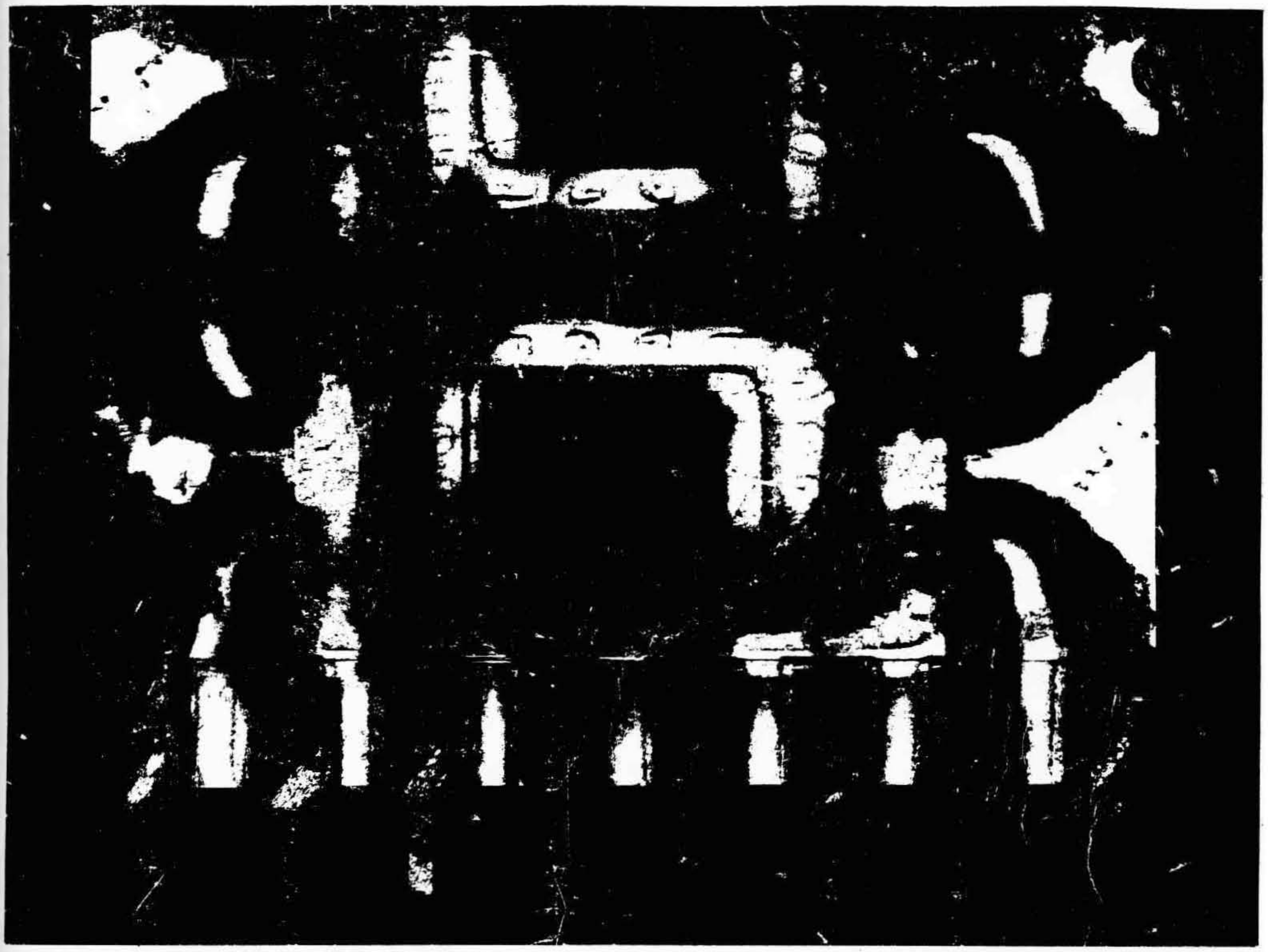


MACHINE LANGUAGE PROGRAMMING FOR THE "8008" and similar microcomputers

SCELBI COMPUTER CONSULTING INC.

MACHINE LANGUAGE PROGRAMMING

FOR THE '8 0 0 8'

(AND SIMILAR MICROCOMPUTERS)

Author:

Nat Wadsworth

MACHINE LANGUAGE PROGRAMMING

FOR THE '8 0 0 8'

(AND SIMILAR MICROCOMPUTERS)

****

TABLE OF CONTENTS

****

INTRODUCTION

This manual is about machine language programming methods and techniques for the '8008' and similar microprocessors. Machine language programming is the most fundamental type of computer programming possible. It is by far the most efficient method, in terms of utilization of the machine's capabilities, with which to program or set up a microcomputer to perform a task. Machine language programming is, on the other hand, the most demanding method of computer programming in terms of human endeavor and skill. However, the fundamental skills and techniques necessary for machine language programming can be applied to virtually any level of computer programming. A clear understanding of machine language programming will give one great insight into higher level language programming.

Machine language programming is the actual step-by-step programming of the computer using the machine codes and memory addresses that are used by the computer directly. It is considerably more detailed than programming in a high level language such as FORTRAN or BASIC. It is the level of programming from which those high level languages must be developed. In fact, if one learns how to develop programs in machine language, one will have the basic skills necessary for developing higher level languages. (That is a tremendous asset over one who only knows how to program using higher level languages.)

The primary reason for having a manual devoted to machine language programming for microprocessors is because this method is by far the most efficient method for packing a program into a small amount of memory. As user's know, memory elements cost a good amount of money. The more one can program into a given amount of memory, the less memory required for a given task. Thus, the more one can do with a low cost machine. High level languages require much more memory because of two major reasons. First, a large amount of memory must be used by the high level language itself. Second, higher level languages must convert user statements or commands to machine language codes. They generally cannot do this any where near as efficiently (memory usage wise) as a trained human programmer.

Another reason for discussing machine language programming at length is because it is the only method whereby many capabilities of the machine can be efficiently utilized. This is particularly true for "real-time" and I/O operations. Many users will want to utilize their microprocessors for unique applications. The contents of this manual will present many ideas and concepts for these people to apply to their individual applications.

Machine language programming in general is nowhere as difficult to learn as many people might tend to think when first introduced to the subject. This is especially true for the '8008' type microprocessor. There are many fundamental concepts that can be readily learned. Once this has been accomplished the novice is on the way to developing original solutions to programming problems that may be of special interest to the individual.

Computer programming, and machine language programming in particular, is in many respects an art, and in other respects a very rigid science. The fun part, and what can be considered artistic, is that individuals can tailor or fashion series of instructions to accomplish a particular task in a variety of ways. The scientific part of programming involves acquiring some basic skills and knowledge about what can and cannot be done. At a higher level this requires an understanding of basic mathematic algorithms and procedures that can be readily applied using computer techniques. Some of the basic skills include knowing just what the available machine instructions are, and some of the most frequently used combinations of instructions that will perform frequently required tasks. These skills are as fundamental as a painter knowing the primary colors and how to combine them to create the commonly used secondary colors. However, like the painter who combines the basic pigments, beyond a certain point the task of computer programming becomes a highly creative individualistic art. It is an art in which one can constantly gain new skills and ability. A high school student or a college professor can both find equally rewarding challenges in computer programming. There are often many different ways to program a computer to perform a given task and many "trade-offs" to consider when developing a program. (Such as how much memory to use, what functions have priority, and how much burden to place on the human operator when the program is operating.) Individuals soon learn to capitalize on the aspects considered most important for the specific applications at hand and to develop their own personal methods for handling various types of programming tasks.

Remember as you read this manual that there are many other ways of programming a computer to perform many of the example programs illustrated. Don't be afraid to develop your own solutions. See if they work as planned. Practice being a creative programmer! By the time you have completed absorbing and understanding the contents of this publication you should be well equipped to develop programs of your own. You will thus be in a position to reap greater benefits from your microprocessor than just being able to operate programs that other people have prepared.

The first chapter of this manual contains a detailed presentation of the instruction set that the '8008' CPU is capable of performing. It goes almost without saying, that the first step towards becoming a proficient machine language programmer is to become thoroughly familiar with all the types of instructions that the machine being utilized can execute. One should especially learn about any special conditions that apply to the execution of specific types of commands. The lead-off chapter presents a comprehensive explanation of all the instructions in the '8008' repertoire along with the mnemonics and machine codes. The reader should become quite familiar with the information presented there before going further in this manual. (At least to the point where one can rapidly locate any class of instructions in the chapter in order to refresh one's memory on just how an instruction operates. Additionally, such familiarity will enable one to be able to quickly locate machine codes when one is preparing the final version of a machine language program!)

THE '8008' CPU INSTRUCTION SET

The '8008' microprocessor has quite a comprehensive instruction set that consists of 48 basic instructions, which, when the possible permutations are considered, result in a total set of about 170 instructions.

The instruction set allows the user to direct the computer to perform operations with memory, with the seven basic registers in the CPU, and with INPUT and OUTPUT ports.

It should be pointed out that the seven basic registers in the CPU consist of one "accumulator," a register that can perform mathematical and logic operations, plus an additional six registers, which, while not having the full capability of the accumulator, can perform various useful operations. These operations include the ability to hold data, serve as an "operator" with the accumulator, and increment or decrement their contents. Two of these six registers have special significance because they may be used to serve as a "pointer" to locations in memory.

The seven CPU registers have arbitrarily been given symbols so that we may refer to them in an abbreviated language. The first register is designated by the symbol 'A' in the following discussion and will be referred to as the "accumulator" register. The next four registers will be referred to as the 'B,' 'C,' 'D' and 'E' registers. The remaining two special memory pointing registers shall be designated the 'H' (for the HIGH portion of a memory address) and the 'L' (for the LOW portion of a memory address) registers.

The CPU also has several "flip-flops" which shall be referred to as "FLAGS." The flip-flops are set as the result of certain operations and are important because they can be "tested" by many of the instructions with the instruction's meaning changing as a consequence of the particular status of a FLAG at the time the instruction is executed. There are four basic flags which will be referred to in this manual. They are defined as follows:

The 'C' flag refers to the carry bit status. The carry bit is a one unit register which changes state when the accumulator overflows or underflows. This bit can also be set to a known condition by certain types of instructions. This is important to remember when developing a program because quite often a program will have a long string of instructions which do not utilize the carry bit or care about its status, but which will be causing the carry bit to change its state from time-to-time. Thus, when one prepares to do a series of operations that will rely on the carry bit, one often desires to set the carry bit to a known state.

The 'Z' for zero flag refers to a one unit register that when desired will indicate whether the value of the accumulator is exactly equal to zero. In addition, immediately after an increment of decrement of the B, C, D, E, H or L registers, this flag will also indicate whether the increment or decrement caused that particular register to go to zero.

The 'S' for sign flag refers to a one unit register that indicates whether the value in the accumulator is a positive or negative value (based on two's complement nomenclature). Essentially, this flag monitors the most significant bit in the accumulator and is "set" when it is a one.

The 'P' flag refers to the last flag in the group which is for indicating when the accumulator contains a value which has even parity. Parity is useful for a number of reasons and is usually used in conjunction with testing for error conditions on words of data especially when transferring data to and from external devices. Even parity occurs when the number of bits that are a logic one in the accumulator is an even value. Zero is considered an even value for this purpose. Since there are eight bits in the accumulator, even parity will occur when zero, two, four or six bits are in the logic one condition regardless of what order they may appear in within the register.

It is important to note that the Z, S, and P flags (as well as the previously mentioned C flag) can all be set to known states by certain instructions. It is also important to note that some instructions do not result in the flags being set so that if the programmer desires to have the program make decisions based on the status of flags, the programmer should ensure that the proper instruction, or sequence of instructions is utilized. It is particularly important to note that load register instructions do not by themselves set the flags. Since it is often desirable to obtain a data word (that is, load it into the accumulator) and test its status for such parameters as whether or not the value is zero, or a negative number, and so forth, the programmer must remember to follow a load instruction by a logical instruction (such as the NDA - "and the accumulator") in order to set the flags before using an instruction that is conditional in regards to a flag's status.

The description of the various types of instructions available using an '8008' CPU which follows will provide both the machine language code for the instruction given as three octal digits, and also a mnemonic name suitable for writing programs in "symbolic" type language which is usually easier than trying to remember octal codes! It may be noted that the symbolic language used is the same as that originally suggested by Intel Corporation which developed the '8008' CPU-on-a-chip. Hence users who may already be familiar with the suggested mnemonics will not have any relearning problems and those learning the mnemonics for the first time will have plenty of good company. If the programmer is not already aware of it, the use of mnemonics facilitates working with an "assembler" program when it is desired to develop relatively large and complex programs. Thus the programmer is urged to concentrate on learning the mnemonics for the instructions and not waste time memorizing the octal codes. After a program has been written using the mnemonic codes, the programmer can always use a lookup table to convert to the machine code if an assembler program is not available. It's a lot easier technique (and less subject to error) than trying to memorize the 170 or so three digit combinations which make up the machine instruction code set!

The programmer must also be aware, that in this machine, some instructions require more than one word in memory. "Immediate" type commands require two consecutive words. JUMP and CALL commands require three consecutive words. The remaining types only require one word.

The first group of instructions to be presented are those that are used to load data from one CPU register to another, or from a CPU register to a word in memory, or vice-versa. This group of instructions requires just one word of memory. It is important to note that none of the instructions in this group affect the flags.

LOAD DATA FROM ONE CPU REGISTER
TO ANOTHER CPU REGISTER

MNEMONIC	MACHINE CODE
LAA	300
LBA	310
.	.
.	.
LAB	301

The load register group of instructions allows the programmer to move the contents of one CPU register into another CPU register. The contents of the originating (from) register is not changed. The contents of the destination (to) register becomes the same as the originating register. Any CPU register can be loaded into any CPU register. Note that loading register A into register A is essentially a NOP (no operation) command. When using mnemonics the load symbol is the letter L followed by the "to" register and then the "from" register. The mnemonic LBA means that the contents of register A (the accumulator) is to be loaded into register B. The mnemonic LAB states that register B is to have its contents loaded into register A. It may be observed that this basic instruction has many variations. The machine language coding for this instruction is in the same format as the mnemonic code except that the letters used to represent the registers are replaced by numbers that the computer can use. Using octal code, the seven CPU registers are coded as follows:

Register A = 0
Register B = 1
Register C = 2
Register D = 3
Register E = 4
Register H = 5
Register L = 6

Also, since the machine can only utilize numbers, the octal number '3' in the most significant location of a word signifies that the computer is to perform a "load" operation. Thus, in machine coding, the instruction for loading register B with the contents of register A becomes '310' (in octal form). Or, if one wanted to get very detailed, the actual binary coding for the eight bits of information in the instruction word would be '11 001 000.' It is important to note that the load instructions do not affect any of the flags.

LOAD DATA FROM ANY CPU REGISTER TO A LOCATION IN MEMORY

LMA	370
LMB	371
LMC	372
LMD	373
LME	374
LMH	375
LML	376

This instruction is very similar to the previous group of instructions except that now the contents of a CPU register will be loaded into a specified memory location. The memory location that will receive the contents of the particular CPU register is that whose address is specified by the contents of the CPU H and L registers at the time the instruction is executed. The H CPU register specifies the HIGH portion of the address desired, and the L CPU register specifies the LOW portion of the address into which data from the selected CPU register is to be loaded. Note that there are seven different instructions in this group. Any CPU register can have its contents loaded into any location in memory. This group of instructions does not affect any of the flags.

LOAD DATA FROM A MEMORY LOCATION TO ANY CPU REGISTER

LAM	307
LBM	317
LCM	327
LDM	337
LEM	347
LHM	357
LLM	367

This group of instructions can be considered the opposite of the previous group. Now, the contents of the word in memory whose address is specified by the H (for HIGH portion of the address) and L (LOW portion of the address) registers will be loaded into the CPU register specified by the instruction. Once again, this group of instructions has no affect on the status of the flags.

LOAD IMMEDIATE DATA INTO A CPU REGISTER

LAI	006
LBI	016
LCI	026
LDI	036
LEI	046
LHI	056
LLI	066

An IMMEDIATE type of instruction requires two words in order to be completely specified. The first word is the instruction itself. The second word, or "immediately following" word, must contain the data upon which "immediate" action is taken. Thus, a load IMMEDIATE instruction in this group means that the contents of the word immediately following the instruction word is to be loaded into the specified register. For example, a typical load immediate instruction would be LAI 001. This would result in the value 001 (octal) being placed in the A register when the instruction was executed. It is important to remember that all IMMEDIATE type instructions MUST be followed by a data word. An instruction such as LDI by itself would result in improper operation because the computer would assume the next word contained data. If the programmer had mistakenly left out the data word, and in its place had another instruction, the computer would not realize the operator's mistake. Hence the program would be fouled-up! Note too, that the load immediate group of instructions does not affect the flags.

LOAD IMMEDIATE DATA INTO A MEMORY LOCATION

LMI	076

This instruction is essentially the same as the load immediate into the CPU register group except that now, using the contents of the H and L registers as "pointers" to the desired address in memory, the contents of the "immediately following word" will be placed in the memory location specified. This instruction does not affect the status of the flags.

The above rather large group of LOAD instructions permits the programmer to direct the computer to move data about. They are used to bring in data from memory where it can be operated on by the CPU. Or, to temporarily store intermediate results in the CPU registers during complicated and extended calculations, and of course allow data, such as results, to be placed back into memory for long term storage. Since none of them will alter the contents of the four CPU flags, these instructions can be called upon to set up data before instructions that may affect or utilize the flag's status are executed. The programmer will use instructions from this set frequently. The mnemonic names for the instructions are easy to remember as they are well ordered. The most important item to remember about the mnemonics is that the TO register is always indicated first in the mnemonic, and then the FROM register. Thus LBA equals "load TO register B FROM register A.

INCREMENT THE VALUE OF A CPU REGISTER BY ONE

INB	010
INC	020
IND	030
INE	040
INH	050
INL	060

This group of instructions allows the programmer to add one to the present value of any of the CPU registers except the accumulator. (Note carefully that the accumulator can NOT be incremented by this type of instruction. In order to add one to the accumulator a mathematical addition instruction, described later, must be used.) This instruction for incrementing the defined CPU registers is very valuable in a number of applications. For one thing, it is an easy way to have the L register successively "point" to a string of locations in memory. A feature that makes this type of instruction even more powerful is that the result of the incremented register will affect the Z, S, and P flags. (It will not change the C or "carry"flag.) Thus, after a CPU register has been incremented by this instruction, one can utilize a flag test instruction (such as the conditional JUMP and CALL instructions to be described later) to determine whether that particular register has a value of zero (Z flag), or if it is a negative number (S flag), or even parity (P flag). It is important to note that this group of instructions, and the decrement group (described in the next paragraph) are the only instructions which allow the flags to be manipulated by operations that are not concerned with the accumulator (A) register.

DECREMENT THE VALUE OF A CPU REGISTER BY ONE

DCB	011
DCC	021
DCD	031
DCE	041
DCH	051
DCL	061

The DECREMENT group of instructions is similar to the INCREMENT group except that now the value one will be subtracted from the specified CPU register. This instruction will not affect the C flag. But, it does affect the Z, S, and P flags. It should also be noted that this group, as with the increment group, does not include the accumulator register. A separate mathematical instruction must be used to subtract one from the accumulator.

ARITHMETIC INSTRUCTIONS USING THE ACCUMULATOR

The following group of instructions allow the programmer to direct the computer to perform arithmetic operations between other CPU registers and the accumulator, or between the contents of words in memory and the accumulator. All of the operations for the described addition, subtraction, and compare instructions affect the status of the flags.

ADD THE CONTENTS OF A CPU REGISTER TO THE ACCUMULATOR

ADA	200
ADB	201
ADC	202
ADD	203
ADE	204
ADH	205
ADL	206

This group of instructions will simply ADD the present contents of the accumulator register to the present value of the specified CPU register and leave the result in the accumulator. The value of the specified register is unchanged except in the case of the ADA instruction. Note that the ADA instruction essentially allows the programmer to double the value of the accumulator (which is the A register)! If the addition causes an overflow or underflow then the carry (C flag) will be affected.

ADD THE CONTENTS OF A CPU REGISTER PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACA	210
ACB	211
ACC	212
ACD	213
ACE	214
ACH	215
ACL	216

This group is identical to the previous group except that the content of the carry flag is considered as an additional bit (MSB) in the specified CPU register. The combined value of the carry bit plus the contents of the specified CPU register are added to the value in the accumulator. The results are left in the accumulator. Again, with the exception of the ACA instruction, the contents of the specified CPU register are left unchanged. Again too, the carry bit (C flag) will be affected by the results of the operation.

SUBTRACT THE CONTENTS OF A CPU REGISTER FROM THE ACCUMULATOR

SUA	220
SUB	221
SUC	222
SUD	223
SUE	224
SUH	225
SUL	226

This group of instructions will cause the present value of the specified CPU register to be subtracted from the value in the accumulator. The value of the specified register is not changed except in the case of the SUA instruction. (Note that the SUA instruction is a convenient instruction with which to "clear" the accumulator.) The carry flag will be affected by the results of a SUBTRACT instruction.

