','
111 1A8C 202519             JSR  STWND
112 1A8F A9D8               LDA  #'X'
113 1A91 202519             JSR  STWND
114 1A94 A9BE               LDA  #'>'
115 1A96 202519             JSR  STWND
116 1A99 AE1011             LDX  X
117 1A9C BD9C1A     PTRW2   LDA  PTRW2,X    INDEXED BY X
118 1A9F            PTRAD2  EQU  PTRW2+1
119 1A9F 20C212             JSR  CNV2HX
120 1AA2 202519             JSR  STWND
121 1AA5 8A                 TXA
122 1AA6 202519             JSR  STWND
123 1AA9 A9A0               LDA  #' '
124 1AAB 202519             JSR  STWND
125 1AAE A9AC               LDA  #','
126 1AB0 202519             JSR  STWND
127 1AB3 A9D9               LDA  #'Y'
128 1AB5 202519             JSR  STWND
129 1AB8 A9BE               LDA  #'>'
130 1ABA 202519             JSR  STWND
131 1ABD AC1111             LDY  Y
132 1AC0 B9C01A     PTRW3   LDA  PTRW3,Y    INDEXED BY Y
133 1AC3            PTRAD3  EQU  PTRW3+1
134 1AC3 20C212             JSR  CNV2HX
135 1AC6 202519             JSR  STWND
136 1AC9 8A                 TXA
137 1ACA 202519             JSR  STWND
138 1ACD A9A0               LDA  #' '
139 1ACF 202519             JSR  STWND
140 1AD2 4C2E1A     EPTRW   JMP  EOW
141 1AD5
```

```
142 1AD5                                        LOAD OBJECT FROM CASSETTE
143 1AD5
144 1AD5 A000       LOAD    LDY  #0
145 1AD7 B9201C     LOAD1   LDA  OTMSG,Y
146 1ADA 99D007             STA  CMLINE,Y
147 1ADD C8                 INY
148 1ADE C013               CPY  #19
149 1AE0 D0F5               BNE  LOAD1
150 1AE2 20E312             JSR  RKB
151 1AE5 C9A0               CMP  #' '
152 1AE7 F003               BEQ  LOAD3
153 1AE9 4C3F10             JMP  RCL3A
154 1AEC A9CC       LOAD3   LDA  #'L'
155 1AEE 8D7405             STA  SCREEN+$174
156 1AF1 A9CF               LDA  #'O'
157 1AF3 8D7505             STA  SCREEN+$175
158 1AF6 A9C1               LDA  #'A'
159 1AF8 8D7605             STA  SCREEN+$176
160 1AFB A9C4               LDA  #'D'
161 1AFD 8D7705             STA  SCREEN+$177
162 1B00 A9FF               LDA  #-1
163 1B02 8D4B1C             STA  ATRG
164 1B05 A900               CLA
165 1B07 8D091C             STA  NRCDS
166 1B0A A000       RH0     LDY  #0
167 1B0C 20ED1B     RH1     JSR  RTRANS
168 1B0F 70FB               BVS  RH1
169 1B11 E01B               CPX  #TTIME
170 1B13 90F5               BLS  RH0
171 1B15 C8                 INY
172 1B16 C050               CPY  #80
173 1B18 90F2               BLS  RH1
174 1B1A 20ED1B     RH2     JSR  RTRANS
175 1B1D E01B               CPX  #TTIME
176 1B1F B0F9               BGE  RH2
177 1B21 20ED1B             JSR  RTRANS
178 1B24 E01B               CPX  #TTIME
179 1B26 B0F2               BGE  RH2
180 1B28 A900       RR1     CLA
181 1B2A 8D451C             STA  CKSM       CLEAR CHECKSUM
182 1B2D 20C91B             JSR  RDBYTE     GET RECORD MARK
183 1B30 C9FF               CMP  #$FF       FOR REAL?
184 1B32 D06A               BNE  RMERR      NO, ERROR
185 1B34 20C91B             JSR  RDBYTE     GET RECORD ORIGIN
186 1B37 8D621B             STA  RORG
187 1B3A 20C91B             JSR  RDBYTE
188 1B3D 8D631B             STA  RORG+1
189 1B40 EE4B1C             INC  ATRG
190 1B43 D009               BNZ  RR1A
191 1B45 8D491C             STA  PORG+1     SAVE PROGRAM ORIGIN
192 1B48 AD621B             LDA  RORG
193 1B4B 8D481C             STA  PORG
194 1B4E 20C91B     RR1A    JSR  RDBYTE     GET RECORD TYPE
195 1B51 8D4A1C             STA  RTYPE
196 1B54 3034               BMI  EOF        RECORD TYPE $FF MEANS
197 1B56                                    END OF FILE
198 1B56 D056               BNZ  RSTYPE     "RES" GENERATED RECORD
199 1B58 20C91B             JSR  RDBYTE     NORMAL DATA RECORD
200 1B5B 8D071C             STA  RCDLNT     SAVE RECORD LENGTH
201 1B5E 20C91B     RR2     JSR  RDBYTE     GET DATA/INST WORD
```