SUBTRACT THE CONTENTS OF A CPU REGISTER AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBA	230
SBB	231
SBC	232
SBD	233
SBE	234
SBH	235
SBL	236

This group is identical to the previous group except that the content of the carry flag is considered as an additional bit (MSB) in the specified CPU register. The combined value of the carry bit plus the contents of the specified CPU register are SUBTRACTED from the value in the accumulator. The results are left in the accumulator. The carry bit (C flag) is affected by the result of the operation. With the exception of the SBA instruction the content of the specified CPU register is left unchanged.

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A CPU REGISTER

CPA	270
CPB	271
CPC	272
CPD	273
CPE	274
CPH	275
CPL	276

The COMPARE group of instructions are a very powerful and somewhat unique set of instructions. They direct the computer to compare the contents of the accumulator against another register and to set the flags as a result of the comparing operation. It is essentially a subtraction operation with the value of the specified register being subtracted from the value of the accumulator except that the value of the accumulator is not actually altered by the operation. However, the flags are set in the same manner as though an actual subtraction operation had occured. Thus, by subsequently testing the status of the various flags after a COMPARE instruction has been executed, the program can determine whether the compare operation resulted in a match or non-match. In the case of a non-match, one may determine if the compared register contained a value greater or less than that in the accumulator. This would be accomplished by testing the Z flag and C flag respectively utilizing a conditional JUMP or CALL instruction (which will be described later).

ADDITION, SUBTRACTION, AND COMPARE INSTRUCTIONS THAT USE WORDS IN MEMORY AS OPERANDS

The five types of mathematical operations: ADD, ADD with CARRY, SUBTRACT, SUBTRACT with CARRY, and COMPARE, which have just been presented for the cases where they operate with the contents of CPU registers, can all be performed with words that are in memory. As with the LOAD instructions that operate with memory, the H and L registers must contain the address of the word in memory that it is desired to ADD, SUBTRACT, or COMPARE to the accumulator. The same conditions for the operations as was detailed when using the CPU registers apply. Thus, for mathematical operations with a word in memory, the following instructions are used.

ADD THE CONTENTS OF A MEMORY WORD TO THE ACCUMULATOR

ADM	207

ADD THE CONTENTS OF A MEMORY WORD PLUS THE VALUE OF THE CARRY FLAG TO THE ACCUMULATOR

ACM	217

SUBTRACT THE CONTENTS OF A MEMORY WORD FROM THE ACCUMULATOR

SUM	227

SUBTRACT THE CONTENTS OF A MEMORY WORD AND THE VALUE OF THE CARRY FLAG FROM THE ACCUMULATOR

SBM	237

COMPARE THE VALUE IN THE ACCUMULATOR AGAINST THE CONTENTS OF A MEMORY WORD

CPM	277

IMMEDIATE TYPE ADDITIONS, SUBTRACTIONS, AND COMPARE INSTRUCTIONS

The five types of mathematical operations discussed above can also be performed with the operand being the word of data immediately after the instruction. This group of instructions is similar in format to the previously described LOAD IMMEDIATE instructions. The same conditions for the mathematical operations as discussed for the operations with the CPU registers apply.

ADD IMMEDIATE

ADI	004

ADD WITH CARRY IMMEDIATE

ACI	014

SUBTRACT IMMEDIATE

SUI	024

SUBTRACT WITH CARRY IMMEDIATE

SBI	034

COMPARE IMMEDIATE

CPI	074

LOGICAL INSTRUCTIONS WITH THE ACCUMULATOR

There are several groups of instructions which allow BOOLEAN LOGIC operations to be performed between the contents of the CPU registers and the A (accumulator) register. In addition there are logic IMMEDIATE type instructions. The boolean logic operations are valuable in a number of programming applications. The instruction set allows three basic boolean operations to be performed. These are: the LOGICAL AND, the LOGICAL OR, and the EXCLUSIVE OR operations. Each type of logic operation is performed on a bit-by-bit basis between the accumulator and the CPU register or memory location specified by the instruction. A detailed explanation of each type of logic operation, and the appropriate instructions for each type is presented below. The logic instruction set is also valuable because all of them will cause the C (carry) flag to be placed in the zero condition. This is important if one is going to perform a sequence of instructions that will eventually use the status of the C flag to arrive at a decision as it allows the programmer to set the C flag to a known state at the start of the sequence. All other flags are set in accordance with the result of the logic operation. Hence, the group often has value when the programmer desires to determine the contents of a register that has just been loaded into a register. (Since the load instructions do not alter the flags.)

THE BOOLEAN 'AND' OPERATION INSTRUCTION SET

When the boolean AND instruction is executed, each bit of the accumulator will be compared with the corresponding bit in the register or memory location specified by the instruction. As each bit is compared a logic result will be placed in the accumulator for each bit comparison. The logic result is determined as follows. If both the bit in the accumulator and the bit in the register with which the operation is being performed are a logic one, then the accumulator bit will be left in the logic one condition. For all other possible combinations (A bit equals one, X bit equals zero; A bit equals zero, X bit equals one; or A bit equals zero, X bit equals zero), then the accumulator bit will be cleared to the zero state. An example will illustrate the logical AND operation.

INITIAL STATE OF THE ACCUMULATOR

10 101 010

CONTENTS OF OPERAND REGISTER

11 001 100

FINAL STATE OF THE ACCUMULATOR

10 001 000

There are seven logical AND instructions that allow any CPU register to be used as the AND operand. They are as follows.

NDA	240
NDB	241
NDC	242
NDD	243
NDE	244
NDH	245
NDL	246

The contents of the operand register is not altered by an AND logical instruction.

There is also a logical AND instruction that allows a word in memory to be used as an operand. The address of the word in memory that will be used is pointed to by the contents of the H and L CPU registers.

NDM 247

And finally there is also a logical AND IMMEDIATE type of instruction that will use the contents of the word immediately following the instruction as the operand.

NDI 044

The next group of boolean logic instructions direct the computer to perform the logical OR operation on a bit-by-bit basis with the accumulator and the contents of a CPU register or a word in memory. The logical OR operation will result in the accumulator having a bit set to a logic one if either that bit in the accumulator, or the corresponding bit in the operand register is a logic one. Since the case where both the accumulator bit and operand bit are a one also satisfies the criteria, that condition will also result in the accumulator bit being left in the one state. If neither register has a logic one in the bit position, then the accumulator bit for that position remains in the zero state. An example illustrates the results of a logical OR operation.

INITIAL STATE OF THE ACCUMULATOR

1 0 1 0 1 0 1 0

CONTENT OF THE OPERAND REGISTER

1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR

1 1 1 0 1 1 1 0

There are seven logical OR instructions that allow any CPU register to be used as the OR operand.

ORA	260
ORB	261
ORC	262
ORD	263
ORE	264
ORH	265
ORL	266

By using the H and L registers as pointers one can also use a word in memory as an OR operand.

ORM	267

There is also the logical OR IMMEDIATE instruction.

ORI	064

As with the logical AND group of instructions, the logical OR instruction does not alter the contents of the operand register.

The last group of boolean logic instructions is a variation of the logic OR. The variation is termed the EXCLUSIVE OR logical operation. The EXCLUSIVE OR operation is similar to the OR except that when the corresponding bits in both the accumulator and the operand register are a one then the accumulator bit will be cleared to zero. Thus, the accumulator bit will be a one after the operation only if just one of the registers (accumulator register or operand register) has a one in the bit position. (Again, the operation is performed on a bit-by-bit basis.) An example provides clarification.

INITIAL STATE OF THE ACCUMULATOR

1 0 1 0 1 0 1 0

CONTENTS OF THE OPERAND REGISTER

1 1 0 0 1 1 0 0

FINAL STATE OF THE ACCUMULATOR

0 1 1 0 0 1 1 0

The seven instructions that allow the CPU registers to be used as operands are shown next.

XRA	250
XRB	251
XRC	252
XRD	253
XRE	254
XRH	255
XRL	256

The instruction that uses registers H and L as pointers to a memory location is:

XRM	257

And the EXCLUSIVE OR IMMEDIATE type instruction is:

XRI	054

As in the case of the logical OR operation, the operand register is not altered except for the special case when the XRA instruction is used. This instruction, which directs the computer to EXCLUSIVE OR the accumulator with itself, will cause the operand register, since it is the accumulator, to have its contents altered (unless it should happen to be zero at the time the instruction is executed). This is because, regardless of what value is in the accumulator, if it is EXCLUSIVE OR'ed with itself, the result will be zero! The example below illustrates the specific operation.

ORIGINAL VALUE OF ACCUMULATOR

10 101 010

EXCLUSIVE OR'ed WITH ITSELF

10 101 010

FINAL VALUE OF ACCUMULATOR

00 000 000

This only occurs when the logical EXCLUSIVE OR is performed on the accumulator itself. It can be shown that the results of performing the logical OR or logical AND between the accumulator and itself will result in the original accumulator value being retained.

INSTRUCTIONS FOR ROTATING THE CONTENTS OF THE ACCUMULATOR

It is often desirable to be able to shift the contents of the accumulator either right or left. In a fixed length register, a simple shift operation would result in some information being lost because what was in the MSB or LSB (depending on in which direction the shift occured) would be shifted right out of the register! Therefore, instead of just shifting the contents of a register, an operation termed ROTATING is utilized. Now, instead of just shifting a bit off the end of the register, the bit is brought around to the other end of the register. For instance, if the register is rotated to the right, the LSB (least significant bit) would be brought around to the position of the MSB (most significant bit) which would have been vacated by the shifting of its original contents to the right. Or, in the case of a shift to the left, the MSB would be brought around to the position of the LSB.

The carry bit (C flag) can be considered as an extension of the accumulator register. The instruction set for this machine allows two types of ROTATE instructions. One considers the carry bit to be part of the accumulator register for the rotate operation. The other type does not. In addition, each type of rotate can be done either to the right or to the left.

It should be noted that the rotate operations are particularly valuable when it is desired to multiply a number or divide a number. This is because shifting the contents of a register to the left effectively multiplies a binary number by a power of two. Shifting a binary number to the right provides the inverse operation.

ROTATING THE ACCUMULATOR LEFT

RLC 002

Rotating the accumulator left with the RLC instruction means the MSB of the accumulator will be brought around to the LSB position and all other bits will be shifted one position to the left. While this instruction does not shift through the carry bit, the carry bit will be set by the status of the MSB of the accumulator at the start of the ROTATE LEFT operation. (This feature allows the programmer to determine what the MSB was prior to the shifting operation by testing the C flag after the rotate instruction has been executed.

ROTATING THE ACCUMULATOR LEFT THROUGH THE CARRY BIT

RAL 022

The RAL instruction will cause the MSB of the accumulator to go into the carry bit. The initial value of the carry bit will be shifted around to the LSB of the accumulator. All other bits are shifted one position to the left.

ROTATING THE ACCUMULATOR RIGHT

RRC 012

The RRC instruction is similar to the RLC instruction except that now the LSB of the accumulator is placed in the MSB of the accumulator. All other bits are shifted one position to the right. Also, the carry bit will be set to the initial value of the LSB of the accumulator at the start of the operation.

ROTATING THE ACCUMULATOR RIGHT THROUGH THE CARRY BIT

RAR 032

Here, the LSB of the accumulator is brought around to the carry bit. The initial value of the carry bit is shifted to the MSB of the accumulator. All other bits are shifted a position to the right.

It should be noted that the C flag is the only flag that is altered by a rotate instruction. All other flags remain unchanged.

JUMP INSTRUCTIONS

The instructions discussed so far have all been "direct action" instructions. The programmer arranges a sequence of these types of instructions in memory. When the program is started the computer proceeds to execute the instructions in the order in which they are encountered. The computer automatically reads the contents of a memory location, executes the instruction it finds there, and then automatically increments a special address register called a PROGRAM COUNTER that will result in the machine reading the information contained in the next sequential memory location. However, it is often desirable to perform a series of instructions located in one section of memory, and then skip over a group of memory locations and start executing instructions in another section of memory. This action can be accomplished by a group of instructions that will cause a new address value to be placed in the PROGRAM COUNTER. This will cause the computer to go to a new section of memory and then execute instructions sequentially from the new memory location.

The JUMP instructions in this computer add considerable power to the machine's capabilities because there are a series of "conditional" JUMP instructions available. That is, the computer can be directed to test the status of a particular FLAG (C, Z, S or P). If the status of the flag is the desired one, then a JUMP will be performed. If it is not, the machine will continue to execute the next instruction in the current sequence. This capability provides a means for the computer to make "decisions" and to modify its operation as a function of the status of the various flags at the time that a program is being executed.

In a manner similar to IMMEDIATE types of instructions, the JUMP instructions require more than one word of memory. A JUMP instruction requires three words to be properly defined. (Remember that IMMEDIATE type instructions required two words.) The JUMP instruction itself is the first word. The second word must contain the LOW ADDRESS portion of the address of the word in memory that the PROGRAM COUNTER is to be set to point to, which is the new location from which the next instruction is to be fetched. The third word must contain the

HIGH ADDRESS (sometimes referred to as the PAGE) of the memory address that the program counter will be set to. That is, the high order portion of the address in memory that the computer will JUMP to in order to obtain its next instruction.

THE UNCONDITIONAL JUMP INSTRUCTION

JMP	1X4

Note: The machine code 1X4 indicates that any code for the second octal digit of the machine code is valid. It is recommended as a standard practice that the code '0' be used. Thus, the typical machine code would be 104.

Remember, the JUMP instruction must be followed by two more words which contain the LOW, and then the HIGH (PAGE) portion of the address that the program is to JUMP to!

JUMP IF THE DESIGNATED FLAG IS TRUE (CONDITIONAL JUMP)

JTC	140
JTZ	150
JTS	160
JTP	170

As with the UNCONDITIONAL JUMP instruction, the CONDITIONAL JUMP instructions must be followed by two words of information. The LOW portion, then the HIGH portion, of the address that program execution is to continue from if the jump is executed. The JUMP IF TRUE group of instructions will only jump to the designated address if the condition of the appropriate flag is TRUE (logical one). Thus, the JTC instruction states that if the carry flag (C) is a logical one (TRUE) then the jump is to be executed. If it is a logical zero (FALSE) then program execution is to continue with the next instruction in the current sequence of instructions. In a similar manner the JTZ instruction states that if the ZERO FLAG is TRUE then the jump is to be performed. Otherwise the next instruction in the present sequence is executed. Likewise for the JTS and JTP instructions.

JUMP IF THE DESIGNATED FLAG IS FALSE (CONDITIONAL JUMP)

JFC	100
JFZ	110
JFS	120
JFP	130

As with all JUMP instructions these instructions must be followed by the LOW address then the HIGH address of the memory location that program execution is to continue from if the jump is executed. This group of instructions is the opposite of the jump if the flag is true group. For instance, the JFC instruction commands the computer to test the status of the carry (C) flag. If the flag is FALSE (a logic zero), then the jump is to be performed. If it is TRUE, then program execution is to continue with the next instruction in the current sequence of instructions. The same procedure holds for the JFZ, JFS and JFP instructions.

SUBROUTINE CALLING INSTRUCTIONS

Quite often when a programmer is developing computer programs the programmer will find that a particular algorithm (sequence of instructions for performing a function) can be used many times in different parts of the program. Rather than having to keep entering the same sequence of instructions at different locations in memory, which would not only consume the time of the programmer, but would also result in a lot of memory being used to perform one particular function, it is desirable to be able to be able to put an often used sequence of commands in just one location in memory. Then, whenever the particular algorithm is required by another part of the program, it would be convenient to jump to the section that contained the often used algorithm, perform the sequence of instructions, and then return back to the main part of the program. This is a standard practice in computer operations. A frequently used algorithm can be designated a SUBROUTINE. A special set of instructions allows the programmer to CALL a SUBROUTINE. In other words, specify a special type of JUMP command that will eventually allow the program to RETURN to the original "jumping" point in the program. A second type of instruction is used to terminate a SUBROUTINE. This special terminator will cause the program to revert back and pick up the next sequential instruction in memory that immediately follows the original CALLING instruction. A great deal of computer power is provided by the instruction set in this machine that allows one to CALL and RETURN from SUBROUTINES. This is because, in a manner similar to that provided for the CONDITIONAL JUMP instructions, there are a number of CONDITIONAL CALL and CONDITIONAL RETURN commands in the instruction set.

Like the JUMP instructions, the CALL instructions all require three words in order to be fully specified. The first word is the CALL instruction itself. The next two words must contain the LOW and HIGH portions of the starting address of the subroutine that is being "called."

When a CALL instruction is encountered by the computer, the CPU will actually save the current value of the PROGRAM COUNTER by storing it in a special PROGRAM COUNTER PUSH-DOWN STACK. This stack is capable of holding six addresses plus the current operating address. What this means is that the machine is capable of "nesting" up to seven subroutines at one time. Thus, one can have a subroutine, that in turn calls another subroutine, that in turn calls another one, up to seven levels, and the machine will still be able to return to the initial calling location. The programmer must ensure that subroutines are not nested more than seven levels otherwise the PROGRAM COUNTER PUSH-DOWN STACK will push the original calling address(es) completely out of the push-down stack. The program could then no longer automatically return to the initial calling location.

The RETURN instruction which terminates a SUBROUTINE only requires one word. When the CPU encounters a RETURN instruction it causes the PROGRAM COUNTER PUSH-DOWN STACK to "pop" up one level. This effectively causes the address saved in the stack by the calling routine to be taken as the new program counter. Hence, program execution returns to the calling location.

THE UNCONDITIONAL CALL INSTRUCTION

CAL 1X6

This instruction followed by two words containing the LOW and then the HIGH order of the starting address of the SUBROUTINE that is to be executed is an UNCONDITIONAL CALL. The subroutine will be executed regardless of the status of the FLAGS. The next sequential address after the CAL instruction is saved in the PROGRAM COUNTER PUSH-DOWN STACK.

THE UNCONDITIONAL RETURN INSTRUCTION

RET 0X7

This instruction directs the CPU to unconditionally "pop" the program counter push-down stack UP one level. Program execution will continue from the address saved by the subroutine calling instruction.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

CTC	142
CTZ	152
CTS	162
CTP	172

In a manner similar to the conditional JUMP IF TRUE instructions, these instructions (which must all be followed by the LOW and HIGH portions of the called subroutine's starting address) will only perform the "call" if the designated flag is in the TRUE (logical one) state. If the designated flag is FALSE then the CALL instruction is ignored. Program execution then continues with the next sequential instruction.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS TRUE

RTC	043
RTZ	053
RTS	063
RTP	073

These one word instructions will cause a SUBROUTINE to be TERMINATED only if the designated flag is in the logical one (TRUE) state.

CALL A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

CFC	102
CFZ	112
CFS	122
CFP	132

These instructions are the opposit of the previous group of calling commands. The subroutine is called only if the designated flag is in the FALSE (logical zero) condition. Remember, these instructions must be followed by two words which contain the LOW and HIGH part of the starting address of the SUBROUTINE that is to be executed if the designated flag is FALSE. If the flag is TRUE, the subroutine will not be called and program operation will continue with the next instruction in the current sequence.

RETURN FROM A SUBROUTINE IF THE DESIGNATED FLAG IS FALSE

RFC	003
RFZ	013
RFS	023
RFP	033

These one word instructions will terminate a subroutine ("pop" the program counter stack UP one level) if the designated flag is FALSE. Otherwise, the instruction is ignored and program operation is continued with the next instruction in the subroutine.

THE SPECIAL RESTART SUBROUTINE CALL INSTRUCTIONS

There is a special purpose instruction available that effectively serves as a one word SUBROUTINE CALL. (Remember that it normally requires three words to specify a subroutine call.) This special instruction allows the programmer to call a subroutine that starts at any one of eight specially designated memory locations. The eight special memory locations are at locations: 000, 010, 020, 030, 040, 050, 060 and 070 on page zero. There are eight variations of the machine code for the RESTART instruction. One for each of the above addresses. Thus, the one word instruction can serve to CALL a SUBROUTINE at the specified starting location (instead of having two additional words to specify the starting address of a subroutine). It is often convenient to utilize a

RESTART command as a quick CALL to an often used subroutine. Or, as an easy way to call short "starting" subroutines for large programs. Hence, the name for the type of instruction. The eight RESTART instructions, in their mnemonic and machine code forms, along with the starting address associated with each one is listed below.

RST 0	005	00 000
RST 1	015	00 010
RST 2	025	00 020
RST 3	035	00 030
RST 4	045	00 040
RST 5	055	00 050
RST 6	065	00 060
RST 7	075	00 070

INPUT INSTRUCTIONS

In order to receive information from an external device the computer must utilize a group of special signal lines. The typical '8008' computer is designed to handle up to eight groups (each group having eight signal lines) of INPUT signals. A group of signals is accepted at the computer by what is referred to as an INPUT PORT. The computer controls the operation of the INPUT PORTS. Under program control, the computer can be directed to obtain the information that is on a group of lines coming in to any INPUT PORT. When this is done the information will be transferred to the accumulator. Various types of external equipment, such as an electronic keyboard or measuring instruments, can be connected to the INPUT PORTS. The INPUT PORTS are typically referred to as having numbers from '0' to '7.' The typical mnemonics and machine codes for INPUT instructions are shown next.

INP 0	101
INP 1	103
.	.
INP 6	115
INP 7	117

It may be interesting to note that the machine codes for input ports increase by a factor of two for each port. Note too, that while the mnemonic for an input instruction has two parts, the machine code only requires one word in memory. It is also important to realize that while an input instruction brings data into the accumulator it does not affect the status of any of the CPU flags!

OUTPUT INSTRUCTIONS

In order to output information to an external device the computer utilizes another group of signal lines which are referred to as OUTPUT PORTS. A Typical '8008' system may be equipped to service up to twenty-four OUTPUT PORTS. (Each OUTPUT PORT actually consists of eight signal lines.) An OUTPUT instruction causes the contents of the accumulator to be transferred to the signal lines of the designated OUTPUT PORT. The output ports are normally designated by octal numbers in the range 10 to 37. The list below shows the typical mnemonics used to specify an OUTPUT PORT along with the associated machine code. (It may be interesting to note again that the machine code increases by a factor of two for each port.)

OUT 10	121
OUT 11	123
.	.
OUT 21	141
.	.
OUT 36	175
OUT 37	177

An OUTPUT instruction only requires one machine code word (even though the mnemonic is typically specified in two parts). OUTPUT PORTS are connected to external devices that one desires to have the computer transmit information to, such as a CRT display, or machinery that is to be placed under computer control.

THE HALT INSTRUCTION

There is one more instruction in the '8008' instruction set. This instruction directs the CPU to stop all operations and to remain in that state until an INTERRUPT signal is received. In a typical '8008' system an INTERRUPT signal may be generated by an operator pressing a switch or by an external piece of equipment sending an electronic signal to the CPU. This instruction is normally used when the programmer desires to terminate a program or when it is desired to have the computer wait for an operator or external device to perform some action. There are three machine codes that may be used for the HALT command.

HLT	000
HLT	001
HLT	377

The HALT instruction does not affect the status of the CPU flags.

INFORMATION ON INSTRUCTION EXECUTION TIMES

When programming for "real-time" applications it is important to know how much time each type of instruction requires to be executed. With this information the programmer can develop "timing loops" or determine with substantual accuracy how much time it will take to perform a particular series of instructions. This information is especially valuable when dealing with programs that control the operations of external devices which might require events to occur at specific times.

The following table provides the nominal instruction execution time for each category of instruction used in an '8008' system. The precise time needed for each instruction depends on how close the master clock has been set to a nominal value of 500 kilohertz. The table shows the number of cycle states required by the type of instruction followed by the nominal time required to perform the entire instruction. Since each state executes in four microseconds, the total time required to perform the instruction as shown in the table was obtained by multiplying the number of states by four microseconds. By knowing the number of states required for each instruction the programmer can often rearrange an algorithm or substitute different types of instructions to provide programs that have events occuring at precisely timed intervals.

INSTRUCTION EXECUTION TIME TABLE

LOAD DATA FROM A CPU REGISTER TO ANOTHER CPU REGISTER	5	20 Us
LOAD DATA FROM A CPU REGISTER TO A LOCATION IN MEMORY	7	28
LOAD DATA FROM MEMORY TO A CPU REGISTER	8	32
LOAD IMMEDIATE DATA INTO A CPU REGISTER	8	32
LOAD IMMEDIATE DATA INTO A LOCATION IN MEMORY	9	36

INSTRUCTION EXECUTION TIME TABLE (CONCLUDED)

INCREMENT OR DECREMENT A CPU REGISTER	5	20 Us.
ARITHMETIC/COMPARE BETWEEN ACCUMULATOR & A CPU REGISTER	5	20
ARITH/COMPARE BETWEEN ACCUMULATOR & A WORD IN MEMORY	8	32
IMMEDIATE ARITHMETIC AND COMPARE	8	32
BOOLEAN OPS BETWEEN ACCUMULATOR AND CPU REGISTERS	5	20
BOOLEAN OPS WITH ACCUMULATOR & A WORD IN MEMORY	8	32
IMMEDIATE BOOLEAN OPERATIONS	8	20
ROTATE THE ACCUMULATOR	5	20
JUMP AND CALL COMMANDS (UNCONDITIONAL)	11	44
JUMP/CALLS WHEN CONDITION NOT SATISFIED (CONDITIONAL)	9	36
JUMP/CALLS WHEN CONDITION SATISFIED (CONDITIONAL)	11	44
RETURN (UNCONDITIONAL)	5	20
RETURN WHEN CONDITION NOT SATISFIED (CONDITIONAL)	3	12
RETURN WHEN CONDITION SATISFIED (CONDITIONAL)	5	20
RESTART COMMAND	5	20
OUTPUT COMMAND	6	24
INPUT COMMAND	8	32
HALT COMMAND	4	16

INITIAL STEPS FOR DEVELOPING PROGRAMS

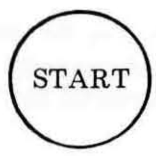
The first task that should be done prior to starting to write the individual instructions for a computer program is to decide exactly what it is that the computer is to perform and to write the goal(s) down on paper! This statement might seem unnecessary to some because it is such an obvious one. It is stated because the majority of people learning to develop programs will realize its significance when they discover, halfway through the writing of a large machine language program, that they left out a vital step. Such an error can typically result in the programmer having to start back at the beginning and rewrite the entire program. The practice of writing down just what tasks a particular program is to perform and the steps in which they are to be done, will save a lot of work in the long run. The written description should be as complete and detailed as necessary to ensure that exactly each step of the program will be clear when actually writing the program in machine language. It is generally wise for the novice programmer to take pains to be quite detailed in the initial description.

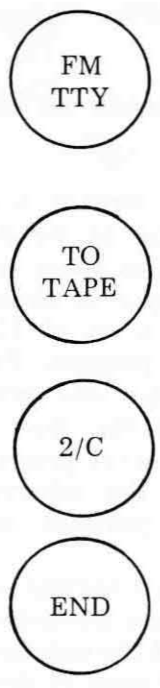
The act of actually writing down the proposed operation of the program desired serves several valuable purposes. First, it forces one to carefully review what is planned. In doing so, it often vividly reveals flaws in original mental ideas. Secondly, it serves as a guide and a check list as the machine language program is developed. Remember, it will often take a number of hours to write a fair sized program. These hours might be spread over several days or weeks. In this period of time the human mind can easily forget original intentions and plans if the human memory is not refreshed by written notes. A program that is not kept carefully organized as it is developed can become a real mess. This is especially so if one keeps forgetting key concepts or has to constantly add in forgotten routines. The time wasted by such sloppy procedures can be avoided if proper work habits are developed from the beginning.

Once one has written a description of the general task(s) to be performed, and has ascertained that there are no flaws to the overall concepts or ideas, it is a good idea to draw up a set of FLOW CHARTS for the proposed program. FLOW CHARTS are detailed written and symbolic descriptive diagrams of the flow of operations that are to occur as the program is executed. They also show the interrelationships between different portions of a program.

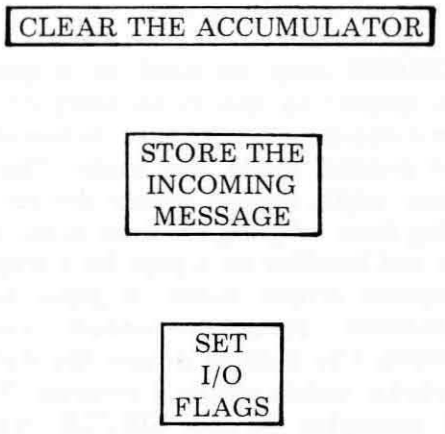
Over the years a variety of symbols and methods have been developed for creating flow charts. All of the varieties have the same basic purpose and most of the differences are the result of individuals pushing their own preferences. Most people can do admirably well using just a few basic symbols to denote fundamental types of operations in a computer program. The small group to be presented here will enable most microcomputer programmers to develop flow charts rapidly, with little confusion, and without having to learn a host of special symbols.

A CIRCLE may be used as a general purpose symbol to specify an entry or exit point in a routine or subroutine. Information may be printed inside the circle. This information might denote where the routine is coming from or going to (such as the page number and location on a page for a program that requires several sheets of paper to be flow charted). It might contain transfer information. Or, it could denote the starting and stopping points within a program. Some typical examples of the CIRCLE symbol are illustrated next.

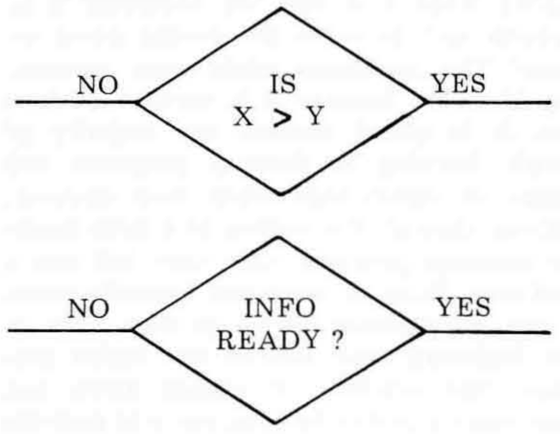




A square or rectangel may be used to denote a general or specific operation. The type of operation may be described inside the box such as illustrated in the following examples.



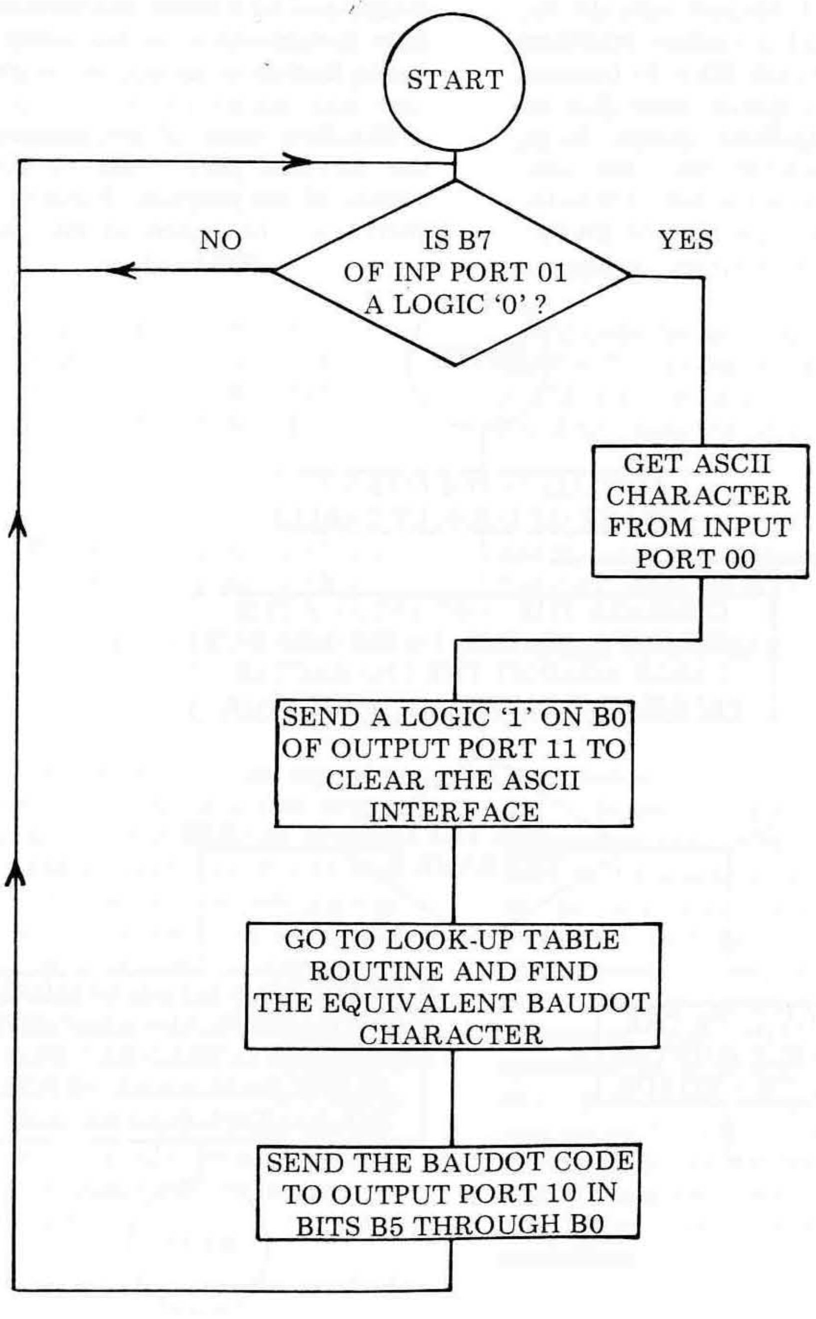
A diamond form may be used to symbolize a decision or branching point in a program. The determining factor(s) for the decision or branching operation may be indicated inside the symbol. The two side points of the diamond are used to illustrate the path taken when a decision has been made. The diamond symbol is illustrated next.



Lines with arrows may be used to interconnect the three types of symbols presented. In this way, the symbols may be connected to form readily understood FLOW CHARTS of operations that are to occur in a program and to show how various operations relate to each other. Flow charts are extremely valuable references when developing programs as well as when one wants to update or expand a program and needs to quickly review the operation of the program of specific interest.

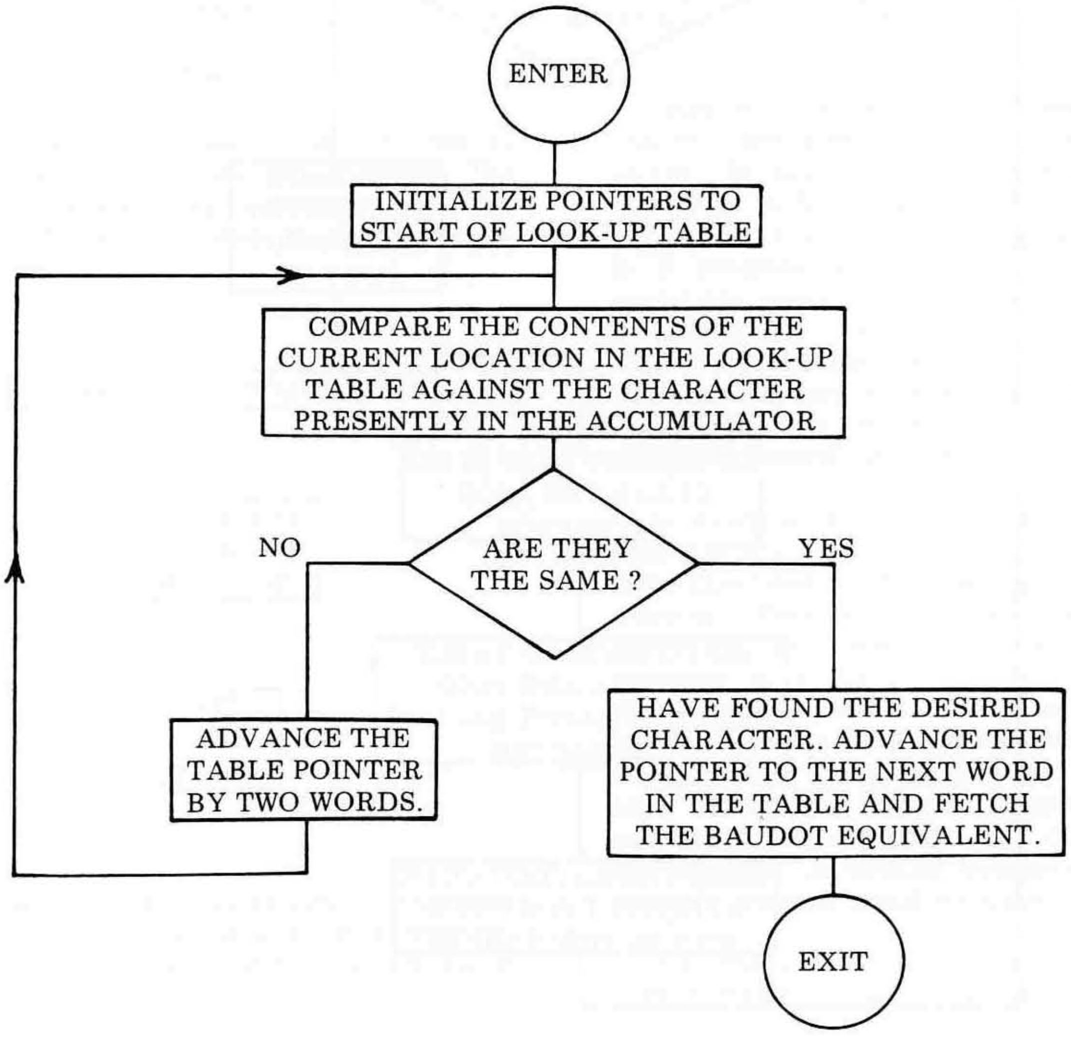
An example of a flow chart for a relatively simple program will be shown next. The program illustrated by the flow chart is to accept characters from an ASCII encoded electric typewriter and send out the equivalent character to a BAUDOT coded device. In this illustration it is assumed that the I/O interfaces to the machines are parallel interfaces (versus the possibility of being bit-serial interfaces). Thus, complex timing operations do not have to be discussed in the example. A written description of the example program could be stated as follows.

The computer is to monitor bit B7 of INPUT PORT 01, which is the control port for an interface to an ASCII encoded electric typewriter. Whenever bit B7 on INPUT PORT 01 goes low (logic '0') it indicates a new character is waiting in parallel format from the typewriter at INPUT PORT 00. The computer is to immediately obtain the character that is waiting at INPUT PORT 00 and as soon as it has obtained the data it is to send a logic '1' (high) signal to bit B0 of OUTPUT PORT 11 to signal the ASCII interface that the character has been accepted by the computer. (The receipt of this signal by the ASCII interface will then cause the ASCII interface to restore the control signal on bit B7 of INPUT PORT 01 to a high (logic '1') condition.)


START
NO
IS B7 OF INP PORT 01 A LOGIC '0' ?
YES
GET ASCII CHARACTER FROM INPUT PORT 00
SEND A LOGIC '1' ON B0 OF OUTPUT PORT 11 TO CLEAR THE ASCII INTERFACE
GO TO LOOK-UP TABLE ROUTINE AND FIND THE EQUIVALENT BAUDOT CHARACTER
SEND THE BAUDOT CODE TO OUTPUT PORT 10 IN BITS B5 THROUGH B0

Whenever a character has been received from the ASCII typewriter on INPUT PORT 00, the computer is to compare the character just received against an ASCII to BAUDOT look-up table which is stored in the computer's memory until it finds a match. It will then obtain the equivalent BAUDOT character from the conversion table. It will then send the BAUDOT code for the character in bit positions B5 through B0 of OUTPUT PORT 10. Bit B5 will serve to indicate to the BAUDOT interface whether the code in bits B4 through B0 is to be processed by the BAUDOT device when it is in the LETTERS or FIGURES mode. It is assumed that the character rate (but not necessarily the baud rate) is the same for both machines so that the example may be simplified by eliminating the requirement for character buffering or stacking in the memory of the computer. However, in practical applications such capability might be required. The feature could be added to the program. However, for this case, as soon as the BAUDOT code has been transmitted (in parallel format) to the BAUDOT device, the computer will simply go back to waiting for the next character to come in from the ASCII machine. The written description of the program just presented is succinctly summarized in the flow chart shown on the previous page!

The flow chart of the program shown on the previous page could be considered an outline of the program. Portions of that flow chart could be expanded into more detailed


ENTER
INITIALIZE POINTERS TO START OF LOOK-UP TABLE
COMPARE THE CONTENTS OF THE CURRENT LOCATION IN THE LOOK-UP TABLE AGAINST THE CHARACTER PRESENTLY IN THE ACCUMULATOR
ARE THEY THE SAME ?
NO
YES
ADVANCE THE TABLE POINTER BY TWO WORDS.
HAVE FOUND THE DESIRED CHARACTER. ADVANCE THE POINTER TO THE NEXT WORD IN THE TABLE AND FETCH THE BAUDOT EQUIVALENT.
EXIT

flow charts to present a detailed view of special operations. For instance, the rectangle labeled GO TO LOOK-UP TABLE ROUTINE AND FIND THE EQUIVALENT BAUDOT CHARACTER really refers to a portion of the program that consists of a number of operations. Those operations could be described in a separate flow chart such as the one just presented.

The reader can see that the expanded flow chart illustrates the operation of the table look-up routine portion of the program. With a little study one can discern that the look-up table consist of an area in memory that has an ASCII encoded character in one word, followed in the next word by the same character in BAUDOT code. This sequence continues for all the possible characters as illustrated below. The flow chart illustrates how the data in the look-up table is scanned by skipping over every other memory location (which contains the BAUDOT codes) until the proper ASCII character is located. When that is located, the routine simply extracts the proper BAUDOT code from the next memory locaction in the table. The flow chart makes the sequence easier to understand than a purely verbal explanation of the routine.