```
202 1B61 8D611B      RR3     STA  RR3        MODIFIED ADDRESS
203 1B64             RORG    EQU  RR3+1
204 1B64 EE621BD0            IDB  RORG
205 1B6C 8DF405              STA  SCREEN+$1F4
206 1B6F CE071C              DEC  RCDLNT
207 1B72 D0EA                BNZ  RR2
208 1B74 AD451C      RR4     LDA  CKSM       SAVE COMPUTED CHECKSUM
209 1B77 8D0A1C              STA  BCHK
210 1B7A 20C91B              JSR  RDBYTE     GET TAPE CHECKSUM
211 1B7D 4D0A1C              EOR  BCHK       VERIFY CHECKSUMS
212 1B80 F003                BZ   RR5
213 1B82 4C0B1C              JMP  CSERR
214 1B85 EE091C      RR5     INC  NRCDS
215 1B88 D080                BNZ  RH0        GET NEXT RECORD
216 1B8A AD091C      EOF     LDA  NRCDS
217 1B8D 20C212              JSR  CNV2HX
218 1B90 8DF505              STA  SCREEN+$1F5
219 1B93 8EF605              STX  SCREEN+$1F6
220 1B96 A9A0                LDA  #' '
221 1B98 8DF405              STA  SCREEN+$1F4
222 1B9B 4C2310              JMP  RCL
223 1B9E A000        RMERR   LDY  #0
224 1BA0 B9331C      RME2    LDA  RMEMSG,Y
225 1BA3 99D007              STA  CMLINE,Y
226 1BA6 C8                  INY
227 1BA7 C012                CPY  #18
228 1BA9 D0F5                BNE  RME2
229 1BAB 4C4910              JMP  RCL4A
230 1BAE
231 1BAE                                     "RES" GENERATED RECORD
232 1BAE
233 1BAE 20C91B      RSTYPE  JSR  RDBYTE     GET FILL FLAG
234 1BB1 8D461C              STA  FLFLAG
235 1BB4 20C91B              JSR  RDBYTE     GET FILL CHARACTER
236 1BB7 8D471C              STA  FLCHR
237 1BBA 20C91B              JSR  RDBYTE
238 1BBD 8D071C              STA  RCDLNT
239 1BC0 20C91B              JSR  RDBYTE     GET RECORD LENGTH
240 1BC3 8D081C              STA  RCDLNT+1
241 1BC6 4C741B              JMP  RR4
242 1BC9 A008        RDBYTE  LDY  #8
243 1BCB A900                LDA  #0
244 1BCD 48                  PHA
245 1BCE 20EA1B      RDB2    JSR  RDBIT
246 1BD1 7014                BVS  RDB4
247 1BD3 68                  PLA
248 1BD4 E01B                CPX  #TTIME
249 1BD6 2A                  ROLA
250 1BD7 48                  PHA
251 1BD8 88                  DEY
252 1BD9 D0F3                BNZ  RDB2
253 1BDB 68                  PLA
254 1BDC 48                  PHA
255 1BDD 186D451C            ADD  CKSM
256 1BE1 8D451C              STA  CKSM
257 1BE4 68                  PLA
258 1BE5 18                  CLC
259 1BE6 60          RDB3    RTS
260 1BE7 68          RDB4    PLA
261 1BE8 38                  SEC
```

```
262 1BE9 60                  RTS
263 1BEA 20ED1B      RDBIT   JSR  RTRANS
264 1BED A200        RTRANS  LDX  #0
265 1BEF E8          RT2     INX
266 1BF0 F010                BZ   RT3
267 1BF2 AD60C0              LDA  TPIN
268 1BF5 4D1F1C              EOR  B4
269 1BF8 10F5                BPL  RT2
270 1BFA 4D1F1C              EOR  B4
271 1BFD 8D1F1C              STA  B4
272 1C00 B8                  CLV
273 1C01 60                  RTS
274 1C02 A980        RT3     LDA  #$80       SET OVERFLOW
275 1C04 6980                ADC  #$80
276 1C06 60                  RTS
277 1C07             TTIME   EQU  27
278 1C07 0000        RCDLNT  $$$
279 1C09 00          NRCDS   ***
280 1C0A             TPIN    EQU  $C060
281 1C0A 00          BCHK    ***
282 1C0B A000        CSERR   LDY  #0
283 1C0D B91B1C      CSE2    LDA  CKSMSG,Y
284 1C10 99F405              STA  SCREEN+$1F4,Y
285 1C13 C8                  INY
286 1C14 C004                CPY  #4
287 1C16 D0F5                BNE  CSE2
288 1C18 4C2310              JMP  RCL
289 1C1B C3CBD3CD    CKSMSG  ASC  'CKSM'
290 1C1F 00          B4      ***
291 1C20 CFC2CAC5    OTMSG   ASC  'OBJECT READY, SPACE'
292 1C33 C3C1CECE    RMEMSG  ASC  'CANNOT FIND RECORD'
293 1C45 00          CKSM    ***
294 1C46 00          FLFLAG  ***
295 1C47 00          FLCHR   ***
296 1C48 0000        PORG    $$$
297 1C4A 00          RTYPE   ***
298 1C4B 00          ATRG    ***
299 1C4C                             SUPPORT ROUTINES FOR
300 1C4C                             FILL, SEARCH AND DUMP
301 1C4C
302 1C4C                             GET LIMITS
303 1C4C
304 1C4C A002        GET2    LDY  #2
305 1C4E 203813              JSR  FPRM
306 1C51 F057                BZ   FTILT
307 1C53 ADAA14              LDA  LAST
308 1C56 C9A0                CMP  #' '
309 1C58 D050                BNE  FTILT
310 1C5A ADA814              LDA  RESULT
311 1C5D 8D3C1D              STA  FIRST
312 1C60 ADA914              LDA  RESULT+1
313 1C63 8D3D1D              STA  FIRST+1
314 1C66 203813              JSR  FPRM
315 1C69 F03F                BZ   FTILT
316 1C6B ADA814              LDA  RESULT
317 1C6E 8D3E1D              STA  SECOND
318 1C71 ADA914              LDA  RESULT+1
319 1C74 8D3F1D              STA  SECOND+1
320 1C77 EE3E1DD0            IDB  SECOND     FORM INCLUSIVE COUNT
321 1C7F AD3C1D              LDA  FIRST
```