ADDRESS		MEMORY CONTENTS
PAGE: XX	LOC: Z	ASCII code for letter A
PAGE: XX	LOC: Z+1	BAUDOT code for letter A
PAGE: XX	LOC: Z+2	ASCII code for letter B
PAGE: XX	LOC: Z+3	BAUDOT code for letter B
.	.	.
.	.	.
.	.	.
PAGE: XX	LOC: Z+2(N-1)	ASCII code for N'th letter
PAGE: XX	LOC: Z+2(N-1)+1	BAUDOT code for N'th letter

ILLUSTRATION OF LOOK-UP TABLE ORGANIZATION FOR THE EXAMPLE PROGRAM

It is strongly recommended that beginning programmers develop the habit of first writing down the function(s) of the desired program they intend to create. Next, one should draw up flow charts as detailed as one feels is necessary to clearly show the operation of the program that is to be developed. A novice programmer will be wise to prepare quite detailed flow charts. More experienced programmers may prefer to leave out details of operations that they thoroughly understand. Flow charts should serve as ready references when the programmer goes on to actually develop the step-by-step machine language instruction sequences for the computer.

Flow charts are also an excellent method for communicating programming concepts to fellow computer technologists. Remember that general flow charts do not have to be machine specific!) Learning how to prepare and read flow charts is an important (yet easy) skill for all computer programmers to acquire. It can also be fun and a highly creative process. Using the technique, one may review the overall operation of a program under development and gain new insights into where to interconnect routines, where common loops exist (which can save valuable memory room if they are subroutined), and find other ways in which to enhance a program's capabilities.

Before one can effectively develop machine language programs for a computer, one must be thoroughly familiar with the instruction set for the machine. It is assumed for the remainder of this manual that the reader has studied the detailed information for the instruction set of the 8008 CPU which was provided in the first chapter. The programmer should become intimately familiar with the mnemonics (pronounced kneemonics) for each type of instruction. Mnemonics are easily remembered symbolic representations of machine language instructions. They are far easier to work with than the actual numeric codes used by the computer when the programmer is developing a program. While the programmer will develop programs and think in terms of the mnemonics, the programmer must eventually convert the mnemonics to the machine codes used by the computer. This, however, is almost purely a look-up procedure. In fact, as will be seen shortly, this task can actually be performed by the computer through the use of an ASSEMBLER program.

Machine language programmers should also be familiar with manipulating numbers in binary and octal form. It is assumed that readers are familiar with representing numbers as binary values. However, there may be a few readers who are not used to the convention of representing binary numbers by their octal equivalents. The technique is quite simple. It consists merely of grouping binary digits into groups of three and representing their value as an octal number. The octal numbering system only uses the digits 0 through 7. This is exactly the range that a group of three binary digits can represent. The octal numbering system makes it a lot easier to manipulate binary numbers. For instance, most people find it considerably more convenient to remember a three digit octal number such as 104 than the binary equivalent 01000100. An octal number is easily expanded to a binary number by simply placing the octal value in binary form using three binary digits.

The information in an eight bit binary register can be readily converted to an octal number by grouping the bits into groups of three starting with the least significant bits. The two most significant bits in the register which form the last group will only be able to represent the octal numbers 0 to 3. The diagram below illustrates the convention.

EIGHT CELL REGISTER

```
                                  .
                                .. ..
             ..................       ..................
             .                                         .
  ...........*******************************************
  .          *     *    †     *     *    †     *     *     *
  .    0     *  0  *  1 †  0  *  0  *  0 †  1  *  0  *  0  *
  .          *     *    †     *     *    †     *     *     *
  ...........*******************************************
  .                     .                   .              .
  ............   ...............   ................  ........
            .. ..               .. ..               .. ..
              ..                  ..                  ..
              1                   0                   4
```

CONVERTING AN 8 BIT REGISTER FROM BINARY TO OCTAL NUMBERS

Note in the diagram how an imaginary additional binary digit with a value of zero was assigned to the left of the most significant bit so that the octal convention for the two most significant bits could be maintained.

A table illustrating the relationship between the binary and octal systems is provided for reference below.

BINARY PATTERN	REPRESENTATIVE OCTAL NO.
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

A person who desires to develop machine language programs for computers should become familiar with standard conventions used when dealing with closed registers (groups of binary cells of fixed length such as a memory word or CPU register). One very simple point to remember is that when a group of cells in a register is in the all ones condition:

```
11 111 111
```

and a count of 1 is added to the register, the register goes to the value:

```
00 000 000
```

Or, if a count of: 10 (binary) was added to a register that contained all ones, the new value in the register would be as shown:

```
 11 111 111
+00 000 010
-----------
 00 000 001
```

Similarly, going the opposite way, if one subtracts a number such as 100 (binary) from a register that contains some lesser value, such as 010 (binary), the register would contain the result shown below:

```
 00 000 010
-00 000 100
-----------
 11 111 110
```

It may be noted that if one uses all the bits in a fixed length register, one may represent mathematical values with an absolute magnitude from zero to the quantity two to the Nth power, minus one (0 to (2**N - 1)) where N is the number of bits in the register. If all the bits in a register are used to represent the magnitude of a number, and it is also desired to represent the magnitude as being either positive or negative in sign, then some additional means must be available to record the sign of the magnitude. Generally, this would require using another register or memory location solely for the purpose of keeping track of the sign of a number.

In many applications it is desirable to establish a convention that will allow one to manipulate positive and negative numbers without having to use an additional register to maintain the sign of a number. One way this may be done is to simply assign the most significant bit in a register to be a sign indicator. The remaining bits represent the magnitude of the number regardless of whether it is positive or negative. When this is done, the magnitude range for an N cell register becomes 0 to (2**(N-1))-1 rather than 0 to (2**N) - 1. The convention normally used is that if the most significant bit in the register is a one then the number represented by the remaining bits is negative in sign. If the MSB is zero, then the remaining bits specify the magnitude of a positive number. This convention allows computer programmers to manipulate mathematical quantities in a fashion that makes it easy for the computer to keep track of the sign of a number. Some examples of binary numbers in an eight bit register are shown next.

BINARY REPRESENTATION	OCTAL	DECIMAL
00 001 000	010	+ 8
10 001 000	210	- 8
01 111 111	177	+127
11 111 111	377	- 127
00 000 001	001	+ 1
10 000 001	201	- 1

While the signed bit convention allows the sign of a number to be stored in the same register (or word) as the magnitude, simply using the signed bit convention alone can still be a somewhat clumsy method to use in a computer. This is because of the method in which a computer mathematically adds the contents of two binary registers in the accumulator. Suppose, for example, that a computer was to add together positive and negative numbers that were stored in registers in the signed bit format.

```
       00 001 000  (+ 8 decimal)
PLUS   10 001 000  (- 8 decimal)
       ------------------------
EQUAL  10 010 000  (This is not 0!)
```

The result of the operation illustrated would not be what the programmer intended! In order for the operation to be performed correctly, it is necessary to establish a method for processing the negative number called the two's complement convention. In the two's complement convention, a negative number is represented by complementing what the value for a positive number would be (complementing is the process of replacing bits that are '0' with a '1,' and those that are '1' with a 0) and then adding the value one (1) to the complemented value. As an example, the number minus eight (-8) decimal would be derived from the number plus eight (+8) by the following operations.

```
00 001 000     (Original + 8)

11 110 111     (Complemented)
00 000 001     (now add +1)
------------------------
11 111 000     (2's complement
                form of - 8)
```

Some examples of numbers expressed in two's complement notation with the signed bit convention are shown below.

BINARY REPRESENTATION	OCTAL	DECIMAL
00 001 000	010	+ 8
11 111 000	370	- 8
01 111 111	177	+127
10 000 001	201	- 127
00 000 001	001	+ 1
11 111 111	377	- 1
00 000 000	000	+ 0
10 000 000	200	- 128

Note that when using the two's complement method, one may still use the convention of having the MSB in the register establish the sign. If the MSB = 1, as in the above illustration, the number is assumed to be negative. Since the number is in the two's complement form, the computer can readily add a positive and a negative number and come up with a result that is readily interpreted. Look!

```
     00 001 000 (+ 8 decimal)
ADD  11 111 000 (- 8 dec as 2's comp)
     ------------------------
     00 000 000 (Correct answer = 0)
```

Another established convention in handling numbers with a computer is to assume that '0' is a positive value. Because of this convention, the magnitude of the largest negative number that can be represented in a fixed length register is one more than that possible for a positive number.

The various means of storing and manipulating the signs of numbers as just discussed have advantages and drawbacks, and the method used depends on the specific application. However, for most user's, the two's complement signed bit convention will be the most convenient, most often used, method. The prospective machine language programmer should make sure that the convention is well understood.

Another area that the machine language programmer must have a thorough knowledge of is the conversion of numbers between the decimal numbering system that most people work with on a daily basis, and the binary and octal numbering system utilized by computer technologists. Programmers working with microcomputers will generally find the octal numbering system most convenient. Because the conversion from octal to binary is simply a matter of grouping binary bits into groups of three as discussed at the start of this chapter, it is easier to remember octal codes than long strings of binary digits. However, most people are used to thinking in decimal terms, which the computer does not use at the machine language level. Thus, it is necessary for programmers to be able to convert back and forth between the various numbering systems as programs are developed.

The conversion process that is generally the most troublesome for people to learn is from decimal to binary, or decimal to octal (and vice-versa)! It is usually a bit easier for people to learn to convert from decimal to octal, and then use the simple octal to binary expansion technique, than to convert directly from decimal to binary. The easier method will be presented here. It is assumed that the reader is already familiar with going from octal to binary (and vice-versa). Only the conversions between decimal and octal (and the reverse) will be presented at this point.

A decimal number may be converted to its octal equivalent by the following technique:

Divide the decimal number by 8. Record the remainder (note that is the REMAINDER!!) as the least significant digit of the octal number being derived. Take the quotient just obtained and use it as the new dividend. Divide the new dividend by 8. The remainder from this operation becomes the next significant digit of the octal number. The quotient is again used as the new dividend. The process is continued until the quotient becomes '0.' The number obtained from placing all the remainders (from each division) in increasing significant order (first remainder as the least significant digit, last remainder as the most significant digit) is the octal number equivalent of the original decimal. The process is illustrated below for clarity.

The octal equivalent of 1234 decimal is:

```
ORIGINAL NUMBER            1234  /  8  =  154       2

LAST QUOTIENT BECOMES
NEW DIVIDEND                154  /  8  =   19     2 .

LAST QUOTIENT BECOMES
NEW DIVIDEND                 19  /  8  =    2   3 . .

LAST QUOTIENT BECOMES
NEW DIVIDEND                  2  /  8  =    - 2 . . .
                                              -------
Thus the octal equivalent of 1234 decimal is: 2 3 2 2
```

The above method is quite easy and straightforward. Since a majority of the time the user will be interested in conversions of decimal numbers less than 255 (the maximum decimal number that can be expressed in an eight bit register) only a few divisions are necessary:

The octal equivalent of 255 decimal is:

					QUOTIENT	REMAINDER
ORIGINAL NUMBER	255	/	8	=	31	7
LAST QUOTIENT BECOMES NEW DIVIDEND	31	/	8	=	3	7
LAST QUOTIENT BECOMES NEW DIVIDEND	3	/	8	=	-	3
Thus the octal equivalent of 255 is:						3 7 7

For numbers less than 63 decimal (and such numbers are used frequently to set counters in loop routines) the above method reduces to one division with the remainder being the LSD and the quotient the MSD. This is a feat most programmers have little difficulty doing in their head!

The octal equivalent of 63 decimal is:

ORIGINAL NUMBER	63	/	8	=	7	7
LAST QUOTIENT BECOMES NEW DIVIDEND	7	/	8	=	-	7
Thus the octal equivalent of 63 is:						7 7

Going from octal to decimal is quite easy too. The process consists of simply multiplying each octal digit by the number 8 raised to its positional (weighted) power, and then adding up the total of each product for all the octal digits:

2 3 2 2 Octal =

. 2	X	(8*0)	=	(2 X 1)	=	2
. . . 2	X	(8*1)	=	(2X8)	=	1 6
. . 3	X	(8*2)	=	(3 X 64)	=	1 9 2
2	X	(8*3)	=	(2 X 512)	=	1 0 2 4
Thus the decimal equivalend of 2322 Octal is :						1 2 3 4

Besides the basic mathematical skills involved with using octal and binary numbers, there are some practical bookkeeping considerations that machine language programmers must learn to deal with as they develop programs. These bookkeeping matters have to do with memory usage and allocation.

As the reader who has read chapter one in this manual knows, each type of instruction used in the 8008 CPU requires one, two, or three words of memory. As a general rule, simple register to register or register to memory commands require but one memory word. Immediate type commands require two memory locations (the instruction code followed immediately by the data or operand). Jump or call instructions require three words of memory storage. One word for the instruction code and two more words for the address of the location specified by the instruction. The fact that different types of instructions require different amounts of memory is important to the programmer.

As programmers write a program it is often necessary for them to keep tabs on how many words of memory the actual operating portion of the program will require (in addition to controlling the areas in memory that will be used for data storage). One reason for maintaining a count of the number of memory words a program requires is simply to ensure that the program will fit into the available memory space.

Often a program that is a little too long to be stored in an available amount of memory when first developed can be rewritten, after some thought, to fit in the available space. Generally, the trade-off between writing compact programs versus not-so-compact routines is simply the programmer's development time. Hastily constructed programs tend to require more memory storage area because the programmer does not take the time to consider memory conserving instruction combinations.

However, even if one is not concerned about conserving the amount of memory used by a particular program, one still often needs to know how much space a group of instructions will consume in memory. This is so that one can tell where another program might be placed without interfering with a previous program.

For these reasons, programmers often find it advantageous to develop the habit of writing down the number of memory words utilized by each instruction as they write the mnemonic sequences for a routine. Additionally, it is often desirable to maintain a column showing the total number of words required for storage of a routine. An example of a work sheet with this practice being followed is illustrated here:

MEMORY WORDS THIS INSTR.	TOTAL WORDS THIS ROUTINE	MNEMONICS	COMMENTS
2	2	LAI 000	Place 000 in accumulator
2	4	LHI 001	Set Register H to 1
2	6	LLI 150	And Regis L to 150
1	7	ADM	Add the contents of memory
1	8	INL	Locations 150 & 151 on page 1
1	9	ADM	Adding second number to first
1	10	RET	End of subroutine

In the example the total number of words used in column was kept using decimal numbers. Many programmers prefer to maintain this column using octal numbers because of the direct correlation between the total number of words used, and the actual memory addresses used by the 8008.

The example just presented can be used to introduce another consideration during program development. That is memory allocation. One must distinguish between program storage areas in memory, and areas used to hold data that is operated on by the program. Note that the sample subroutine was designed to have the computer add the contents of memory locations 150 and 151 on page 01. Thus, those two locations must be reserved for data. One must ensure that those specific memory locations are not inadvertantly used for some other purpose. In a typical program, one may have many locations in memory assigned for holding or manipulating data. It is important that one maintain some sort of system of recording where one plans to store blocks of data and

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
01	000				ADD,		Add no's @ 150 & 151
01	010						
01	020						
01	030						
01	040						
01	050						
01	060						
01	070						
01	100						
01	110						
01	120						
01	130						
01	140						
01	150						Number storage
01	151						Number storage
01	152						
01	153						
01	154						
01	155						
01	156						
01	157						
01	160						
01	170						
01	200						

MEMORY USAGE MAP

where various operating routines will reside as a program is developed. This can be readily accomplished by setting up and using memory usage maps (often commonly referred to as core maps). An example of a memory usage map being started for the subroutine just discussed is shown on the previous page.

The same type of form may also be used as a program development sheet as shown below. One may observe that the form provides for memory addresses, the actual octal values of the machine codes, labels and mnemonics used by the programmer, and additional information.

Memory usage maps are extremely valuable for keeping large programs organized as they are developed, or for displaying the locations of a variety of different programs that one might desire to have residing in memory at the same time. It is suggested that the person

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
01	000	006	000		ADD,	LAI 000	Set ACC = 000
01	002	056	001			LHI 001	Set pntr PG = 1
01	004	066	150			LLI 150	Set pntr LOC = 150
01	006	207				ADM	Add 1'st no. to ACC
01	007	060				INL	Adv pntr to 2'nd no.
01	010	207				ADM	Add 2'nd no. to 1'st
01	011	007				RET	Exit subroutine

PROGRAM DEVELOPMENT WORK SHEET

intending to do even a moderate amount of machine language programming make up a supply of such forms (using a ditto or mimeograph machine) to have on hand.

There are some important factors about machine language programming that should be pointed out as they have considerable impact on the total efficiency and speed at which one can develop such programs and get them operating correctly. The factors relate to one simple fact. People developing machine language programs (especially beginners) are very prone to making programming mistakes! Regardless of how carefully one proceeds, it always seems that any fair sized program needs to be revised before a properly operating program is achieved. The impact that changes in a program have on the development (or redevelopment) effort vary according to where in the program such changes must be made. The reason for the seriousness of the problem is because program changes generally result in the addresses of the instructions in memory being altered. Remember, if an instruction is added, or deleted, then all the remaining instructions in the routine being altered must be moved to different locations! This can have multiplying effects if the instructions that are moved are referred to by other routines (such as call and jump commands) because then the addresses referred to by those types of commands must be altered too! To illustrate the situation, a change will be made to the sample program presented several pages ago. Suppose it was decided that the subroutine should place the result of the addition calculation in a word in memory before exiting the subroutine, instead of simply having the result in the accumulator. The original program, for example, could have been residing in the locations shown on the program development work sheet on the previous page. Changing the program would result in it occupying the following memory locations:

PAGE	LOC	MEMORY CONTENTS	MNEMONICS	COMMENTS
01	000	006	LAI 000	Place 000 in accumulator
01	001	000		
01	002	056	LHI 001	Set Reg H to 1
01	003	001		
01	004	066	LLI 150	Set Reg L to 150
01	005	150		
01	006	207	ADM	Add contents of memory
01	007	060	INL	Locations 150 & 151
01	010	207	ADM	Add 2nd to 1st
01	011	066	LLI 160	Set Reg L to 160
** 01	012	160		
** 01	013	370	LMA	Save answer @ 160
** 01	014	007	RET	End of subroutine

The ** locations denote the additional memory locations required by the modified subroutine. If the programmer had already developed a routine that resided in locations 012, 013, or 014, the change would require that it be moved!

If one was using a program development work sheet, one would have had to erase the original RET instruction at the end of the routine and then written in the two new commands, and added the RET instruction at the end. The effects would not be too devestating since the change was inserted at the end of the subroutine. But, suppose a similar change was necessary at the start of a subroutine that had 50 instructions in it? The programmer would have to do a lot of erasing!

The effects of changes in program source listings was recognized early as a problem in developing programs. Because of this people developed programs called EDITORS that would enable the computer to assist people in the task of creating and manipulating source listings for programs. An EDITOR is a program that will allow a person to use a computer as a text buffer. Source listings may be entered from a keyboard or other input device and stored in the computer's memory. Information that is placed in the text buffer is kept in an organized fashion, usually by lines of text. An Editor program generally has a variety of commands available to the operator to allow the information stored in the text buffer to be manipulated. For instance, lines of information in the text buffer may be added, deleted, moved about or inserted before other lines, and so forth. Naturally, the information in the buffer can be displayed to the operator on an output device such as a cathode ray tube (CRT) or electromechanical printing mechanism. Using this type of program, a programmer can rapidly create a source listing and modify it as necessary. When a permanent copy is desired, the contents of the text buffer may be punched on paper tape or written on a magnetic tape cassette. It turns out that the copy placed on paper tape or a cassette can often be further processed by another program to be discussed shortly which is termed an ASSEMBLER program. However, an important reason for making a copy of the text buffer on paper tape or magnetic cassette tape is because if it is ever necessary to make changes to the source listing, then the old listing can be quickly reloaded back into the computer. Changes may then be rapidly made using the Editor program, and a new clean listing obtained in a fraction of the time that might be required to erase and rewrite a large number of lines using pencil and paper.

Relatively small programs can be developed using manual methods. That is, by writing the source listings with pencil and paper. But, anyone that is planning on doing extensive program development work should obtain an Editor program in order to substantually increase their overall program development efficiency. Besides, an Editor program can be put to a lot of good uses besides just making up source listings! Such as enabling one to edit correspondence or prepare written documents that are nice and neat in a fraction of the time required by conventional methods.

Changes in source listings naturally result in changes to the machine codes (which the mnemonics simply symbolize). Even more important, the addresses associated with instructions often must be changed due to additions or deletions of words of machine code. For instance, in the example routine being used in this section, memory address PAGE 01 LOCATION 011 originally contained the code for a RET (RETURN) instruction which is 007. When the subroutine was changed by adding several more instructions (so the answer could be stored in a memory location), the RET instruction was shifted down to the address PAGE 01 LOCATION 014. The address where it formerly resided was changed to hold the code for the first part of the LLI 160 instruction which is 066. Had changes been made earlier in the routine, then many more memory locations would need to be assigned different machine codes. However, the changes caused by adding on to the sample program previously discussed are not as far reaching as the one presented on the following page. There the changes result in the addresses of subroutines referred to by other routines being changed, so that it is then necessary to go back and modify the machine codes in all of the routines that refer to the subroutine that was changed!

```
                   MEMORY
PAGE    LOC      CONTENTS   MNEMONICS                  COMMENTS

 00     000        026      OVER,     LCI 100          Load reg C with 100
 00     001        100
 00     002        106                CAL NEWONE       Call a new subroutine
 00     003        013
 00     004        000
 00     005        106                CAL LOAD         And then another
 00     006        023
 00     007        000
 00     010        104                JMP OVER         Jump back & repeat
 00     011        000
 00     012        000
 00     013        056      NEWONE,   LHI 000          Load reg H with zeroes
 00     014        000
 00     015        066                LLI 200          And L with 200
 00     016        200
 00     017        317                LBM              Fetch mem contents to B
 00     020        010                INB              Increment the value in B
 00     021        371                LMB              Place B back into memory
 00     022        007                RET              End of subroutine
 00     023        056      LOAD,     LHI 003          Set H to PG 03
 00     024        003
 00     025        361                LLB              Place register B into L
 00     026        370                LMA              Place ACC into memory
 00     027        021                DCC              Decrement value in reg C
 00     030        013                RFZ              Return if C is not zero
 00     031        000                HLT              Halt when C = zero
```

Suppose it was decided to insert a single word instruction right after the LCI 100 command in the above program. The new program would appear as shown next.

```
                   MEMORY
PAGE    LOC      CONTENTS   MNEMONICS                  COMMENTS

  00    000        026      OVER,     LCI 100          Load reg C with 100
  00    001        100
  00    002        250                XRA              Clear the accumulator
* 00    003        106                CAL NEWONE       Call a new subroutine
* 00    004     ** 014
* 00    005        000
* 00    006        106                CAL LOAD         And then another
* 00    007     ** 024
* 00    010        000
* 00    011        104                JMP OVER         Jump back and repeat
* 00    012        000
* 00    013        000
```

PAGE	LOC	MEMORY CONTENTS	MNEMONICS		COMMENTS
* 00	014	056	NEWONE,	LHI 000	Load Reg H with zeroes
* 00	015	000			
* 00	016	066		LLI 200	And L with 200
* 00	017	200			
* 00	020	317		LBM	Fetch mem contents to B
* 00	021	010		INB	Increment the value in B
* 00	022	371		LMB	Place B back into memory
* 00	023	007		RET	Exit subroutine
* 00	024	056	LOAD,	LHI 003	Set H to PAGE 03
* 00	025	003			
* 00	026	361		LLB	Place reg B into L
* 00	027	370		LMA	Place ACC into memory
* 00	030	021		DCC	Decrement value in reg C
* 00	031	013		RFZ	Return if C is not zero
* 00	032	000		HLT	Halt when C is zero

Note in the illustration how not only the addresses of all the instructions beyond location 002 (denoted by the *) change, but even more important, that parts of the instructions themselves (the address portion of the CAL instructions, denoted by the **) must now be altered. The essential point being made here is that if the starting address of a routine or subroutine that is referred to by any other part of the program is changed, then each and every reference to that routine must be located and the address portion corrected! This can be an extremely formidable, time consuming, tedious, and down right frustrating task if all the references must be found and corrected by manual means in a large program!

Early computer technologist soon became disgusted with making such program corrections by hand methods after learning that it was almost impossible to develop large programs without making a few errors. They went to work on finding a method to ease the task of making such corrections and came up with a type of program called an ASSEMBLER that could utilize the computer itself to perform such exacting tasks. ASSEMBLER programs are types of programs that are able to process source listings when they have been written in mnemonic (symbolic) form and translate them into the OBJECT code (actual machine language code) that is utilized directly by the computer. An ASSEMBLER also keeps track of assigning the proper addresses to references to routines and subroutines. This is accomplished through a process initiated by the programmer assigning LABELS to routines in the source listing. One may now see that the combination of an Editor and an Assembler program can greatly ease the task of developing machine language programs over that of the purely manual method. The use of such programs is almost mandatory when programs become large because the manual method becomes highly unwieldy. A primary reason that an Editor and Assembler are so useful is because if a mistake is made in the program, one can use the relatively quick method of utilizing the Editor program to revise the source listing. Then, one may use the Assembler program to reprocess the corrected source listing and produce a new version of the machine code assigned to new addresses if appropriate.

For quite small programs, say less than 100 instructions, the use of Editor and Assembler programs are not mandatory. In fact, even if one uses these aids for small programs, one should know how to manually convert mnemonic listings to object code. This is because it may occasionally be desirable to make minor program changes (patches) without having to go through the process of using an Editor and Assembler. This is particularly true when one is DEBUGGING large programs and wants to ascertain whether a minor correction will correct a problem. The process of converting from a mnemonic listing to actual machine code is not difficult in concept. Many readers will have discerned the process from the examples already provided. However, for any who are in doubt, the process will be explained for the sake of clarity.

Suppose a person desired to produce a small program that would set the contents of all the words in PAGE 01 of memory to 000. The programmer would first develop the algorithm and write it down as a mnemonic (source) listing. Such an algorithm might appear as follows.

```
         MNEMONIC                    COMMENTS

              LHI 001         Set the high address register to PAGE 01.
              LLI 000         Set the low address register to the first
                              location on the page assigned by reg. H.
     AGAIN,   LMI 000         Load the contents of the memory location
                              specified by registers H & L to 000.
              INL             Advance register L to the next memory
                              location (but do not change the page).
              JFZ AGAIN       If the value of register L is not 000
                              after it has been incremented then JUMP
                              back to the part of the program denoted by
                              the label AGAIN and repeat the process.
              HLT             If the value of register L is 000, then have
                              the computer stop as the program is done!
```

To convert the source listing to machine (object) code the programmer must first decide where the program is to reside in memory. In this particular case it would certainly not be wise to place the program anywhere on PAGE 01 as the program would self-destruct! The program could safely be placed anywhere else. For the sake of demonstration it will be assumed that it is to reside on PAGE 02 starting at LOCATION 100. To convert the source listing to machine code the programmer would simply make a list of the addresses to be occupied by the program. Then the programmer would simply look up the machine code corresponding to the mnemonic for each instruction and place this number next to the address in which it will reside. (The machine code for each mnemonic used by the '8008' CPU is provided in Chapter ONE of this manual.) Since some instructions are location dependent in that they require the actual address of referenced routines, it is often necessary to assign the machine code in two processes. The first process consist of assigning the machine codes to specific memory addresses wherever possible. When the machine code requires an address that has not yet been determined, the memory location is left blank. The second process consist of going back and filling in any blanks once the addresses of referenced routines have been determined. In the example being used for illustration, only one process is required because the address specified by the label AGAIN is defined before the label (address) is referenced by the JFZ instruction. The sample program when converted to machine language code would appear as shown next.

ORIGINAL MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LHI 001	02 100	056	Machine code for LHI mnemonic
	02 101	001	Immediate part of LHI mnemonic
LLI 000	02 102	066	Machine code for LLI mnemonic
	02 103	000	Immediate part of LLI mnemonic
AGAIN, LMI 000	02 104	076	Machine code for LMI mnemonic Note that the label AGAIN now defines an address of LOCATION 104 on PAGE 02
	02 105	000	Immediate part of LMI mnemonic
INL	02 106	060	Increment low address here
JFZ AGAIN	02 107	110	Machine code for JFZ mnemonic
	02 110	104	Low address portion of the CONDITIONAL JUMP instruction as defined by label AGAIN above
	02 111	002	PAGE address portion of the CONDITIONAL JUMP instruction defined by label AGAIN
HLT	02 112	377	Alternately, the code 000 or 001 could have been used here as the machine code for a HALT command

Once the program has been put in machine language form the actual machine code may be placed in the assigned locations in memory. The programmer may then proceed to verify the algorithm's validity. For small programs such as the example just illustrated the machine code can simply be loaded into the correct memory locations using manual methods typically provided on microcomputer systems. Such small programs can then be easily checked out by stepping through the program one instruction at a time.

If the program is relatively large then a special loader program which is typically provided with an ASSEMBLER program could be used to load in the machine code.

Checking out and DEBUGGING large programs can sometimes be difficult if a few simple rules are not followed. A good rule of thumb is to first test out each subroutine independently. One may choose to STEP through a subroutine, or else to place HALT instructions at the end of each subroutine. Then one may verify that data was manipulated properly by a particular subroutine before going on to the next section in a program. The use of strategically located HALT instructions in a program initially being tried out is an important technique for the programmer to remember. When a HALT is encountered the user may check the contents of memory locations and examine the contents of CPU registers to determine if they contain the proper values at that point in the program. (Using the manual operator controls and indicator lamps typically provided with microcomputer development systems.) If all is well at a check point then the programmer may replace the HALT instruction with the actual instruction for that point. One may then continue checking the operation of the program after making certain that any registers that were altered by the examination procedure (typically registers H and L in an '8008' system) have been reset to the desired values if they will effect operation of the program as it continues!

It is often helpful to use a utility program known as a MEMORY DUMP program to check the contents of memory locations when testing a new program. A memory dump program is a small utility program that will allow the contents of areas in memory to be displayed on an output device. Naturally, the memory dump program must reside in an area of memory outside that being used by the program being checked. By using this type of program the operator may readily verify the contents of memory locations before and after specific operations occur to see if their contents are as expected. A memory dump program is also a valuable aid in determining whether a program has been properly loaded or that a portion of a program is still intact after a program under test has gone errant.

One will find that having flow charts and memory maps at hand during the DEBUGGING process is also very helpful. They serve as a refresher on where routines are supposed to be in memory and what the routines are supposed to be doing.

If minor corrections are necessary or desired, then one may often make program corrections, or PATCHES as they are commonly referred to by software people, to see if the corrections believed appropriate will work as planned. An easy way to make a PATCH to a program is to replace a CALL or JUMP instruction with a CALL to a new subroutine that contains the desired corrections (plus the original CALL or JUMP instruction if necessary). If a CALL or JUMP instruction is not available in the vicinity of the area where a correction must be made then one can replace three words of instructions with a CALL patch provided that one is very careful not to split up a multi-word instruction. If this cannot be avoided, then the remaining portion of a split-up multi-word instruction must be replaced with a NO-OPERATION instruction such as a LAA command (in an '8008' system). One must also make certain that the instructions displaced by the inserted CALL instruction are placed in the patching subroutine (provided that they are not being removed purposely). An example of several patches being made to the small example program previously discussed will be illustrated next.

Suppose, in the example just presented, that the operator decided not to clear (set to 000) all the words in PAGE 01 of memory, but rather to only clear the locations 000 to 177 (octal) on the page. The program could be modified by replacing the JFZ AGAIN instruction which started at LOCATION 107 on PAGE 02 with the command CAL 000 003 (CALL the subroutine starting at LOCATION 000 on PAGE 03 which will be the PATCH). Now at LOCATION 000 on PAGE 03 one could put:

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LAI 200	03 000	006	Put value 200 into the ACC
	03 001	200	Note value of 200 used because contents of register L has been incremented

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
CPL	03 002	276	Compare contents of the ACC with the contents of register L
JFZ AGAIN	03 003	110	If accumulator and L do not
	03 004	104	match then continue with the
	03 005	002	original program
RET	03 006	007	End of PATCH subroutine

Suppose instead of filling every word on PAGE 01 with zeroes the programmer decided to fill every other other word? A patch could be made by replacing the LMI 000 command at LOCATION 106 on PAGE 02 and again inserting a CAL 000 003 command to a patch subroutine that might appear as illustrated below.

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LMI 000	03 000	076	Keep the LMI instruction
	03 001	000	as part of the PATCH
INL	03 002	060	Keep original increment L
INL	03 003	060	And add another increment L to skip every other word
RET	03 004	007	Exit from PATCH subroutine

Finally, to illustrate a patch that splits a multi-word command, consider a hypothetical case where the programmer decided that prior to doing the clearing routine, it would be important to save the contents of register H before setting it to PAGE 01. If a three word CALL command is placed starting at LOCATION 100 on PAGE 02 in the original routine to serve as a PATCH, it may be observed that the second half of the LLI 000 instruction would cause a problem when the program returned from the patch. (The value of 000 at LOCATION 103 on PAGE 02 in the example program would be interpreted as a HLT command by the computer when it returned from the patch subroutine.) In order to avoid this problem the programmer could place a LAA (effectively a NO-OPERATION command) at LOCATION 103 on PAGE 02 after placing the patch command CAL 000 003 instruction beginning at LOCATION 100 on PAGE 02. The actual patch subroutine might appear as shown below.

MNEMONIC	MEMORY ADDRESS	MEMORY CONTENTS	COMMENTS
LEH	03 000	345	Save register H in register E
LHI 001	03 001	056	Now set register H to point
	03 002	001	to PAGE 01
LLI 000	03 003	066	And set the low address
	03 004	000	pointer to LOCATION 000
RET	03 005	007	End of PATCH subroutine

In the balance of this manual numerous techniques for developing machine language programs will be presented and discussed. Many of the examples used will be presented as subroutines that the reader may use when developing customized programs. It is important for the new programmer to learn to think of programs in terms of routines or subroutines and then learn to combine subroutines into larger programs. This practice makes it easier for the programmer to initially develop programs. It is generally much easier to create small algorithms and then combine them, in the form of subroutines, into larger programs. Remember, subroutines are sequences of instructions that can be CALLED by other parts of a program. They are terminated by RETURN or CONDITIONAL RETURN commands. It is also wise when developing programs to leave some room in memory between subroutines so that patches can be inserted or routines lengthened without having to rearrange the contents of a large amount of memory. Finally, while speaking of subroutines, it will be pointed out that the user would be wise to keep a note book of subroutines that the individual develops in order to build up a reference library of pertinent routines. It takes time to think up and check out algorithms. It is very easy to forget just how one had solved a particular problem six months after one initially accomplished the task. Save your accrued efforts. The more routines you have to utilize, the more valuable your machine becomes. The power of the machine is all determined by WHAT YOU PUT IN ITS MEMORY!

Before going on to the next section of this manual, the essential steps in the process of creating a program will be presented for review and to serve as a reference.

1. First, the programmer should clearly define and write down on paper exactly what the program is to accomplish.

2. Next, flow charts to aid in the complex task of writing the mnemonic (source) listings are prepared. They should be as detailed as necessary for the programmer's level of experience and ability.

3. Memory maps should be used to distribute and keep track of program storage areas and data manipulating regions in available memory.

4. Using the flow charts and memory maps as guides, the actual source listings of the algorithms are written using the symbolic representations of the instructions. An Editor program is frequently used to good advantage at this point.

5. The mnemonic source listings are converted into the actual machine language numerical codes assigned to specific addresses in memory. An Assembler program makes this task quite easy and should be used for large programs.

6. The prepared machine code is loaded into the appropriate addresses in the computer's memory and operation of the program is verified. Often the initial check out is done using the STEP mode of operation, or by exercising individual subroutines. The judicial use of inserted HALT instructions at key locations will often be of value during the initial testing phase.

7. If the program is not performing as intended then problem areas must be isolated. Program PATCHES may be utilized to make minor corrections. If serious problems are found it may be necessary to return to step no. 3, or step no. 1!

The first section of this chapter will be devoted to illustrating a number of simple instructions and sequences of instructions that may be used to accomplish commonly required functions. Novice programmers need to build up a repertoire of such routines in their mind so that they can learn to think in terms of the functions they perform as they prepare to develop programs of their own. Alternative ways of performing functions will sometimes be presented to illustrate advantages and disadvantages of one method over another. There will often be many other ways of performing the desired function other than that presented and the reader should feel free to think of other ways and look at possible advantages and negative aspects of such alternatives.

CLEARING THE ACCUMULATOR

It is often desirable to set the contents of the accumulator (ACC for abbreviation in this text) to zero before starting an operation, such as a mathematical calculation. One obvious way to do this is to use an LAI 000 instruction. A less obvious way is to use an XRA (EXCLUSIVE OR the contents of the ACC with itself)! The XRA method only requires one word, whereas the LAI 000 requires two. Also, the XRA method will set all the CPU flags to known states as any Boolean Logic instruction causes the S and P flags to be affected and the C flag to be set to the zero state. (Whenever necessary the reader should refer to the appropriate section in Chapter One of this programming manual to review the detailed function(s) of each type of instruction available in an 8008 based microcomputer). Since the XRA instruction will set the ACC to all zeroes, then the Z and P flags will be placed in the '1' condition, and the S flag to the '0' state at the conclusion of the instruction's execution. It is important to remember the types of instructions that affect the operation of the CPU flags. This is because it is often necessary to use the status of a flag or flags to control the operation of a program. Or, to see if a flag's status has changed. To do this, one must at some time know what the condition of a flag was. That is often achieved by using an instruction such as the XRA that will force them to desired states. On the other hand, while the LAI 000 method of clearing the ACC requires two memory words, the execution of an LAI 000 instruction does not affect the status of the CPU flags. This fact should be remembered, because there may be times when it is desirable to set the ACC to the zeroes condition without altering the CPU flags!

SETTING THE ACCUMULATOR TO ALL ONES

This function can be accomplished with several types of instructions, such as the LAI 377 or ORI 377. Both these instructions require two words of memory. It should be noted again that the LAI 377 type will not affect the status of the CPU flags, while the ORI 377 one will result in the C and Z flags being set the the '0' state, and the S and P flags set to the '1' condition. If a particular program requires the accumulator to be set to the all ones state frequently, then it may be worthwhile to set up a CPU register to contain 377. Then one may use a one word instruction, such as LAX (X = a CPU Register) or an ORX, depending on whether or not one wants to save the status of the CPU flags.

COMPLEMENTING THE ACCUMULATOR

Often it is desirable to COMPLEMENT the value in the accumulator. That is to change all the bits set to a '1' to be '0' and vice-versa. This can be readily accomplished by using an XRI 377 instruction. Again, if the function must be performed often in a routine, it may be worthwhile to keep the value 377 in a CPU register and use a XRX instruction. The complement function is often utilized when performing mathematical operations using signed numbers (as explained in the previous chapter) in order to obtain the two's complement form of a number. The two's complement of a number is obtained by first complementing the value and then adding one to the complemented value. Thus, this function could be obtained by performing two kinds of instructions in sequence. First an XRI 377, and then an ADI 000 command.

FORMING BIT MASKS

When utilizing a computer, it is frequently desirable not to use all the bit positions within a word, or to isolate and determine the status of a particular bit within a register. This technique may be used to quickly determine whether a number in a register is odd or even (by examining just the least significant bit). Or, whether a number has reached a certain size (by sampling the most significant bit of interest). Or, whether some particular external event has occurred (by checking a specific bit on an input port).

The process of ridding a register of unwanted data in selected bit positions is commonly referred to by computer technologists as MASKING. Masking can be accomplished in several ways depending on what the programmer desires. Suppose, for instance, that one desired to determine whether a number in the accumulator was odd or even. One way to do this would be to simply execute an NDI 001 instruction. Then test to see if the accumulator was zero (using a JTZ or JFZ command). Suppose the original number in the accumulator had been 251. (Remember that this text is using octal numbers unless otherwise stated!) The result of performing the logic AND operation between the accumulator containing 251 and the number 001 is illustrated below.

```
   ACC = 1 0  1 0 1  0 0 1 = Octal 251

ANI 001 = 0 0  0 0 0  0 0 1 = Octal 001
          -----------------------
 RESULT = 0 0  0 0 0  0 0 1 = Octal 001
```

It may be observed that all the bit positions ANDED with a '0' will go to the '0' condition regardless of whether they were a '1' or a '0.' Thus, the seven most significant bit positions in the example have been effectively eliminated. However, a bit position ANDED against a '1' will be a '1' if, and only if, the position under test contains a '1.' In the above case, a '1' was present in the test position and thus the result was a '1.' A JFZ instruction would quickly direct the program to proceed on the basis that the original number in the ACC had been an odd number.

Note that the above particular masking method was destructive to the original value in the accumulator. Had it been important, the original number could have been saved in a CPU register or a memory location. A slightly different approach could have been taken. The number to be masked could be placed in a memory location, or a CPU register. Then the accumulator could be filled with the appropriate MASK. Finally, a simple one word NDM or NDX instruction could be utilized. The result of the masking operation would be left in the accumulator after the execution of the instruction. The original number would be available for further manipulation. This different approach is pointed out as an example of how a programmer should look for the best method to approach a particular problem. The computer, with its variety of instructions, provides many different methods to choose from for such problems.