```
322 1C82 38ED3E1D          SUB  SECOND
323 1C86 8D401D            STA  DIFF
324 1C89 AD3D1D            LDA  FIRST+1
325 1C8C ED3F1D            SBC  SECOND+1
326 1C8F 8D411D            STA  DIFF+1
327 1C92 B016              BCS  FTILT      DIFF. MUST BE NEGATIVE
328 1C94 ADAA14            LDA  LAST
329 1C97 60                RTS
330 1C98
331 1C98                                   GET LIMITS AND COMPARE
332 1C98                                   OR FILL VALUE
333 1C98
334 1C98 204C1C     GET3   JSR  GET2
335 1C9B C9A0              CMP  #' '
336 1C9D D00B              BNE  FTILT
337 1C9F 203813            JSR  FPRM
338 1CA2 ADAA14            LDA  LAST
339 1CA5 C9AE              CMP  #'.'       MUST END IN PERIOD
340 1CA7 D001              BNE  FTILT
341 1CA9 60                RTS
342 1CAA 4C3F10     FTILT  JMP  RCL3A
343 1CAD
344 1CAD                                   FILL FUNCTION
345 1CAD
346 1CAD 20981C     FILL   JSR  GET3
347 1CB0 AD3C1D            LDA  FIRST
348 1CB3 8DC01C            STA  FLADR
349 1CB6 AD3D1D            LDA  FIRST+1
350 1CB9 8DC11C            STA  FLADR+1
351 1CBC ADA814            LDA  RESULT
352 1CBF 8DBF1C     FL2    STA  FL2        OVERLAID ADDRESS
353 1CC2            FLADR  EQU  FL2+1
354 1CC2 EEC01CD0          IDB  FLADR
355 1CCA EE401DD0          IDB  DIFF
356 1CD2 D0EB              BNZ  FL2
357 1CD4 4C2310     FL3    JMP  RCL
358 1CD7
359 1CD7                                   SEARCH FUNCTION
360 1CD7
361 1CD7 20981C     SEARCH JSR  GET3
362 1CDA AD3C1D            LDA  FIRST
363 1CDD 8DF11C            STA  SRCADR
364 1CE0 AD3D1D            LDA  FIRST+1
365 1CE3 8DF21C            STA  SRCADR+1
366 1CE6 A016              LDY  #22
367 1CE8 A9A0              LDA  #' '
368 1CEA 99D007     SCC    STA  CMLINE,Y
369 1CED 88                DEY
370 1CEE 10FA              BPL  SCC
371 1CF0 ADF01C     SRC2   LDA  SRC2
372 1CF3            SRCADR EQU  SRC2+1
373 1CF3 CDA814            CMP  RESULT
374 1CF6 D030              BNE  SRC4
375 1CF8 ADF21C            LDA  SRCADR+1
376 1CFB 20C212            JSR  CNV2HX
377 1CFE 8DD107            STA  CMLINE+1
378 1D01 8ED207            STX  CMLINE+2
379 1D04 ADF11C            LDA  SRCADR
380 1D07 20C212            JSR  CNV2HX
381 1D0A 8DD307            STA  CMLINE+3
```

```
382 1D0D 8ED407              STX  CMLINE+4
383 1D10 ADA814              LDA  RESULT
384 1D13 20C212              JSR  CHV2HX
385 1D16 8DD607              STA  CMLINE+6
386 1D19 8ED707              STX  CMLINE+7
387 1D1C AD00C0      SRC3    LDA  KB
388 1D1F 10FB                BPL  SRC3
389 1D21 8D10C0              STA  KBS
390 1D24 C9AE                CMP  #'.'
391 1D26 F0AC                BEQ  FL3
392 1D28 EEF11CD0    SRC4    IDB  SRCADR
393 1D30 EE401DD0            IDB  DIFF
394 1D38 D0B6                BNZ  SRC2
395 1D3A F098                BZ   FL3
396 1D3C 0000        FIRST   $$$
397 1D3E 0000        SECOND  $$$
398 1D40 0000        DIFF    $$$
399 1D42
400 1D42                                        DUMP TO PRINTER
401 1D42
402 1D42 204C1C      DUMP    JSR  GET2
403 1D45 C9AE                CMP  #'.'
404 1D47 F003                BEQ  DMP2
405 1D49 4C3F10              JMP  RCL3A
406 1D4C AD3C1D      DMP2    LDA  FIRST
407 1D4F 8D711D              STA  DMPADR
408 1D52 AD3D1D              LDA  FIRST+1
409 1D55 8D721D              STA  DMPADR+1
410 1D58 AD721D      DUMP2   LDA  DMPADR+1
411 1D5B A000                LDY  #0
412 1D5D 20991D              JSR  CVP
413 1D60 88                  DEY
414 1D61 AD711D              LDA  DMPADR
415 1D64 20991D              JSR  CVP
416 1D67 99D41D              STA  DMPLN,Y
417 1D6A C8                  INY
418 1D6B A918                LDA  #24
419 1D6D 8D2111              STA  DCNT
420 1D70 AD701D      DUMP3   LDA  DUMP3      OVERLAID ADDRESS
421 1D73             DMPADR  EQU  DUMP3+1
422 1D73 20991D              JSR  CVP
423 1D76 EE711DD0            IDB  DMPADR
424 1D7E EE401DD0            IDB  DIFF
425 1D86 F00B                BZ   DLL
426 1D88 CE2111              DEC  DCNT
427 1D8B D0E3                BNZ  DUMP3
428 1D8D 20AC1D              JSR  PLINE
429 1D90 4C581D              JMP  DUMP2
430 1D93 20AC1D      DLL     JSR  PLINE
431 1D96 4C2310              JMP  RCL
432 1D99 20C212      CVP     JSR  CHV2HX
433 1D9C 99D41D              STA  DMPLN,Y
434 1D9F C8                  INY
435 1DA0 8A                  TXA
436 1DA1 99D41D              STA  DMPLN,Y
437 1DA4 C8                  INY
438 1DA5 A9A0                LDA  #' '
439 1DA7 99D41D              STA  DMPLN,Y
440 1DAA C8                  INY
441 1DAB 60                  RTS
```

```
442 1DAC A98D        PLINE   LDA  #$8D
443 1DAE 99D41D              STA  DMPLN,Y
444 1DB1 A000                LDY  #0
445 1DB3 A200                LDX  #0
446 1DB5 B9D41D      PL2     LDA  DMPLN,Y
447 1DB8 48                  PHA
448 1DB9 9DB0C0              STA  PRTR,X
449 1DBC BDC1C3      PL3     LDA  PRFLAG,X
450 1DBF 30FB                BMI  PL3
451 1DC1 68                  PLA
452 1DC2 C98D                CMP  #$8D
453 1DC4 F003                BEQ  PL4
454 1DC6 C8                  INY
455 1DC7 D0EC                BNZ  PL2
456 1DC9 A98A        PL4     LDA  #$8A
457 1DCB 9DB0C0              STA  PRTR,X
458 1DCE BDC1C3      PL5     LDA  PRFLAG,X
459 1DD1 30FB                BMI  PL5
460 1DD3 60                  RTS
461 1DD4             PRTR    EQU  $C0B0
462 1DD4             PRFLAG  EQU  $C3C1
463 1DD4             DMPLN   RES  80
464 1E24 8D                  DATA $8D
465 1E25                     END
```