Masking is most effective when there are several bits in a register to be isolated, or when a bit of interest is in the middle of a word. Or, when it may not be expedient to bring a piece of data into the accumulator. For if one desires to examine the status of a bit in the ACC that is at either end of a register, one may do this by using a rotate instruction such as RAL or RAR to put the bit of interest into the CARRY position of the ACC (represented by the CARRY FLAG). Then use a JTC or JFC instruction to determine the status of the bit. Naturally, if the programmer wanted to retain the original setting of the accumulator after the test, the program would have to execute the reverse rotate instruction (to the one originally used). This would bring the ACC back to its original position.

SETTING UP POINTERS AND COUNTERS

In many applications it is desirable to perform a particular sequence of operations a precise number of times. The number of times an operation is performed can be controlled in a routine by forming a program loop. A program loop is established by setting up a counter system that keeps track of how many times an operation is performed and including a program test to ascertain when a particular value has been reached so that program control can be branched out of the loop.

In an 8008 system, CPU registers make handy loop counters as they not only can be directly incremented or decremented by one word commands, but they also directly affect the status of the Z, S, and P CPU flags after each increment or decrement. It is thus an easy matter to use any one of the conditional type instructions immediately following a CPU register increment or decrement to see if a critical value has been reached!

For instance, suppose register B is initially set to the value 012 (10 decimal) by a LBI 012 instruction prior to execution of the following program loop.

```
MORE,  LMA        Load contents of ACC into memory
       INL        Advance memory pointer
       DCB        Decrement the loop counter
       JFZ MORE   If reg B is not = 000, continue loop
DONE,  HLT        Exit subroutine when counter = 000
```

As may be observed, the above subroutine would loop upon itself and load data into consecutive words in memory until the value placed in register B (prior to starting the subroutine) reached zero. In the above example, B was loaded with 012 so 12 octal (10 decimal) locations in memory would have been loaded with data. (It can be assumed that the calling routine set up registers H and L to point to the proper memory locations, and placed the correct data into the accumulator!)

To illustrate how powerful the simple concept of a program loop is, a second example will be used to illustrate how such a loop technique may be used to perform multiplication of small numbers. (There are much more efficient programming techniques available for use with large numbers.) Since multiplication is really just repeated addition, one could multiply two numbers, designated X and Y, by performing the following operations. Assume X is the multiplicand and it has been loaded into CPU register C. The number Y is the multiplier, and it has been placed in register B. The following routine containing a program loop will multiply the two numbers.

```
    START,  XRA           Clear the accumulator
   CONTIN,  ADC           Add contents of register C to ACC
            DCB           Decrement value of the multiplier
            JFZ CONTIN    Repeat addition if mult. is not = zero
     EXIT,  RET           Exit subrtn with mult. answer in ACC
```

As readers know, the CPU registers H and L are able to serve as ordinary CPU registers and also have the special function of being able to point to addresses in memory whenever memory reference instructions are used. The H register holds the high address or page portion of the pointer. The L register holds the low address or location on a page. Naturally, when one desires to operate on data at a location in memory via a memory reference command, one must first set up the H and L registers to contain the desired address. This is readily done with a LHI XXX and LLI YYY combination of instructions. However, many times it is desirable to do a whole sequence of operations that operate upon sequential locations in memory. In this case, once the initial starting address has been loaded into the memory pointer registers, all that is needed is a subroutine that can be referred to that will increment the address held in the two registers. A simple subroutine to accomplish that objective is presented here.

```
     ADV,  INL           Increase value of register L by 1
           RFZ           Exit subrtn if not going to new page
           INH           Increment H by 1 if on new page
           RET           Exit subrtn
```

The above subroutine takes care of the case where the address crosses page boundaries. Each time register L is advanced, the RFZ instruction is used to test whether or not register L went to 000. This would occur if the last value in the register had been 377. That is the largest octal address that can be represented in an 8 bit register. Consequently, it is the highest address that can be assigned on a page of memory. If the RFZ instruction is executed (because the contents of L did not go to 000) then the routine is immediately exited. If the RFZ command is not followed, then the subroutine continues and advances the contents of register H to update the pointer to a new page.

Fine. But what about the opposite case when a programmer might desire to process areas of memory in descending order? Well, a similar subroutine to decrement the memory pointer registers could be used. But, the programmer will have to be careful when going to a new page. In the previous case, when the L register was advanced beyond location 377 to 000, it was an easy matter to check for the 000 condition to see if it was necessary to advance the H register too. Now, however, when the L register goes from 000 to 377 it will be necessary to decrement the H register to the next lower page. Testing for this condition is not quite as easy. Remember, the status of the CPU flags are set by the conditions in the register immediately after they have been incremented or decremented, not before. While one may use a JTZ or RFZ type of instruction to quickly determine if a register went to 000, the case where it did not go to 000 does not mean it is necessarily at 377. It could be at any non-zero value. However, the case can be handled. One way to handle the problem would be with the subroutine shown next.

```
DEC,    XRA           Clear ACC to 000
        CPL           Compare contents of ACC with L
        JTZ DECH      If 000 now, then DECR both H & L
        DCL           Otherwise just decrement L
        RET           And exit subroutine
DECH,   DCL           For this case decrement L
        DCH           And register H
        RET           Then exit subroutine
```

While the above subroutine will accomplish the objective, it does have several minor flaws that the programmer might want to consider. First, it alters the contents of the accumulator. Remember that the above subroutine might often be used in a program that is manipulating data between the accumulator and memory. The above subroutine would require that the programmer make sure any valuable data in the accumulator is saved elsewhere before the subroutine is called. This is one more burden on the programmer who is developing a large program and many have a lot of other details to think about. Secondly, the above routine requires 10 decimal memory storage locations. It is always a good practice to try and develop routines that operate in a minimum amount of memory. Lets take a look at another subroutine that accomplishes exactly the same objective, that saves 20 percent of memory space, and that will not interfere with the original contents of the accumulator.

```
DECR,   DCL           Decrement contents of L
        INL           Now check to see if it had been 000
        JFZ NOTO      If not 000 then not going to new page
        DCH           If 000 then DECR H to next lower page
NOTO,   DCL           Decrement L to complete subroutine
        RET           Exit subroutine
```

The above subroutine used a little programming creativity to come up with a method of accomplishing the desired objective. Register L was decremented and then incremented back to its original value. The process of incrementing it back to its original value would cause the CPU flags to be set so that a flag testing instruction could be used to see if the original value was 000. If that was the case, decrementing it would cause it to go to 377, and thus register H should be decremented to the next lower page. That is done if necessary. Then register L is decremented to its final value whether or not the address is going to a new page!

While registers H and L are the only registers that may be used to point to memory locations when using memory reference instructions in an 8008 machine, it is often necessary to use other CPU registers to temporarily hold memory addresses. It may be desirable, for instance, to transfer blocks of data from one area in memory to another. This must be done one word at a time. First a word must be extracted from memory location M by say a LAM instruction, with registers H and L pointing to address M. Then H and L must be altered to an address, lets call it N, where the data is to be deposited. An LMA instruction could then be used to place the data in the new memory location. Often a string of data words might be transferred in such a fashion. It would be rather cumbersome if one had to keep using LHI MMM and LLI MMM commands followed by LHI NNN and LLI NNN instructions in order to keep altering the memory pointer registers between the two areas in memory. However, if H and L were initially set to point to memory location M, and CPU registers D (say for the page address) and E (for the address on the page) were set to point to memory location N, then a switching program to exchange the contents of H with D and L with E could be developed to considerably ease the task. Such a subroutine might be as follows.

```
SWITCH,  LCH          Load contents of H into C temporarily
         LHD          Now load D into H
         LDC          Move original H from C into D
         LCL          Similarly load L into C temporarily
         LLE          Put E into L
         LEC          And store original L in E
         RET          Exit subroutine
```

Now, by simply calling the subroutine to switch the contents of the registers, the programmer has a means of changing the memory pointer registers between two different areas in memory. To illustrate how quickly a library of small subroutines starts developing into real potential, two subroutines illustrated on the last several pages will be used in a small program to accomplish the task just discussed, which is that of moving data from one area of memory to another. Let's assume that a programmer desired to move the data in 100 (octal!) words of memory starting at location 000 on page 02 up to an area starting at location 200 on page 03. The following program would do the job nicely.

```
SETUP,   LHI 002      Set up H to page of 1st memory area
         LLI 000      And L to starting location of 1st area
         LDI 003      Set D to page of 2nd memory area
         LEI 200      And E to starting location of 2nd area
         LBI 100      Set up a counter in CPU register B
MOVIT,   LAM          Get contents of word from 1st mem area
         CAL ADV      Advance memory pointer (in 1st area)
         CAL SWITCH   Change H & L to point to 2nd area
         LMA          Deposit word in 2nd area
         CAL ADV      Advance memory pointer (in 2nd area)
         CAL SWITCH   Change back to point to 1st memory area
         DCB          Decrement counter
         JFZ MOVIT    If counter not = 000, then continue moving
         RET          Exit RTN (or HLT, JMP, etc.)
```

USING MEMORY LOCATIONS TO STORE POINTERS AND COUNTERS

While CPU registers make ideal storage places for pointers and counters because they can be directly incremented and decremented, there are simply not enough of them to store all the pointers and counters that might be used in a fair sized program. It then becomes necessary to hold the values of counters and pointers in memory locations so that the CPU registers can be opened up for other uses. This practice does have a drawback. Since the contents of memory locations cannot be directly incremented, the contents must first be loaded into a CPU register, then the increment or decrement performed. Then the new value put back into its memory storage location. This takes a lot of extra instructions over that required if the counter or pointer can be kept permanently in a CPU register. This is especially so since to even obtain the counter from memory, it will always be necessary to first set up the H & L registers to point to the memory location where the counter or pointer is stored! However, since that is what has to be done in all but small programs, the best thing to do is to try and organize the process using subroutines that will reduce the amount of memory used by the operating program.

Perhaps the first item to consider is where to store the counters and pointers for a program. Well, it is generally a good idea to set aside a section of memory to be used exclusively for storing counters and pointers for the program. Preferably this should be on one page of memory (versus crossing page boundaries). While essentially any page may be used, it may be that for large programs having the pointers and counters on page 00 will save a bit of programming room. This is because whenever the program needs to refer to a counter, register H (as well as L) must be set up to point to the page where the counter is stored. It seems that there is often a zero register (one set to 000) around among the CPU registers. Thus a LHX one word instruction can be used to set H to the page instead of having to use a LHI XXX command as will generally be the case if the pointers and counters are not stored in an area on page 00.

Once one has decided where particular counters are to be stored, a subroutine to retrieve any one of them and increment or decrement the value, then restore it back to memory, is quite straight-forward.

```
CNTUP,   LCM       Fetch CNTR indicated by H & L
         INC       Increment value of the counter in reg C
         LMC       Restore new counter value to memory
         RET       Exit subroutine

CNTDWN,  LCM       Fetch counter
         DCC       Decrement value
         LMC       Return counter to storage
         RET       Exit subroutine
```

The two subroutines just illustrated can be called as desired to obtain a counter and increment or decrement the value once registers H and L have been loaded with the address of the counter. Note, too, that the subroutine would also allow the result of the increment or decrement to be tested by a conditional instruction after the subroutine is finished. This is because there are no instructions after the INC or DCC that affect the status of the CPU flags!

Storing pointers in memory is generally a little more complicated than storing counters because pointers generally require two storage locations. One word for the page address, and one for the location on the page. In addition, since the H & L registers will have to be used to point to where the pointers are stored in memory, and since the pointers stored in memory cannot be used as pointers until they are placed in the H & L registers, a method of first obtaining the new pointer into unused CPU registers, then swapping it with the H & L registers, must be used. The process is not so difficult if use is made of some of the subroutines (such as SWITCH) which have already been presented in this chapter.

The example illustrated next shows a general subroutine that will obtain a two word pointer stored in memory. Then use the pointer obtained to put the contents of the accumulator into a memory location specified by the pointer just obtained. Next, it will increment the pointer, then restore it back to its storage place in memory. The routine assumes that the H & L registers will be set to the page address of the location where the pointer is stored by the calling program, and that the pointer is stored in two consecutive words. First the page, and then the location on the page.

```
POINT1,  LDM           Fetch pointer page addr into reg D
         INL           Advance to pick up contents of next word
         LEM           Get location addr into register E
         CAL SWITCH    Put new pointer into H & L
         LMA           Put ACC into mem indicated by new pointer
         CAL ADV       Increment the new pointer
         CAL SWITCH    Restore new pointer storage address
         LME           Deposit pointer location addr in mem
         DCL           Decrement back to page addr storage word
         LMD           Deposit pointer page addr in mem
         RET           Exit subroutine
```

The reader should note a nice feature of the above subroutine. When the subroutine is finished the contents of H & L are set to point to the storage area of the pointer stored in memory. Thus, the subroutine could now be called again if desired without having to set up the H & L registers. Furthermore, when the routine is exited, CPU registers D & E will contain the latest value of the pointer stored in memory. This might be valuable in cases where further processing was to be done in the section of memory where the stored pointer was operating. For instance, examine the small program illustrated next.

```
BUFFIN,  LHI 000       Set page where buffer pointer stored
         LLI 240       Set location on page of buffer pointer
 INAGN,  CAL INPUT     Get a character from input device
         CAL POINT1    Put the character into mem buffer area
         CPI 215       See if char was ASCII code for CR
         JFZ INAGN     If not, get another character
         RET           Exit rtn when find a CR character
```

The above program, as short and simple as it looks, is really quite powerful. The reader may observe that it is a program that will store a string of characters received from an input device into a buffer area in memory. It will continue placing characters into the memory buffer area until it detects a CR (carriage-return) character. The location of the memory buffer area is stored in a pointer that is located at locations 240 (page) and 241 (location on the page) on page 00. Of course, before the above routine was used, the programmer would want to put the proper address for the buffer area into those locations. The above routine is really a general purpose routine to accept text sentences and store them in a memory buffer. To expand the above subroutine into a complete program requires very little additional effort. The following example illustrates this point.

```
DATAIN,  LHI 000      Set page where POINT1 pointer stored
         LLI 240      And address on the page for POINT1
         LMI 003      Set start of memory buffer area (PAGE)
         INL          Advance to next word
         LMI 000      Set start of memory buffer area (LOCATION on PAGE)
         LLI 250      Address of a LINE COUNTER
         LMI 012      Set LINE COUNTER to 10 (decimal)
MORIN,   CAL BUFFIN   Get a line of text
         LHI 000      Setup storage address of line counter
         LLI 250      Setup storage address of line counter
         CAL CNTDWN   Decrement LINE COUNTER value
         JFZ MORIN    If not 10 (decimal) lines then get another line
         HLT          End of program (could be JMP, RET, and so forth)
```

The above program first initializes the starting location of the text buffer to PAGE 03 LOCATION 000 by setting those values into the POINT1 memory storage words. It also initializes a counter stored in memory to a value determined by the programmer. Then the subroutine that inputs lines of text is called. Each time a line of text is obtained, the LINE COUNTER is decremented and a decision made as to whether or not another line of text should be obtained. When a predetermined number of lines of text have been obtained, the program stops. Instead of halting, however, the program could have been directed to proceed elsewhere by using a JMP command. Or, the entire program could have been made a subroutine by using a RET as the last instruction in the routine!

It is hoped that the reader is rapidly beginning to understand how quickly small, general purpose subroutines, start developing tremendous potential as they are teamed with other routines. Also, the reader should begin to see how the use of memory augments the capability of the CPU registers. By using memory locations to store pointers and counters, the programmer opens a whole new dimension in the world of programming. It is hoped the novice programmer becomes a little bit excited as these concepts are grasped and understood. These concepts are just the beginning! A little excitement stimulates the imagination and gives one incentive to go forward, investigate, and learn more!

Before going further, however, it might be wise to slow things down for just a moment and reiterate the importance of keeping a program organized as it is developed. In the last several pages, a number of subroutines were presented. They were then combined to form larger subroutines. Finally a small TEXT BUFFER INPUT PROGRAM was presented. The program presented used memory storage in a variety of ways. First the program itself had to be stored in memory. Secondly, operational portions of the program required memory storage areas for pointers and counters. Last, but not least, the program required the use of memory for manipulating DATA in the area called the TEXT BUFFER. Furthermore, the TEXT BUFFER INPUT PROGRAM really consisted of a whole group of small subroutines. Subroutines that could be stored in different areas in memory. What is needed, as has been discussed in the previous chapter, is a MEMORY MAP to help the programmer plan the allocation of memory. It might be worthwhile practice for the reader to develop a memory map for the program that has just been developed. A good method to follow would be to set aside room for the main part of the program (perhaps leaving a good amount of space for expanding the program if desired). Then the various subroutines may be assigned to areas, possibly leaving some room between each one in the event future modifications are desired or required. One might use a separate memory map for each page of memory in which subroutines are stored. In areas where counters and pointers are stored, the maps might be expanded to show the actual individual addresses of where the information is stored.

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
00	240				BUFPTH,		Pg addr of pointer
00	241				BUFPTL,		Low addr of pointer
00	242						
00	243						
00	244						
00	245						
00	246						
00	247						
00	250				COUNT,		Text LINE COUNTER
00	251						
00	252						
00	253						
00	254						
00	255						
00	256						
00	257						
00	260						
00	261						
00	262						
00	263						
00	264						
00	265						
00	266						
00	267						

EXPANDED MEMORY MAP SHOWING LOCATIONS OF POINTERS AND COUNTERS FOR THE TEXT BUFFER INPUT PROGRAM

The sample maps illustrated here show one way the program could be assigned to memory locations. The pointers and counters are placed on PAGE 00 as originally defined. The subroutines have been assigned to various areas in memory on PAGE 02. Some space has been left between each subroutine in case modifications to the program should be desired at some later time. Note how the use of the memory maps gives a coherence to the program that was not readily discernable when one simply tried to maintain the mental image of the organization of the program. (PAGE 03 is assumed to be used solely as the TEXT BUFFER storage area for the program and a memory map for the usage of that area is not illustrated.)

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
02	000				DATAIN,		Input 10 decimal lines of
02	010						text into buffer area - PG
02	020						03. Main rtn uses abt 30
02	030						octal locations - but leave
02	040						room for expansion.
02	200				BUFFIN,		Input 1 line of text, a CR
02	210						ends line of input.
02	220						
02	230				POINT1,		Fetch pntr locs in memory
02	240						designated by calling rtn -
02	250						dep ACC in memory, etc.
02	260				SWITCH,		Exch H&L with D&E
02	270				ADV,		Incr value in H&L
02	300				CNTDWN,		Decr cntr stored in mem.

SAMPLE MEMORY MAP OF THE TEXT BUFFER INPUT PROGRAM ILLUSTRATING THE MAIN ROUTINE AND SUBROUTINES ASSIGNED TO MEMORY AREAS ON PAGE 02

Once the memory maps have been made up and the starting addresses of all the subroutines assigned, it is an easy matter to convert the mnemonics to machine code. An assembler program may be used if available. For practice, the reader might want to try developing the machine code for the TEXT BUFFER INPUT PROGRAM just presented by hand. For comparison purposes the object code for the program would appear as listed below if the subroutines are assigned to the addresses as shown in the example memory map presented on the previous page.

```
02 000   056    DATAIN,  LHI 000       Set page where POINT1 pointer stored
02 001   000
02 002   066             LLI 240       And address on the page for POINT1
02 003   240
02 004   076             LMI 003       Set start of memory buffer area (page)
02 005   003
02 006   060             INL           Advance to next word
02 007   076             LMI 000       Set start of mem buff area (loc on page)
02 010   000
02 011   066             LLI 250       Address of a LINE COUNTER
02 012   250
02 013   076             LMI 012       Set LINE COUNTER to 10 (decimal)
02 014   012
02 015   106    MORIN,   CAL BUFFIN    Get a line of text
02 016   200
02 017   002
02 020   056             LHI 000       Setup storage address of LINE COUNTER
02 021   000
02 022   066             LLI 250       Setup storage address of LINE COUNTER
02 023   250
02 024   106             CAL CNTDWN    Decrement LINE COUNTER value
02 025   300
02 026   002
02 027   110             JFZ MORIN     If not 10 (dec) lines, get another line
02 030   015
02 031   002
02 032   000             HLT           End of pgm (could use RET, JMP, etc.)


02 200   056    BUFFIN,  LHI 000       Set page where buffer pointer stored
02 201   000
02 202   066             LLI 240       Set location on page of buffer pointer
02 203   240
02 204   106    INAGN,   CAL INPUT     Get a character from input device
02 205   †††
02 206   †††
02 207   106             CAL POINT1    Put the character into mem buffer area
02 210   230
02 211   002
02 212   074             CPI 215       See if char was ASCII code for CR
02 213   215
02 214   110             JFZ INAGN     If not, get another character
02 215   204
02 216   002
02 217   007             RET           Exit rtn when find a CR character
```

```
02 230   337      POINT1,  LDM          Fetch pointer page addr into register D
02 231   060               INL          Adv to pick up contents of next word
02 232   347               LEM          Get location address into register E
02 233   106               CAL SWITCH   Put new pointer into H & L
02 234   260
02 235   002
02 236   370               LMA          Put ACC into mem indicated by pntr
02 237   106               CAL ADV      Increment the new pointer
02 240   270
02 241   002
02 242   106               CAL SWITCH   Restore new pointer storage address
02 243   260
02 244   002
02 245   374               LME          Deposit pntr location address in mem
02 246   061               DCL          Decr back to page addr storage word
02 247   373               LMD          Deposit pointer page addr in memory
02 250   007               RET          Exit subroutine

02 260   325      SWITCH,  LCH          Load contents of H into C temporarily
02 261   353               LHD          Now load D into H
02 262   332               LDC          Move original H from C into D
02 263   326               LCL          Similarly load L into C temporarily
02 264   364               LLE          Put E into L
02 265   342               LEC          And store original L in E
02 266   007               RET          Exit subroutine

02 270   060         ADV,  INL          Increase value of register L by one
02 271   013               RFZ          Exit subroutine if not going to new page
02 272   050               INH          Increment H by one if on new page
02 273   007               RET          Exit subroutine

02 300   327      CNTDWN,  LCM          Fetch COUNTER
02 301   021               DCC          Decrement value
02 302   372               LMC          Return COUNTER to storage
02 303   007               RET          Exit subroutine
```

ORGANIZING AND MANIPULATING TABLES

A very powerful feature of a digital computer is its ability to store data and to process it as the programmer desires. The programmer may desire, perhaps, to have the data arranged into some specific kind of order, or to obtain some mathematical information such as the average of a group of data values. Or, one might desire to have the computer condense raw data into some sort of compact form by directing it to scan the data for relevant information. The digital computer is well suited for rapidly extracting information of particular interest from a memory storage area by performing such functions as matching similar types of data. Or, it may be used as a converting machine whereby data in one type of form (code) can be quickly changed to a different representation. In such applications as these, it is frequently necessary to develop programs that organize data into TABLES or to create programs that can process information stored in table-like format.

There are a variety of ways to organize tables for computer processing. The reader has already, whether it has been realized or not, been introduced to several types of TABLES in this manual. In the first chapter mention was made of using a LOOK-UP TABLE to convert between ASCII and BAUDOT codes used in various kinds of electric typing machines. In this chapter, the discussion and programming considerations for a text buffer were actually concerned with a FREE-FORM type of table.

For the purposes of the following discussion, two basic types of table organizations will be discussed. One will be referred to as FIXED-FORMAT, the other as FREE-FORMAT. The fixed-format type of table refers to tables that are fixed by programming considerations into strict, unchanging patterns of organization. The free-format kind use different programming techniques to allow the storage of data in random length sections of memory. There are advantages and disadvantages to each format. The choice of which one to use is generally a function of the type of task that is to be performed. Free-format organization is generally more suitable to text handling tasks. Fixed-format organization is generally the choice for conversion tables. There are also cases where the choice is a relatively minor one, and it becomes a matter of the programmer's preference.

To begin delving into the subject, a table with many practical applications will be discussed. Programming considerations for developing it in both types of formats will be presented. In many situations, it is desirable for a computer program to have a CONTROL TABLE. That is, a table that will interpret commands from an input device, and, depending on what is received, perform a specific type of function. For the purposes of this illustration, it will be assumed that an operator will type in commands from a keyboard. The commands will be in the form of words that may vary in length from '2' to '6' characters. Whenever a word has been inputted to the computer, the computer will check to see if the control table contains a matching word. If so, the computer will obtain the address of a routine that it is to perform, and execute the function. When it is through performing the routine, or if a match for the command was not found, the program will return to the COMMAND MODE. It will then wait for a new keyboard entry (after sending a response on a output device to notify the operator it is ready for a new entry). For this example, the output device will be assumed to be an electric typewriter.

For a hypothetical example, it will be proposed that the control words consist of the following: GO, LIST, MEDIAN, AVG, COUNT, and ERASE. These control words might be associated with a program that is to be used by a scientist conducting some type of experiment. Suppose the control command GO indicated the computer was to start a 10 second timing loop. At the start of the 10 second time period, the program would send a reset pulse to some sort of external counting device. The device might be counting the events that occurred in some kind of experiment. When the 10 second period was over, the computer would immediately obtain the value registered by the external counter. It would then store the number obtained in a data buffer. The LIST command might direct the computer to print out all the data values stored in the data buffer. (Perhaps the scientist could look for patterns or just have a copy of the raw experimental data.) The MEDIAN command could direct the computer to determine the median, or middle value out of all the values stored in the data buffer, and print out that number. Similarly, the AVG directive could signify that the program was to execute a routine to calculate the average value of the data. The COUNT command might be used to have the computer indicate how many 10 second experiments had been conducted. And, the ERASE command could signify that the data buffer was to be cleaned out for a new set of experiments.

The control table needs to be constructed so that the program can search for a word that is the same as that entered on the keyboard. If a match is found, then the table would contain information (an address) that would direct the computer to the proper routine to be executed. The control table could be constructed by setting aside an area in memory. That area could contain the proper code for the letters in each control word, followed by two memory words containing the page and low address of where the appropriate routine resided. If the control table was constructed in FIXED-FORMAT, it might appear as follows.

FIXED-FORMAT CONTROL TABLE

02 000	307	Code for letter G
02 001	317	Code for letter O
02 002	000	Not used for this command
02 003	000	Not used for this command
02 004	000	Not used for this command
02 005	000	Not used for this command
02 006	001	Page where GO routine starts
02 007	100	Loc on pg where GO starts
02 010	314	Code for letter L
02 011	311	Code for letter I
02 012	323	Code for letter S
02 013	324	Code for letter T
02 014	000	Not used for this command
02 015	000	Not used for this command
02 016	001	Pg where LIST routine starts
02 017	140	Loc on pg where LIST starts
02 020	315	Code for letter M
02 021	305	Code for letter E
02 022	304	Code for letter D
02 023	311	Code for letter I
02 024	301	Code for letter A
02 025	316	Code for letter N
02 026	001	Pg where MEDIAN rtn starts
02 027	200	Loc on page for MEDIAN
02 030	301	Code for letter A
02 031	326	Code for letter V
02 032	307	Code for letter G
02 033	000	Not used for this command
02 034	000	Not used for this command
02 035	000	Not used for this command
02 036	001	Pg where AVG routine starts
02 037	240	Loc on page where AVG starts
02 040	303	Code for letter C
02 041	317	Code for letter O
02 042	325	Code for letter U
02 043	316	Code for letter N

02 044	324	Code for letter T
02 045	000	Not used for this command
02 046	001	Pg where COUNT rtn starts
02 047	300	Loc on pg where COUNT starts
02 050	305	Code for letter E
02 051	322	Code for letter R
02 052	301	Code for letter A
02 053	323	Code for letter S
02 054	305	Code for letter E
02 055	000	Not used for this command
02 056	001	Pg where ERASE starts
02 057	340	Loc on pg where ERASE starts
02 060	000	**End of table marker**

It may be noted that the fixed-format table occupies memory from location 000 to 060 (including an end of table marker which will be discussed later). Observation of the table shows that there is a lot of wasted space where memory locations are filled with zeros when the command word did not require six characters. More characteristics of the above format will be presented shortly. First, two similar free-format versions of the same control table will be illustrated.

FREE-FORMAT CONTROL TABLE - VERSION NO. 1

02 000	307	Code for letter G
02 001	317	Code for letter O
02 002	000	*End of command word marker*
02 003	001	Page where GO routine starts
02 004	100	Loc on pg where GO starts
02 005	314	Code for letter L
02 006	311	Code for letter I
02 007	323	Code for letter S
02 010	324	Code for letter T
02 011	000	*End of command word marker*
02 012	001	Pg where LIST routine starts
02 013	140	Loc on pg where LIST starts
02 014	315	Code for letter M
02 015	305	Code for letter E
02 016	304	Code for letter D
02 017	311	Code for letter I
02 020	301	Code for letter A
02 021	316	Code for letter N
02 022	000	*End of command word marker*
02 023	001	Pg where MEDIAN rtn starts
02 024	200	Loc on pg for MEDIAN
02 025	301	Code for letter A
02 026	326	Code for letter V
02 027	307	Code for letter G
02 030	000	*End of command word marker*
02 031	001	Pg where AVG routine starts

02 032	240	Loc on page where AVG starts
02 033	303	Code for letter C
02 034	317	Code for letter O
02 035	325	Code for letter U
02 036	316	Code for letter N
02 037	324	Code for letter T
02 040	000	*End of command word marker*
02 041	001	Pg where COUNT rtn starts
02 042	300	Loc on pg where COUNT starts
02 043	305	Code for letter E
02 044	322	Code for letter R
02 045	301	Code for letter A
02 046	323	Code for letter S
02 047	305	Code for letter E
02 050	000	*End of command word marker*
02 051	001	Pg where ERASE starts
02 052	340	Loc on page where ERASE starts
02 053	000	**End of table marker**

FREE-FORMAT CONTROL TABLE - VERSION NO. 2

02 000	307	Code for letter G
02 001	317	Code for letter O
02 002	001	Page where GO routine starts
02 003	100	Loc on pg where GO starts
02 004	314	Code for letter L
02 005	311	Code for letter I
02 006	323	Code for letter S
02 007	324	Code for letter T
02 010	001	Pg where LIST routine starts
02 011	140	Loc on pg where LIST starts
02 012	315	Code for letter M
02 013	305	Code for letter E
02 014	304	Codr for letter D
02 015	311	Code for letter I
02 016	301	Code for letter A
02 017	316	Code for letter N
02 020	001	Pg where MEDIAN rtn starts
02 021	200	Loc on pg for MEDIAN
02 022	301	Code for letter A
02 023	326	Code for letter V
02 024	307	Code for letter G
02 025	001	Pg where AVG routine starts
02 026	240	Loc on page where AVG starts
02 027	303	Code for letter C
02 030	317	Code for letter O
02 031	325	Code for letter U
02 032	316	Code for letter N
02 033	324	Code for letter T

02 034	001	Pg where COUNT rtn starts
02 035	300	Loc on page where COUNT starts
02 036	305	Code for letter E
02 037	322	Code for letter R
02 040	301	Code for letter A
02 041	323	Code for letter S
02 042	305	Code for letter E
02 043	001	Pg where ERASE starts
02 044	340	Loc on page where ERASE starts
02 045	000	**End of table marker**

The reader may immediately notice that both of the free-format organizations take less memory storage for the table itself than the fixed-format arrangement. This is generally the case when there are large variations in the length of the data strings (number of memory words to a FIELD, such as the control words in the tables) that are held in the table. For fixed-format tables, each BLOCK (in the example being discussed a BLOCK would be 8 memory words) must be long enough to contain the largest possible field that could be encountered in the application. (In the present illustration, the fields in a block would be the control word field, and the address field. The largest control word field requires 6 memory words. All the address fields require 2 words. Thus each block must have 8 memory locations available.) Note that a fixed-format table may not require more room than a free-format table of the type shown in version no. 1 if there is not a large variation in the length of data within fields. For instance, had all of the control words been selected to be 5 and 6 letters in length, then version no. 1 would have actually required more memory space for the table than the fixed-format configuration.

However, the amount of memory space occupied by the table itself is not the only programming point to be considered when choosing the table format to be used in a particular program. One must also look at some other parameters that will have an effect on the total size of the program. One subtle parameter, for instance, is how will the inputted character string for a control word be delimited. Suppose, for example, that a control word character string is inputted via an ASCII keyboard subroutine and stored in a small buffer area in memory. One may assume that the actual input string was delimited (ended) by a special character such as a carriage return. The carriage return would inform the input routine to cease accepting characters and return to the calling program. However, since the character string that was received must also be used by some other routine (when searching the control table for a match), and since the character string may vary in length, then some means must be provided for telling the table search routine just how many characters are in the particular string of characters stored in the buffer!

This can be done in several different ways. One way would be to have the carriage return code received by the ASCII input routine be stored as the last character in the character string buffer. The table search routine could use the CR symbol as a delimiter to signify the end of the character string. The character string buffer would then contain information stored as shown here:

ADDRESS LOCATION	CONTENTS
WORD NO. 1	Code for character No. 1
WORD NO. 2	Code for character No. 2

WORD NO. 3	Code for character No. 3
.	.
.	.
WORD NO. N	Code for character No. N
WORD NO. N + 1	Code for carriage-return

Note, then, that the character buffer would have to be a block of locations in memory long enough to hold (N + 1) characters where N is the maximum number of characters allowed in a control word.

A second way to delimit the character string in the buffer would be to set up a counter that increased in value each time a character was accepted into the buffer. The value in the counter could then be used by the table search routine to indicate how long the character string was.

Still another technique would be to utilize a buffer address pointer that would point to the actual address of the last character in the buffer.

The second and third schemes allow the character buffer to be just N characters in length (instead of N + 1). However, the savings in buffer space is hardly enough to be concerned with, particularly since some other location(s) would have to be set aside for storing the value of the counter or buffer address pointer.

The different methods are mentioned, however, to demonstrate the important fact that there is more than one way to approach the problem. The programmer must develop the practice of examining alternative ways. While the differences are often subtle, certain choices may be of particular value in certain applications.

An idea that should be mentioned at this point concerns the practice of trying to develop programs that are goof-proof, or human-engineered. The importance of this factor should not be overlooked. For those that do will often find themselves spending many hours reworking programs that have suddenly gone beserk while in operation. The ability to plan programs that take this important parameter into consideration generally distinguishes the novice from the experienced programmer. What is meant by human-engineering can be demonstrated by the following discussion.

Suppose, for the example being developed here, that the programmer elected to develop the character string input routine using scheme no. 1 presented above by setting aside a character buffer N + 1 words in length. (Which would be 7 in this case, as the maximum size of a control word in the example is 6 characters.) Now, a novice, or unwary beginner might proceed to develop the routine along the following lines.

```
     MNEMONIC                       COMMENTS

 INCTRL, LHI XXX        Set page addr of start of char buffer
         LLI YYY        Set loc on page of start of char buffer
INCHAR,  CAL INPUT      Get a character from input subroutine
         LMA            Store in character string buffer
         CPI 215        See if character was a CR
         RTZ            Exit subroutine if CR
         CAL ADV        Advance buffer pointer
         JMP INCHAR     Loop to get next character
```

An experienced programmer would more likely have the subroutine appear like:

```
          MNEMONIC                          COMMENTS

INCTRL,   LHI XXX              Set page addr of start char buffer
          LLI YYY              Set loc on page of start of char buffer
          LBI 006              Set SAFETY counter
INCHAR,   CAL INPUT            Get a character from input subroutine
          CPI 215              See if character was a CR
          JFZ CHECK            If not CR go to safety check routine
          LMA                  If CR then store in buffer
          RET                  And exit subroutine
CHECK,    INB                  Exercise register B to set flags
          DCB                  For its original contents
          JTZ INCHAR           If B was 000, ignore present character
          DCB                  Otherwise, decrement value of B
          LMA                  Store character in buffer
          CAL ADV              Advance buffer pointer
          JMP INCHAR           And loop to get next character
```

What does the second subroutine do that the first did not? It guarantees that if somebody types in a character string more than six characters long, that the buffer will not expand beyond its intended length, possibly resulting in characters being loaded into portions of memory that contain program instructions or other data, the altering of which might eventually result in a program blow-up!

Still another way to delimit an input character buffer, and a method particularly suited to dealing with a fixed format table, is to clear out the buffer prior to the start of entering a character string by inserting all zero words into the buffer. When using this method, it is not desirable to insert a CR at the end of the string, but rather to simply allow the presence of a zero word denote the end of the character string.

Once the input character buffer has received a character string and a method of delimiting the string been selected, one may proceed to develop methods to search the control table for a control word that matches the character string in the buffer. The search routine will reflect the method used to organize the table, as well as the delimiting format used in the character string buffer. The various ramifications of what is meant by this can perhaps best be clarified by considering a few programming examples.

Examine the following portion of a search routine designed to look for a match between the characters in a buffer (terminated by a zero word) and the characters contained in the control word fields of the blocks making up the table.

```
          MNEMONIC                          COMMENTS

SEARCH,   LDI 002              Set pointers to starting addr of table
          LEI 000              Set pointers to starting addr of table
INITBF,   LHI XXX              Set pointers to start of char buffer
          LLI YYY              Set pointers to start of char buffer
```

```
             LBI 006        Set control word field size counter
CMATCH,      LAM            Get char fm buffer (form char match loop)
             CAL ADV        Subroutine to advance buffer pointer
             CAL SWITCH     Exchange buffer pntr for table pointer
             CPM            See if have a match condition
             JFZ NXWORD     If no match, go to next block in table
             DCC            If match, decr field size counter
          ** JTZ FOUNDW     All chars in field matched if cntr = 0
             CAL ADV        Char match but not finished, adv pntr
             CAL SWITCH     Exchange table pntr for buffer pointer
             JMP CMATCH     Loop to see if next character matches
NXWORD,      DCB            Decr field size cntr to find end of
             JTZ SETNXW     Current control word field, JMP when find
             CAL ADV        Otherwise advance table pointer
             JMP NXWORD     And loop to look for end of CW field
SETNXW,      CAL ADV        At end of control word field need to
             CAL ADV        Advance pntr over the address field
             CAL ADV        To the start of next control word field
         *** CAL SWITCH     And then exchange table for buffer pntr
             JMP INITBF     And form loop to check next block in table
```

Remember, the above routine assumes that the input character buffer is cleared before a new input character string is accepted. Thus, the input buffer would contain zeros in the locations from N + 1 to the end of the buffer (where N is the last character of the input string). If, for example, the input buffer contained the following:

WORD NO.	CONTENTS
1	Code for G
2	Code for O
3	000
4	000
5	000
6	000

then the routine just presented would find a match in the first block of the fixed format table described several pages earlier. When the match with the control word in the table was found, the routine would jump to the as yet undefined FOUNDW routine to extract the address of the GO routine from the table. However, had the input character contained:

WORD NO.	CONTENTS
1	Code for A
2	Code for V
3	Code for G
4	000
5	000
6	000

then the routine would fail to find a match in the first control word field. When the match failed it would jump to the NXWORD portion of the program to advance the table pointer to the start of the next control word field in the table. Then jump back to the INITBF portion to initialize the character buffer pointer and proceed to look for a match in the next block of the table. This loop would continue until the matching control word AVG was found about halfway down the table.

Had some smart-aleck operator keyed in the following to the input character buffer:

WORD NO.	CONTENTS
1	Code for S

```
2          Code for I
3          Code for L
4          Code for L
5          Code for Y
6            000
```

then the program would eventually bomb! Reason? (Here comes human engineering again!) Simply that the above routine has no way of determining where the end of the table exists in memory. The handling of that problem will be discussed shortly after some more examples related to the current topic have been presented. The reader should note here that the *** mark near the end of the routine denotes a point where an end of table test might be inserted in the above routine.

It is desirable at this point to illustrate several other search routines to demonstrate how they are affected by the table organization and the method used to delimit the input character buffer. Suppose one is still using the fixed-format table, but instead of clearing out the input buffer before accepting a new character string (so that it is delimited by locations containing zeros), one uses an input routine that delimits the buffer by using a CR symbol. The routine to look for a match between the contents of the buffer and a control word in the table might appear as follows.

```
 SEARCH,   LDI 002       Set pointer to starting addr of table
           LEI 000       Set pointer to starting addr of table
 INITBF,   LHI XXX       Set pointers to start of char buffer
           LLI YYY       Set pointers to start of char buffer
           LBI 006       Set control word field size counter
 CMATCH,   LAM           Get char fm buffer (form char match loop)
           CPI 215       See if symbol for CR
           JTZ LCHAR     If so, go to last character routine
           CAL ADV       Otherwise, advance buffer pointer
           CAL SWITCH    Exchange buffer pntr for table pointer
           CPM           See if have match condx in table
           JFZ NXWORD    If no match, go to next block in table
           CAL ADV       If match, advance table pointer
           CAL SWITCH    Exchange table pointer for buffer pntr
           DCB           Decrement counter value (for nxword rtn)
           JMP CMATCH    Loop to see if next character matches
  LCHAR,   XRA           If CR in buffer, clear accumulator
           CAL SWITCH    Exchange buffer pointer for table pntr
           CPM           And see if have 000 code in table
      **   JTZ FOUNDW    If so, all chars in field matched
           INB           If not, see if counter is at 000
           DCB           Indicating max control word field
      **   JTZ FOUNDW    Encountered so have control word match
 NXWORD,   DCB           If not, decr field size counter
           JTZ SETNXW    If cntr = 0, at end of control word fld
           CAL ADV       If not, advance table pointer
           JMP NXWORD    And loop to look for end of field
 SETNXW,   CAL ADV       At end of control word field need to
           CAL ADV       Advance pntr over the address field
           CAL ADV       To the start of next control word field
           CAL SWITCH    And then exchange table for buffer pntr
           JMP INITBF    And form loop to check next block in tbl
```

The above routine is a bit more complicated than the previous one. This is because one must still keep track of the number of characters that have been examined within a control word field in the table section (for use by the NXWORD routine), and also make an additional test for the end of the character string in the input buffer which is signified by the code for a carriage-return. It is assumed in the above routine that the routine that accepts a character string into the input buffer limits the string to a maximum of six characters. Note that one must also make special provisions for the case when the character string is six characters in length by testing the counter in the LCHAR portion of the above routine.