```
*******************************
*  S Y M B O L   T A B L E  *
*******************************
```

A	110F	ACSPTR	1932	ADVR	10FA	ADVR2	10FE	ASCW	193D	ASCW2	193F
ASCW3	194A	ATRG	1C4B	B4	1C1F	BANNER	1123	BCHK	1C0A	BCNT	1588
BCNT1	158D	BCNT2	159D	BKPT	152B	BKPT1	1543	BKPT1A	1548	BKPT2	155C
BKPT3	1569	BPA	157A	BPA2	157E	BPF2	164A	BPFAIL	1648	BPFLAG	14FA
BPFM	1658	BPI	1584	BPRM	163C	BRTN	15AC	BRTN2	15CA	BRTN3	15E4
BRTN4	15EF	BRTN5	1608	BRTN6	1613	BTBL	14F3	CFLASH	14A6	CKSM	1C45
CKSMSG	1C1B	CLADR	10CB	CLEAR	17AE	CLPTR	12DF	CLS2	10CA	CLSCR	10B4
CMLINE	07D0	CNV2HX	12C2	CSE2	1C0D	CSERR	1C0B	CTBL	1087	CVHX2	12DB
CVHXDG	12D2	CVP	1D99	D	1113	DBG	1000	DBG2	1006	DBG2A	1012
DBG3	101B	DBLW	1998	DBLW0	19AA	DBLW1	19B1	DBLW2	19B6	DBLW3	19D1
DBS	0020	DCNT	1121	DIFF	1D40	DIGIT	1A11	DLL	1D93	DMP2	1D4C
DMPADR	1D71	DMPLN	1DD4	DOK	1356	DSPADR	11BA	DUMP	1D42	DUMP2	1D58
DUMP3	1D70	EOF	1B8A	EOL2	1A3C	EOW	1A2E	EPTRW	1AD2	EWTBL	17B5
FBW	1579	FCH	11AD	FILL	1CAD	FIRST	1D3C	FL2	1CBF	FL3	1CD4
FLADR	1CC0	FLCHR	1C47	FLFLAG	1C46	FLN	07D0	FPRM	1338	FPRM2	1340
FST1	1295	FST2	129B	FTILT	1CAA	FWD	1931	GET2	1C4C	GET3	1C98
HEXW	1A12	HEXW1	1A17	J2	150E	J3	1523	JADR	1526	JMP	1922
JSCR	16D4	JTBL	1096	JTILT	1528	JUMP	14FB	KB	C000	KBS	C010
LAST	14AA	LOAD	1AD5	LOAD1	1AD7	LOAD3	1AEC	M0	13AF	M0A	13B3
M1	13D5	M1A	13E8	M1B	13F7	M2	1401	M2A	1404	M3	140E
M3A	1422	M4	1425	M5	142C	MADR	13D6	MCNT	1224	MEMORY	1394
NPB	162F	NRCDS	1C09	NTL	0750	OTMSG	1C20	PBKF	1587	PBTBL	14EC
PL2	1DB5	PL3	1DBC	PL4	1DC9	PL5	1DCE	PLINE	1DAC	PNG	137A
POK	1389	PORG	1C48	PRFLAG	C3C1	PRTR	C0B0	PSW	1114	PT	1A06
PTRAD2	1A9D	PTRAD3	1AC1	PTRADR	1A79	PTRW	1A3F	PTRW1	1A78	PTRW2	1A9C
PTRW3	1AC0	RADR	11AE	RCDLNT	1C07	RCL	1023	RCL2	1027	RCL3	1036
RCL3A	103F	RCL4	1043	RCL4A	1049	RCL5	104F	RCL6	105F	RCL7	1064
RCL8	106E	RCNT	1225	RDB2	1BCE	RDB3	1BE6	RDB4	1BE7	RDBIT	1BEA
RDBYTE	1BC9	REG	1445	REGADR	1490	RESULT	14A8	RG0	146F	RG1	1483
RG2	1485	RG3	148F	RG3A	149A	RG4	149D	RGE	1473	RGI	1476
RH0	1B0A	RH1	1B0C	RH2	1B1A	RKB	12E3	RKB1	12EC	RKB2	12FC

RKB3	1301	RKB4	130D	RKB5	1319	RKB6	1321	RKB7	1324	RL1	0450
RME2	1BA0	RMEMSG	1C33	RMERR	1B9E	RORG	1B62	RP	14AC	RP0	14AE
RP1	14B8	RP2	14C3	RP3	14DD	RP4	14E3	RPADR	1402	RR1	1B28
RR1A	1B4E	RR2	1B5E	RR3	1B61	RR4	1B74	RR5	1B85	RSTYPE	1BAE
RT2	1BEF	RT3	1C02	RTILT	12E0	RTRANS	1BED	RTYPE	1C4A	RVCTR	110F
S	1112	SAVE	1581	SBLINE	07A8	SBW	157D	SCC	1CEA	SCRADR	16E0
SCREEN	0400	SEARCH	1CD7	SECOND	1D3E	SFMT	10E1	SNGLW	194D	SNW0	1953
SNW1	1955	SRC2	1CF0	SRC3	1D1C	SRC4	1D28	SRCADR	1CF1	STCH	11B9
STKADR	11FD	STKFLG	14AB	STSC2	16D3	STSC3	16D9	STSCRN	16D1	STWND	1925
SWN2	195D	SWN3	1967	SWN4	196F	SWN4A	1982	SWN5	1996	TA	1629
TD	162D	THEX	191A	TILT	1391	TPIN	C060	TPSW	162E	TPTR	1912
TS	162C	TTIME	001B	TUSTK	1226	TX	162A	TY	162B	UBA1	1244
UBA2	1247	UBP0	1241	UBP1	1243	UBP2	1246	UBP3	125A	UBP4	1268
UBPD	1228	UDSD	11C5	UDSD1	11DA	UDSD2	11F8	UDSD3	11FC	UDSD4	121E
UPDW	1858	UPDW2	1867	UPDW2A	1869	UPDW2B	1885	UPDW3	18B5	UPDW3A	18C3
UPDW4	18F0	URD	114C	URD0	1165	URD1	116A	URD2	118E	URDT	1227
URTS	1286	USTK	1112	V2	1670	V3	1676	V4	167A	V5	1682
V7	1695	V8	16B2	VE	1692	VIEW	1663	W2	17CA	W3	17D3
W4	17DC	W5	17E5	W6	17EE	W7	17F7	W8	1800	W9	1809
WA	1812	WADR	1849	WB	181B	WC	1824	WCNT	1A10	WD	182D
WDTBL	17C1	WE	1836	WF	183F	WFADR	1926	WINDOW	1708	WN2	171E
WN3	1732	WN4	173D	WN5	1748	WN6	175B	WN7	1772	WN8	179D
WNAME	184B	WNO	1857	WPTR	17B6	WTBL	1851	WTILT	173A	WTYPE	1848
WUPTR	186A	WUPTR2	18B6	X	1110	Y	1111				

APPENDIX E

SUMMARY OF ASSEMBLER OPERATIONS AND PSEUDO-OPS

SUMMARY OF ASSEMBLER OPERATIONS AND PSEUDO-OPS

The assembly program operates under control of 9 keyboard commands which direct its operations. The system indicates its readiness to accept a command by displaying the prompt character ">" at the left end of the terminal device line. All system commands are terminated by period (.). The commands, shown here with the prompt character, are:

>/. Initialize the system. Any program in the source buffer is lost.

>M. Set memory limits. The M command displays the decimal number of memory words available in the source buffer and the hexadecimal address of the highest available word. The system is configured originally for 48K and the M command will result in the initial display:

31504 LEFT EOB = BFFF

The cursor period will remain at the right end of this displayed line waiting for a response from the user. If a period is typed nothing changes. If a hexadecimal number terminated by a period is typed the upper memory limit will be changed to that value. This will be necessary for systems having less than 48K memory.

>Ufn. Use peripheral n for function f. The operation of this command is described in detail on pages 42 through 45.

>An. Add text to source buffer after line n. If N is omitted or zero the text which follows will be inserted before any that may be in the buffer currently. Using the symbol F in place of a line number adds text after the last current line. The addition of text is

terminated when a period is used as the first character of a line. The system will respond with a prompt, ready to accept another command. During typing of a line characters may be eliminated by backspacing. If backspace ($88) appears as the last character of a line the entire line is ignored. Lines above the ones added are renumbered.

>Dn. or >Dn m. Delete text. Line n or lines n through m inclusive are deleted from the source buffer. Lines above the deleted ones are renumbered. >Dn F. deletes from line n through the end of the buffer.

>Ln. or >Ln m. List text. Line n or lines n through m inclusive are listed on the system list device which was specified by the U command. The lines are listed in formatted form with line numbers.

>Wn. or >Wn m. Write text to external storage medium. Line n or lines n through m inclusive are written to the external medium specified by the U command. The text so written is in exactly the form in which it was typed, with no superfluous blanks.

>R. Read source text into buffer from external medium. Text lines are read from the device specified by the U command. The text which is read is <u>added</u> to any which may already be in the source buffer. If this is not desired clear the buffer first with a / command.

>#E. Error scan. A full assembly of the program in the buffer is performed. Nothing is listed except lines which contain errors detectable by the assembler. This is a fast way to eliminate clerical errors like misspelled

labels and mnemonics.

>#O. Assemble for object code. A full assembly of the program in the source buffer is performed and the object code sent to the device specified by the U command.

>#Ln m. Assemble for listing. The program in the source buffer is assembled and listed, including symbol table, on the listing device specified by the U command. >#L1 F. produces the entire listing. The line limit option is provided to allow partial listing of programs for which significant sections of an earlier listing are still valid. This can save a lot of printing time if used carefully. Assembly to the Apple screen produces 12 lines at a time. The next 12 are viewed by tapping the space bar. To change this number see appendix C.

SUMMARY OF ASSEMBLER OPERATIONS AND PSEUDO-OPS

ASSEMBLER PSEUDO-OPS AND SYNONYMS

The assembler provides a number of convenience features designed to save writing and cut down the amount of meaningless detail and repetition required of the programmer. Some of these include compound operations which assemble as two or more instructions. These operations are:

ADD - Assembles as CLC + ADC. ADD accepts all of the addressing modes accepted by ADC.

SUB - Assembles as SEC + SBC. SUB accepts all of the addressing modes accepted by SBC.

INA - Increment A. Assembles as CLC + ADC #1. Note that, unlike INX and INY, INA changes the V and C flags.

DEA - Decrement A. Assembles as SEC + SBC #1. Unlike DEX and DEY, DEA changes the V and C flags.

CLA - Clear A. Assembles as LDA #0.

CMA - Ones complement A. Assembles as EOR #FF.

TCA - Twos complement A. Assembles as CLC + EOR #FF + ADC #1. Note that this <u>will</u> set overflow if the operand is $80, the maximum negative number.

To aid in the creation and manipulation of pointers and double precision counts the assembly system provides two compound operations. They are:

```
CDB  PTR,ADR
```

and

```
IDB  LOC
```

CDB creates a double precision constant at location PTR and PTR+1. The value of this constant is ADR. To create a pointer in scratchpad location BPTR to a location named BENNY elsewhere in memory write:

```
CDB  BPTR,BENNY
```

The contents of BENNY may now be accessed indirectly by:

```
LDA  *BPTR,Y
```

providing Y has been set to zero. The pointer BPTR need not be in scratchpad. It may <u>not</u> straddle the scratchpad boundary, i.e., the low half in $FF and the high half in $100. This will draw an error message from the assembler. The above use of CDB assembles as:

```
LDA  #<BENNY
STA  BPTR
LDA  #>BENNY
STA  BPTR+1
```

CDB assembles to either 8 or 10 words, depending on the location of the pointer being created. It may also be used to create numeric constants, e.g.:

```
CDB  GROVER,-5000
```

will create the constant -5000 in GROVER and GROVER+1 in proper double precision form. The above assembles

as:

```
        LDA  #<-5000
        STA  GROVER
        LDA  #>-5000
        STA  GROVER+1
```

The IDB operation performs a double precision incrementation of the number named in its operand. The constant GROVER created above can be incremented by:

```
        IDB  GROVER
```

which assembles to:

```
        INC  GROVER
        BNZ  loc
        INC  GROVER+1
loc     ...
```

When control reaches location loc above the zero flag reflects the true result of the double incrementation. Z is set if and only if both halves are zero, i.e., if the number was $FFFF, or -1, before the incrementation. IDB can thus be used for loop control with large counts as well as for control of pointers.

BRANCH SYNONYMS

As a convenience for use after compare instructions the assembler supports a number of synonyms. For use after comparison of eight bit unsigned magnitudes are the following:

BGE - Branch if A greater than or equal to memory. BGE assembles as BCS.

BLS - Branch if A less than memory. BLS assembles as BCC.

For use after comparison of signed numbers are:

B>= - Branch if A greater than or equal to memory. B>= assembles as BP.

B< - Branch if A less than memory. B< assembles as BMI.

The assembler also supports the following synonyms:

BZ - Branch if zero. BZ assembles as BEQ.

BNZ - Branch if nonzero. Assembles as BNE.

SUMMARY OF ASSEMBLER OPERATIONS AND PSEUDO-OPS

LOCATION COUNTER CONTROL

The system supports the ORG pseudo-op with its standard meaning. END and EQU are also supported with their standard meanings.

Scratchpad data allocation is controlled by a pair of pseudo-ops, SPD and ESP. These pseudo-ops must bracket all definitions of scratchpad data locations. No instruction may appear within the range of an SPD, only data defining operations. There may be any number of SPD-ESP pairs in a program.

The data defining pseudo-ops and their meanings are:

```
DATA  -1,'A',@77,%10100101,$B5
```

The word DATA in the instruction field causes the contents of the operand field to be assembled as a string of constants occupying successively higher memory addresses. The data types are shown above. No prefix is taken to means decimal, as in the -1. Enclosure between apostrophes means ASCII. Prefix @ means octal. Prefix % specifies binary and prefix $ means hexadecimal.

ASCII strings may be created with the ASC pseudo-op, i.e.:

```
ASC  'THIS IS AN ASCII STRING'
```

The string must be enclosed by apostrophes as shown.

Double precision data are specified by the DBL pseudo-op, as in:

```
DBL  -1,LOC1,$FFFE
```

In addition to supporting the data types supported by DATA above, DBL allows symbolic data like the LOC1 item above. A constant is created in this case which is a

pointer to LOC1.

Besides allowing the use of asterisk in the first column to denote a line of all commentary, the system supports a REM pseudo-op. REM is a documentation device which aligns its own operand field with the comment field of normal instructions. Thus, if a comment following an instruction is too long to fit in the comment field it can be continued as the field of a REM. This aligns the commentary in a neat readable form.

To aid in identifying locations which are to change during program execution the assembler supports 2 data defining operations, *** and $$$. When *** is used in the instruction field a word of zero is created in the object program. When $$$ is used two words of zero are created. These are logically no different from:

```
DATA 0
```

and

```
DBL  0
```

but allows a visual means of detecting "don't care" cells while scanning the program listing. Labels may be appended to *** and $$$ in the normal way.

Finally, a banner can be printed at the top of each page of the assembly listing by the TITL pseudo-op. The operand field of this pseudo-op will be used as a page heading, being printed just to the right of the page number

APPENDIX F

SUMMARY OF DEBUG PROGRAM OPERATIONS

SUMMARY OF DEBUG PROGRAM OPERATIONS

The debug program operates under the control of ten commands, all but one of these followed by a hexadecimal number or numbers. The terminating character for all commands is period (.).

In the discussion which follows a number of symbols will be used. Their values are:

aaaa or bbbb - A 4 digit hexadecimal number which represents a memory address.

hh - A two digit hexadecimal number typed by debug.

xx - A two digit hexadecimal number typed by the user.

@ - The debug cursor, a flashing period.

Leading zeros need never be typed in, though they will do no harm if they are typed. As far as debug is concerned 0003 and 3 are the same. Further, leading hexadecimal digits beyond the number called for are ignored, e.g., in a place requiring a two digit response from the user only the last two digits typed count. Thus, if an error is made in hexadecimal input it can be corrected by just typing the correct value. The same rule is true for 4 digit addresses. Only the last four typed digits count.

MEMORY MODIFICATION

Memory may be modified by the M and F commands. The M command allows inspection of memory one word at a time. The form of the command is:

#Maaaa.

The debug responds with the following display in the command line:

#Maaaa hh * @

where hh is the content of the memory location aaaa and * is the ASCII value of the number hh. Debug then waits for a response from the user. If a hexadecimal number is typed, followed by a terminating character the typed number replaces hh as the content of location aaaa. If the content of aaaa is to be changed to xx the line looks like this:

#Maaaa hh * xx@

If location aaaa is not to be changed the xx is omitted. A terminating character must now be typed. There are three choices: period, space and comma. If a period is typed the M command is terminated. If a space is typed the value of aaaa+1 is displayed for inspection and possible modification. A series of higher and higher addresses can thus be inspected and modified by using space as a terminating character. The inspection can be ended at any time by using period as a terminator.

If comma is used instead of space the content of aaaa-1 is displayed for inspection. The comma works exactly as does the space except for the direction of advance. A series of lower and lower addresses can be inspected/modified by using the comma as terminator. As before, the inspection is ended by using period.

The F command is used to fill a block of memory with a constant value. Its form is:

#Faaaa bbbb xx.

Address bbbb must be greater than aaaa. These addresses are the inclusive limits of the fill and xx is the value that is to be stored into memory.

SUMMARY OF DEBUG PROGRAM OPERATIONS

MEMORY SEARCHES

Blocks of memory may be searched for a value by the S command. Its form is:

#Saaaa bbbb xx.

where aaaa and bbbb are the inclusive limits, aaaa<bbbb. The block will be searched beginning at aaaa. If the value xx is found the search is suspended with the display looking like this:

nnnn xx@

where nnnn is the address at which the number xx was found. typing space continues the search.

TRANSFER OF CONTROL

Control is transferred to a program in memory by the J command, whose form is:

#Jaaaa.

All machine registers are loaded from the debug's registers before the jump and the screen is set as per the V command specification.

BREAKPOINTING

Breakpointing is done by the B command, whose form is:

#Baaaa bbbb.

This causes a breakpoint to be set at bbbb and control

to be transferred to aaaa with the machine registers set to the debug register values and screen setting. When control reaches bbbb the values of the machine registers replace debug's registers, the breakpoint at bbbb is restored and the normal debug screen is displayed. The screen display is updated and the message:

BKPT REACHED

displayed momentarily.

If control fails to reach the breakpoint stop the program in progress with the RESET key and transfer control to the beginning of debug. The breakpoint will be restored and the message:

BKPT FAILED

displayed momentarily.

If you want to continue from a previous successful breakpoint only one address need be typed. If the last breakpoint was successful and if only one address is typed for this breakpoint, debug will assume that the new transfer address is the old breakpoint address. The single address typed in will be used as the breakpoint address. This saves a lot of redundant typing when a series of breakpoints is to be executed.

REGISTER MODIFICATION

Debug's registers are modified by the R command whose form is:

#Rnxx.

where n is one of the register names A, X, Y or S. The program status word uses another format. Debug allows individual modification of the PSW bits. The form of the command for PSW modification is:

#RPxb.

where x is one of the letters N, V, B, D, I, Z or C and b is a bit value, 0 or 1. Any number of PSW bits can be set or cleared in a single RP command, e.g.:

#RPN1V0Z1C1.

sets the N, Z and C bits to 1 and the V bit to 0.

DUMPING OF MEMORY

Blocks of memory may be dumped to a parallel printer in slot 3 by the D command. Its form is:

#Daaaa bbbb.

The block will be printed on the parallel printer 24 words per line with the address of the leftmost word of the line displayed at the left of the page.

WINDOW CREATION

Memory windows may be created by the W command, whose form is:

#Wn,T,ADR,name.

where n is the window number, a single hexadecimal digit 1 - F, T is the window type, ADR is the hexadecimal address of the window and name is the string of six or fewer characters which is to be displayed in the window. The name is optional and may be omitted. Its purpose is to allow direct correspondence of the data displayed in the window with the label given this item in the assembly listing. The window types are:

SUMMARY OF DEBUG PROGRAM OPERATIONS

A - ASCII

S - Single precision decimal

D - Double precision decimal

H - Hexadecimal

P - Pointer

Any window may be cleared by using C as the window type, i.e.:

#Wn,C.

clears window n.

SCREEN SPECIFICATION

The screen to be used when the next J or B command is executed is specified by the V command, whose form is:

#Vsn.

where s is one of the letters T, L or H and n is a single digit. The possible values of sn are:

T1	Text screen 1
T2	Text screen 2
L1	LORES screen 1
L2	LORES screen 2
L3	Mixed LORES screen 1
L4	Mixed LORES screen 2
H1	HIRES screen 1
H2	HIRES screen 2
H3	Mixed HIRES screen 1
H4	Mixed HIRES screen 2

When the terminating period is typed the chosen screen is immediately displayed. To return to the normal debug screen type VN. The normal screen will return but the screen which you chose will be the one displayed during breakpoints.

LOADING OBJECT OUTPUT

Object output of the assembly program is loaded by the L command, i.e.:

#L.

Debug responds with:

OBJECT READY, SPACE

Place the cassette containing the object program in the cassette reader and press the space bar. The word LOAD will appear at the right side of the screen. The screen position below the L in LOAD will begin to flash in a few seconds and will do so intermittently while the load is going on. When the load is finished the number of records loaded will appear in the space below the word LOAD.

$$$ 86,101
*** 101
< 76
> 76
ADC 54-55
ADD 55-56
addition
  binary 14-16
  decimal 181
  floating point 189-193, 199-201
address 2
  direct 46-47
  immediate 47
  indexed 49-50
  indirect 51-53
  preindexed 51
  postindexed 52
AND 9,67
arguments 140
  in registers 141
  addresses in scratchpad 141-142
  addresses in line 142
arithmetic shift 22
array 144
  search 158-161
ASC 100-101
ASCII codes 179
ASL 81
ASLA 81
assembler
  definition of 30
  structure of 313-321
  errors 320
  first pass functions 316
  second pass functions 317
  third pass functions 319
  error flags 43-44
B< 65
B>= 64
BCC 61
BCD 180
  addition 181,186
  division 180-182
  multiplication 186
  subtraction 180-181
BCS 61
BEQ 61
BGE 65
binary
  addition 14-16
  coded decimal 180
  division 116-120
  multiplication 109-110
  multiplication by constant 104-107
  multiply, double precision 114-115
  multiply, signed 112-114
  subtraction 16-18
  system of numbers 7-26
BLS 65
BMI 61
BNE 61
BNZ 62
borrow 17
BPL 61
branch instruction 60-61
breakpoint address 296
breakpointing 296-297
BRK 276
bubble sort 149-151
BVC 62
BVS 62
BZ 62
carry 15-16
  use of 84-85
  bit 19
cartouche of Ptolemy 258
CDB 77
Champollion, Jean 258
checksum 146
cipher 259
  substitution 260
  key 267
CLA 80
CLC 54
CLD 180
Cleopatra, Queen 257
CLI 275
CMA 80
CMP 28,66
comment
  field 32
complementation 100-102
conversion 10
  binary to hexadecimal 205-222
  BCD to binary 183-184
  double precision 205-208
  decimal to binary fraction 210-212
  binary to decimal integer 212
  binary to decimal, 213-216
    double precision
  binary to decimal fraction 216-218
  binary to ASCII 218-220
  of floating point by 183-184
    radix deflation method
CPX 220-222
CPY 67
cryptography 67
  monoalphabetic 259-272
    encipherment
  letter frequencies 261-262
  polyalphabetic 265-266
    encipherment
DATA 269-270
data transfer 100
  under interrupt control 284-289
DBL 84,100
DEA 80
debug program
  screen display 292

commands 292-295
memory dump (D) 295-296
memory inspect/modify (M) 294
screen control (V) 295
breakpointing (B) 296-297
register set (R) 297-298
screen display 299
memory search (S) 308
memory fill (F) 308
memory windowing (W) 309-311

debugging 291-312
DEC 79
decimal mode 180
device address 162
DEX 73
DEY 73
DI 275
direct addressing 46-47
displacement of
indexed instruction 154
disk, interference
by interrupt 284
divide fault 116
dividend 115
divisor 115
double precision 83
addition 85
subtraction 86-87
increment 90
decrement 92
logical shifts 93-94
arithmetic shifts 94-95
twos complement 97
multiplication 114-115

effective address 49
EI 275
Elizabeth I, Queen 261
END 98,317
EOR 67
EQU 101-102
equivalent decimal precision
of binary number of N bits 14
ESP 99,102
Exlusive OR 101-102
fixed point 188
range of numbers 19

floating point
representation 188
addition 189-193
subtraction 193
precision loss 190-192
number range 202
exponent bias 197
conversion 220-222
exponent 197-198
fraction 189
noisy mode 201
normalization 193
sign 198
true hexadecimal exponent 202
unnormalized 193

graphic output 237-258
memory addressing 238-239
to graphics printer 250-257

Henry VIII, King 262
hexadecimal system 24
IDB 91
image region 237
immediate mode 47
INA 80
INC 79
Inclusive OR 8
index register 50
indexed addressing 49-50
indirect addressing 51
input
Apple II keyboard 162-163
under interrupt control 285-287

instruction 4
execution cycle 4-5
field 31
register 3-4
interface 167
interrupt 273-290
definition of 274
service routine 276
device arm/disarm 275
enable 274
real time clock 278
disable 275
sequence of events 276

INX 73
INY 73
JMP 59-60
JSR 129
label field 31
LDA 47
LDX 47
LDY 47
LIFO stack 123
load 3
loader 146
logical difference 11
logical product 9
logical shift 21
logical sum 8
looping 74-75
LSR 81
LSRA 81
masking 10
memory 1-2
memory mapped I/O 170
minuend 16
Monte Carlo methods 225
multiple precision 83
addition 88-89
subtraction 86-87

multiplication, decimal 186
negation 18
NOP 306
number range
floating point 202
fixed point 19

object program 34
octal 23
ones complement 10-11
operand field 32
ORA 67

ORG 30-31
output
  to printer 166-169
  to CRT 170-177
  to graphics CRT 237-258
  under interrupt control 288-289
overflow 3
overflow bit 20
page zero 46
patching 306
peripherals 162
PHA 124
PHP 124
PLA 124
PLP 124
pointer 4,51
powers of two 13
processor status word 58-59
  B bit 58
  C bit 59
  D bit 58
  I bit 59
  N bit 58
  V bit 58
  Z bit 59

program counter 4
PSW 58-59
Ptolemy of Egypt, King 257-258
quotient 115
radix deflation method 220-222
random numbers
  uniform distribution 223
  power residue method 225-226
  generation of
  frequency distribution 228-229
  random BCD digits 230
  determination of pi 232-234

ready flag 163,167
real time clock 278
real time executive 290
registers 2-4
REM 102
remainder 115
RES 99-100
ROL 81
ROLA 81
ROR 81
RORA 81
rotary shift 22
RTC 278
RTI 278-279
RTS 129
SBC 56
scratchpad 46
SEC 56
SED 180
SEI 275
shifting 20-23
source program 32-36
source buffer 36
SPD 99,102
STA 48
stack 122
stack pointer 122
status word
  peripheral 166
  processor 58-59
store 3
Stuart, Mary, Queen of Scots 261-263
STX 150
STY 214
SUB 57
subroutine 130
subtraction, decimal 184
subtraction, floating point 193
subtrahend 16
table 144
table lookup 151-157
TAX 139
TAY 78
TCA 80
tens complement 185
TITL 102
transfer address 296
transfer vector 144
twos complement system 18
TXA 132
TXS 123
TYA 127
validity check 209
vector 144
Walsingham, Sir Francis 263
word 1
wordlength 1

To receive object code of the editor/assembler and debug programs fill out both parts of the coupon on the next page, including signature, and send it to:

Northern Technology Books
Box 62
Evanston, IL 60204

Please print or type all information. The lower part of the coupon will be used as your mailing label.

DISCLAIMER AND LICENSING AGREEMENT

Though the cassettes and paper tapes are prepared under stringent quality control, the great variability of physical parameters such as read head alignment makes it impossible to guarantee that they will be readable on any particular computer system, and the user so agrees by his signature on the coupon.

While the software and examples in the text have been prepared with all prudent caution and care and have been subjected to extensive testing, neither they nor any other computer program can be certified to be completely error free. Northern Technology Books and the author assume no liability for either the function of the programs or any direct or indirect damage which may occur as a consequence of their use, and the user so agrees by the purchase of this book.

By completing and returning the coupon on the facing page the purchaser of this book is granted license for his individual use of the programs supplied. This individual use includes the making of a reasonable number of duplicates for backup purposes, but excludes the transmission of the programs to any other person or corporate or business entity, and, for value received, the user so agrees by his signature on the coupon.

COUPON

Send this coupon only. No copies will be accepted. Both parts and the signature must be completed. Please allow 45 days for delivery.

_ Apple II cassette

_ Paper Tape

_ Apple II disk

The disk can be run on an Apple II with 32K or more of RAM. If the disk is selected include check or money order for $7.50. U.S. Funds only.

6502-0

Name_____

Street_____

City/State/Zip_____

System (Apple, Pet, etc)_____

Memory Size (RAM)_____

Signature_____

6502-0

Name_____

Street_____

City/State/Zip_____

To receive object code of the editor/assembler and debug programs fill out both parts of the coupon on the next page, including signature, and send it to:

Northern Technology Books
Box 62
Evanston, IL 60204

Please print or type all information. The lower part of the coupon will be used as your mailing label.

DISCLAIMER AND LICENSING AGREEMENT

Though the cassettes and paper tapes are prepared under stringent quality control, the great variability of physical parameters such as read head alignment makes it impossible to guarantee that they will be readable on any particular computer system, and the user so agrees by his signature on the coupon.

While the software and examples in the text have been prepared with all prudent caution and care and have been subjected to extensive testing, neither they nor any other computer program can be certified to be completely error free. Northern Technology Books and the author assume no liability for either the function of the programs or any direct or indirect damage which may occur as a consequence of their use, and the user so agrees by the purchase of this book.

By completing and returning the coupon on the facing page the purchaser of this book is granted license for his individual use of the programs supplied. This individual use includes the making of a reasonable number of duplicates for backup purposes, but excludes the transmission of the programs to any other person or corporate or business entity, and, for value received, the user so agrees by his signature on the coupon.

COUPON

Send this coupon only. No copies will be accepted. Both parts and the signature must be completed. Please allow 45 days for delivery.

_ Apple II cassette

_ Paper Tape

_ Apple II disk

The disk can be run on an Apple II with 32K or more of RAM. If the disk is selected include check or money order for $7.50. U.S. Funds only.

6502-0

Name_____

Street_____

City/State/Zip_____

System (Apple, Pet, etc)_____

Memory Size (RAM)_____

Signature_____

6502-0

Name_____

Street_____

City/State/Zip_____

About the Author

Walt Weller is an application software specialist working in industrial, medical and educational uses of small computers. In addition to this book he is the author of PRACTICAL MICROCOMPUTER PROGRAMMING: THE M6800, PRACTICAL MICROCOMPUTER PROGRAMMING: THE Z80 and Assembly Level Programming for Small Computers (Lexington, 1975). He is also coauthor of PRACTICAL MICROCOMPUTER PROGRAMMING: THE INTEL 8080 and An Editor/Assembler System for 8080/8085 Based Computers. In addition to writing and software development, his activities include seminars given at the University of Wisconsin Engineering Extension in Milwaukee. He currently resides in the Chicago area.

Other titles in this series available prepaid, are

PRACTICAL MICROCOMPUTER PROGRAMMING: THE INTEL 8080 $23.95

PRACTICAL MICROCOMPUTER PROGRAMMING: THE M6800 $23.95

PRACTICAL MICROCOMPUTER PROGRAMMING: THE Z80 $32.95

Order directly from:

Northern Technology Books
Box 62
Evanston, IL 60204

PRACTICAL MICROCOMPUTER PROGRAMMING: THE 6502

WELLER

PRACTICAL MICROCOMPUTER PROGRAMMING:

THE 6502

W.J. WELLER