The combination of using a CR terminated buffer and a free-format table (such as the free-format version No. 1 illustrated earlier) is less complicated to search because one can drop the maintenance of the table control word field counter. Instead, one may test for the end of buffer marker (CR) and use the end of field marker (000) in the table when a match fails and it is necessary to advance to the next control word in the table. This search routine is illustrated next.

```
 SEARCH,  LDI 002       Set pointer to starting addr of table
          LEI 000       Set pointer to starting addr of table
 INITBF,  LHI XXX       Set pointer to start of char buffer
          LLI YYY       Set pointer to start of char buffer
 CMATCH,  LAM           Get char fm buffer (form char match loop)
          CPI 215       See if symbol for CR
          JTZ LCHAR     If so, go to last character routine
          CAL ADV       Advance buffer pointer
          CAL SWITCH    Exchange buffer pntr for table pointer
          CPM           See if have match condition in table
          JFZ NXWORD    If not, go to next block in table
          CAL ADV       If yes, advance table pointer
          CAL SWITCH    Exchange table pntr for buffer pointer
          JMP CMATCH    Loop to test next character
  LCHAR,  XRA           Clear accumulator if have CR in buffer
          CAL SWITCH    Exchange buffer pointer for table pntr
          CPM           See if also have end of field marker
     **   JTZ FOUNDW    If so, have found matching control word
 NXWORD,  LAM           If not, see if have end of field marker
          NDA           ***Trick to set flags after a load op***
          JTZ SETNXW    Found marker, go to next block
          CAL ADV       Marker not found, advance table pointer
          JMP NXWORD    And continue looking for marker
 SETNXW,  CAL ADV       After marker found, advance table pntr
          CAL ADV       Over the address field to the start
          CAL ADV       Of the next control word field
    ***   CAL SWITCH    Exchange table pntr for buffer pointer
          JMP INITBF    And form loop to check next block in tbl
```

At first glance, developing a search routine for the fixed-format table version No. 2, would appear rather difficult because there is no apparent end of control word field marker! However, that table was organized to take advantage of a particular fact that the developer was aware of that would enable the first part of the address field to be used as an end of control word field marker. This fact is that all of the character codes that might be used in the control word field (which consists of ASCII formatted symbols) have a '1' bit in one or both of the two most significant bits within a memory word that contains the character. Additionally, it is known that the maximum page address that can be utilized in a typical 8008 system is 077 (octal) which means that a memory word containing a memory page address cannot have a '1' condition in either one of the two most significant bits of the memory word that holds the page address! Thus, by making a simple test, using a masking operation described earlier in this section, a routine can be developed that will safely utilize the page address part of the address field to serve as an end of a control word field! Thus, to search version No. 2 of the free-format table, one could replace the routines LCHAR and NXWORD used above with the following substitute:

```
LCHAR,   CAL SWITCH    Exchange buffer pointer for table pntr
         LAM           Test for end of control field
         NDI 300       By seeing if two MSB's are both 0
         JTZ FOUNDW    If so, have found matching control word
NXWORD,  LAM           Test for end of control field
         NDI 300       By seeing if two MSB's are both 0
         JTZ SETNXW    If so, have marker, go to next block
         CAL ADV       Otherwise advance table pointer
         JMP NXWORD    And continue looking
```

As mentioned earlier, some means of determining when the entire table has been searched in the event a non-existent term is placed in the input buffer must be incorporated in the search routine. Again, this task can be accomplished in several different ways. One way would be to set a counter at the start of the search routine that contained the total number of blocks in the table and decrement it each time a block was checked. The counter could be tested for a zero condition to signify that the table had been searched. Another way to accomplish the objective would be to test the value of the table pointer to see if it had reached a specific value which would denote the end of the table. These two methods have several drawbacks. One is that the counter method would require storage space. A CPU register could be used, but more than likely one would have to resort to maintaining a counter in a memory location in order to conserve CPU registers. This would require a somewhat more lengthy routine to handle the updating and testing of the counter. Testing to see if the table pointer address had reached a certain value could be done with an immediate type comparison thus avoiding the maintenance of a storage location. But, the method (along with the counter method) is more cumbersome if the programmer decides to expand the size of the table at some future time. This is because the program would have to be modified at two different points, the table itself, and the portion of the routine that signifies the end of the table, either the counter value, or the address pointer value.

A method that is generally more convenient is to place a zero word at the end of the table as was shown for the example tables. Then, at the start of each new block, the search routine can conduct a simple test to see if a zero word is present indicating the end of the table. (Naturally, in special cases where, for instance, a data block might contain a zero word at the first location in a block, the method would not be appropriate and one could resort to one of the above techniques.) The method of using a zero word also makes it easy to expand the size of the table without having to modify any part of the search routine. More blocks can simply be added (replacing the former zero word) and a new zero word added after the additional blocks. The search routine, using the algorithm presented below, would then automatically be able to find the new ending point of the table. The following instructions could simply be inserted at the point indicated by the three asterisks in the search routines presented earlier.

```
        LAM             Fetch first character in new block
        NDA             ***Trick to set flags after load op***
        JTZ NOSUCH      If zero, end of table, no match found
```

The routine NOSUCH referred to by the end of table test might be a small routine to display a message to the operator indicating that there was no such command in the table. Or, the JTZ instruction might be replaced by an RTZ instruction that would return the program to the calling routine. The calling routine could simply direct the program back to the routine which fetches a new string of characters into the input buffer.

One other portion of the search routine that has not been touched upon is what the program would do once a match was found between the characters in the input buffer, and a control word field in the table. This portion of the routine was referred to as FOUNDW in the previous examples. FOUNDW would simply be a routine that would advance the table pointer to the end of the current control word field (where the match occurred. Then extract the address from the address field to enable the program to jump to the location given by the address and proceed to perform a specific function. The routine FOUNDW as given in the example that follows, contains an intrigueing portion that illustrates one of the powerful aspects about a computer. That is, a program may be designed to alter the execution of the program itself! This is done in the execution of the FOUNDW routine. When the program extracts the address from the table, it inserts it in a portion of the program for the address portion of a jump instruction which the program then proceeds to execute! Care must be taken when developing such a program to ensure that exactly the right locations are modified by the program. This will be apparent after examination of the following routine.

```
FOUNDW,  INB             Check to see if the field counter is 000
         DCB             Indicating end of the control word field
FNDEND,  JTZ SETJMP      If 0, set up the jump address
         CAL ADV         Otherwise advance table pointer
         DCB             Decrement field counter
         JMP FNDEND      And keep looking for end of field
SETJMP,  CAL ADV         Advance pointer to 1st part (page) of address
         LDM             And extract page address & store temp
         CAL ADV         Now advance pointer to location on page address
         LEM             And store it temporarily
         LHI MMM         Now set memory pointer (H & L) to point to the
         LLI NNN         2nd byte of the jump instr. coming up
         LME             Put the LOW order address in byte 2
         INL             Advance the memory pointer
         LMD             And the PAGE address in byte 3 of the JUMP
         JMP NNNMMM      Now jump to the addr just loaded into
     NNN AAA             These two (LOW address)
     MMM BBB             Bytes (PAGE address)
```

The above FOUNDW routine was for the case where the table was in the fixed-format organization and a counter was used to find the end of the control word field. Had the free-format table been used, then the beginning portion of FOUNDW would be appropriately modified to find the end of the control field. This could be done using the techniques illustrated in the NXWORD portion of the previously illustrated routines for that type of table.

The discussion of handling tables has extended over quite a few pages of text. A variety of routines have been presented showing various parts of the process. It might be beneficial to the reader to present a nicely packaged summary by presenting two table search routines. One using the fixed-format table coupled with an input character string buffer (that is cleared prior to accepting a new character string), the other using a free-format table (version No. 2) coupled with an input buffer that is delimited by a carriage return. (The actual routine that accepts characters from an I/O device will simply be noted as a subroutine call in the following examples. That routine would be a function of the I/O device used.)

```
                              Main program calling sequence
NEXCMD,   CAL CLEARB    Clear the input character string buffer
          CAL INCTRL    Fetch the command string from input device
          CAL SEARCH    Search table & perform command inputted
          JMP NEXCMD    Repeat loop for next command by operator


                        Clear input buffer subroutine
CLEARB,   LHI 003       Set page pointer to start of buffer
          LLI 372       Assumed to be at location 372 of page 003
          LBI 006       Set clearing counter
          XRA           Clear the accumulator
CLEARN,   LMA           Put 000 into buffer position
          INL           Advance buffer pointer
          DCB           Decrement counter
          JFZ CLEARN    If not through, put 000 in next location
          RET           When through return to calling routine


                        Fetch input command string
INCTRL,   LHI 003       Set page address of start of character buffer
          LLI 372       Set location on page of start of character buffer
          LBI 006       Set counter for maximum size of buffer
INCHAR,   CAL INPUT     Call subroutine to input character from I/O
          CPI 215       See if character was a CR
          RTZ           If so, make no entry
CHECK,    INB           Exercise register B (Counter) to set flags
          DCB           According to original contents
          JTZ INCHAR    Ignore new character if counter was 000
          DCB           Otherwise decrement value of counter
          LMA           And store character in buffer
          CAL ADV       Advance buffer pointer
          JMP INCHAR    And loop to fetch next character from I/O
```

```
                           Table search routine - compares character
                           String in input buffer against entries in
                           The control word fields of fixed-format
                           Table (six locations in the field)

SEARCH,   LDI 002          Set pointers to starting address of table
          LEI 000          Set pointers to starting address of table
INITBF,   LHI 003          Set pointers to start of character buffer
          LLI 372          Set pointers to start of character buffer
          LBI 006          Set control word field size counter
CMATCH,   LAM              Get character from buffer (form char match loop)
          CAL ADV          Subroutine to advance buffer pointer
          CAL SWITCH       Exchange buffer pointer for table pointer
          CPM              See if have a character match condition
          JFZ NXWORD       If no match, go to next block in table
          DCB              If match, decrement field size counter
          JTZ FOUNDW       If counter = 0, all characters in field matched
          CAL ADV          Character match but not finished, advance pointer
          CAL SWITCH       Exchange table pointer for buffer pointer
          JMP CMATCH       Loop to see if next character matches
NXWORD,   DCB              Decrement field size counter to find end of
          JTZ SETNXW       Current control word field, jump when find
          CAL ADV          Otherwise advance table pointer
          JMP NXWORD       And loop to look for end of control word field
SETNXW,   CAL ADV          At end of control word field need to
          CAL ADV          Advance pointer over the address field
          CAL ADV          To the start of next control word field
          LAM              And then fetch 1st character in new block
          NDA              Set the flags after the load operation
          RTZ              Return if end of table (no match found)
          CAL SWITCH       Otherwise exchange table pointer for buffer
          JMP INITBF       And form loop to check next block in table
FOUNDW,   CAL ADV          Advance pointer to 1st part (page) of address
          LDM              And extract page address to store temp
          CAL ADV          Advance pointer to location on page address
          LEM              And store it temporarily
          LHI MMM          Now set memory pointer (H & L) to point to the
          LLI NNN          2nd byte of the jump instruction coming up
          LME              Put the  low order address in byte 2
          INL              Advance the memory pointer
          LMD              And the PAGE address in byte 3 of the JUMP
          JMP NNNMMM       Now jump to the address just loaded into
    NNN   AAA              These two (LOW address)
    MMM   BBB              Bytes (PAGE address)

                           At the conclusion of the routine that
                           The search routine jumps to when a
                           Match is found, a RET instruction
                           Should be executed to return the program
                           To the main calling routine
```

The subroutines SWITCH and ADV have been detailed earlier in this chapter, and are not repeated in the previous example.

The next example is for the case where the input buffer is delimited by a CR and a free-format table (version No. 2) is used.

```
                                  Main program calling sequence
NEXCMD,    CAL INCTRL             Fetch the command string from input device
           CAL SEARCH             Search table & perform command inputted
           JMP NEXCMD             Repeat loop for next command by operator

INCTRL,    LHI 003                Set page address of start of character buffer
           LLI 371                Set location on page of start of buffer (N + 1)
           LBI 006                Set counter for maximum number usable characters
INCHAR,    CAL INPUT              Call subroutine to input character from I/O
           CPI 215                See if character was a CR
           JFZ CHECK              If not, check for buffer overflow
           LMA                    If yes, store CR as last character in buffer
           RET                    And return to calling routine
CHECK,     INB                    Exercise register B (counter) to set flags
           DCB                    According to original contents
           JTZ INCHAR             Ignore new character if counter was 000
           DCB                    Otherwise decrement value of counter
           LMA                    And store character in buffer
           CAL ADV                Advance buffer pointer
           JMP INCHAR             And loop to fetch next character from I/O

                                  Table search routine
SEARCH,    LDI 002                Set pointers to starting address of table
           LEI 000                Set pointers to starting address of table
INITBF,    LHI 003                Set pointers to start of character buffer
           LLI 371                Set pointers to start of character buffer
CMATCH,    LAM                    Get character from buffer (form char match loop)
           CPI 215                See if symbol for CR
           JTZ LCHAR              If so, go to last character routine
           CAL ADV                Otherwise, advance buffer pointer
           CAL SWITCH             Exchange buffer pointer for table pointer
           CPM                    See if have match condition in table
           JFZ NXWORD             If not, go to next block in table
           CAL ADV                If yes, advance table pointer
           CAL SWITCH             Exchange table pointer for buffer pointer
           JMP CMATCH             Loop to test next character
LCHAR,     CAL SWITCH             Exchange buffer pointer for table pointer
           LAM                    Test for end of control field
           NDI 300                By seeing if two MSB's are both 0
           JTZ FOUNDW             If so, have found matching control word
NXWORD,    LAM                    Test for end of control field
           NDI 300                By seeing if two MSB's are both 0
           JTZ SETNXW             If so, have marker, go to next block
           CAL ADV                Otherwise, advance table pointer
           JMP NXWORD             And continue looking
```

```
SETNXW,  CAL ADV       At end of control word field need to
         CAL ADV       Advance pointer over the address field
         LAM           And then fetch 1st character in new block
         NDA           Set the flags after the load operation
         RTZ           Return if end of table (no match found)
         CAL SWITCH    Otherwise, exchange table pointer for buffer
         JMP INITBF    And form loop to check next block in table
FOUNDW,  LDM           Extract page address and store temp
         CAL ADV       Advance table pointer
         LEM           Store location on page temporarily
         LHI MMM       Now set memory pointer (H & L) to point to the
         LLI NNN       2nd byte of the JUMP instruction coming up
         LME           Put the low order address in byte 2
         INL           Advance the memory pointer
         LMD           And the page address in byte 3 of the JUMP
         JMP NNNMMM    Now JUMP to the address just loaded into
     NNN AAA           These two (LOW address)
     MMM BBB           Bytes (PAGE address)

                       After processing command, return to main routine
```

SORTING OPERATIONS

Another particularly powerful capability of a mini-computer is its ability to rapidly manipulate and organize information. A typical operation is to sort data into some desired form, such as to arrange a list of names into alphabetical order. Or, possibly, to arrange a list of addresses by zip code zone numbers.

The key ingredient in developing a program to perform sorting operations is to plan the organization of the storage of the data in memory so that the operating portion of the program is relatively simple. A simple technique involves justifying the data into fields so that simple comparing algorithms may be utilized.

As an example of a sorting program, assume one had a list of names that one wished to have the computer place in alphabetical order. A hypothetical list might consist of the following names:

JONES, R. M.
SMITH, C.
WILLIAMS, P. K.
DAVIS, Z. T.
THOMPSON, A. R.
THOMAS, F.
ALLISON, A. B.
SMITH, T. P.

It may be supposed that the names will be inputted and stored in the computer in the order shown above. The first objective of the program would be to have the incoming names stored in a manner that would be easy for the sort routine to operate on. A good technique to use would be to set up fields for the information being stored. In this case, one would want to set up three fields. One for the last name, one for the first initial, and one for the middle initial. The size of each field would need to be determined. For the example list shown above, the longest last name encountered has eight letters. Thus, the field for the last names must have space for at least eight characters since one computer word in memory will store the code for one letter in the name. However, in order to make the program flexible, one could select a longer field length to allow longer names to be stored. For illustrative purposes, a last name field of 14 (decimal) units will be planned. (Note that this a purely arbitrary selection.) The field length for each initial would only have to be 1 memory word. Thus, the total length of the three fields making up a block would be 16 (decimal) or 20 (octal) memory words. Note that in selecting the field lengths for this example, space was not included for the comma (,) sign after the last name, or the periods (.) after each initial. This is because since these signs are repititious, one can save valuable memory space by deleting these marks during the input operation. Then simply add them back in at the appropriate point when the data is displayed by the output device.

The input routine would need to always start inserting characters at the beginning of a field. Then insert spaces or some special code (such as a 000 word) in all of the unused memory words in a field so that the names could be considered as being left justified in each field. The reason for this will be made clear shortly.

The following routine might be used to accept information from a keyboard device and store the names in memory in the desired format.

```
    ACCEPT,   LHI 004        Initialize names storage area pointer
              LLI 000        To start of storage area
    NOTFND,   LAM            Now fetch 1st location in a block
              NDA            Set flags after load operation
              JTZ FNDEND     And test for end of storage area
              LAI 020        If not end, then advance pointer
              ADL            To next block by adding 20 octal
              LLA            To memory pointer address & restore pointer
    CKPAGE,   CTZ INCRH      Advance page address of pointer if required
              LAI 010        Now text to see if still
              CPH            In storage area (pages 04 - 07 octal)
          *   JTZ TOMUCH     Optional display message if storage filled
              JMP NOTFND     Keep looking for end of storage area
    FNDEND,   LBI 016        Setup last names field counter
              CAL INPUT      And fetch a character from input routine
              CPI 252        Check for * code (finished indicator)
              JFZ NOTDON     Proceed if not * code
              XRA            If * code, then place a 000 word at
              LMA            Start of block as an ending marker
              RET            And exit subroutine
    NOTDON,   CPI 215        Test for carriage-return code
              JTZ FNDEND     And ignore if 1st character in field
              CPI 256        Test for period (.) code
              JTZ FNDEND     And ignore if 1st character in field
              CPI 254        Test for comma (,) code
              JTZ FNDEND     And ignore if 1st character in field
              LMA            If none of above, put character in field
              DCB            Decrement the field size counter
              INL            Advance the storage pointer
 ** NEXTIN,   CAL INPUT      And fetch the next character in last name
              CPI 215        Test for carriage-return
```

```
            JTZ HAVECR    Finished block if have CR here
            CPI 254       Test for comma
            JTZ HAVECM    Finished last name field if have comma
            LMA           Otherwise place character in last name field
            INL           Advance the storage pointer
            DCB           Decrement last names field size counter
            JTZ FULFLD    And see if field is filled
            JMP NEXTIN    If not, get next character in last name
   HAVECR,  XRA           If have CR put a 000 in memory words
            LMA           For rest of current block
            LAL           Fetch memory pointer to accumulator
            NDI 017       Mask off 4 most significant bits
            CPI 017       Test for end of block
            JTZ NEXBLK    Prepare for next block if done
            INL           Otherwise advance pointer
            JMP HAVECR    And continue putting 000 words in block
   HAVECM,  XRA           If have comma, put 000 words in rest
            LMA           Of last name field
            INL           Advance field pointer
            DCB           Decrement last names field counter
            JTZ FULFLD    Go process initials when done
            JMP HAVECM    Else continue to clear rest of field
   NEXBLK,  INL           Advance memory pointer to start of next block
            JMP CKPAGE    And prepare for next name entry
** FULFLD,  CAL INPUT     Get character for 1st initial of name
            CPI 254       Test for comma
            JTZ FULFLD    Ignore comma at this point
            CPI 215       Test for CR
            JFZ SAVIN1    If not CR, store character
            XRA           But, if CR, put in 000 word
            LMA           For both initial fields
            INL           By above instruction, then advance pointer
            JMP SAVIN2    And then following this jump command
   SAVIN1,  LMA           Store 1st initial in 1st initial field
            INL           Then advance storage pointer
** INITF2,  CAL INPUT     Look for 2nd initial
            CPI 256       Check for period
            JTZ INITF2    Ignore a period
            CPI 215       Test for CR
            JFZ SAVIN2    If not CR then store 2nd initial
            XRA           But if was CR, place 000 word in memory
   SAVIN2,  LMA           Store the character or 000 substitute
            INL           Advance pointer to new block
            CTZ INCRH     Advance page address of pointer if required
            LAI 010       Now test to see if still in
            CPH           Storage area (Pages 04 - 07)
            JTZ TOMUCH    Optional display message if storage filled
            JMP FNDEND    Go process next input
   INCRH,   INH           Subroutine to increment register H
            RET           And then return to calling routine
```

The above routine has a number of special factors included in it to illustrate considerations that programmers must learn to take into account when developing such programs. Some of these factors are pointed out in the following discussion of the above routine.

The first function the above routine performs is to look for the end of the name storage area. This is done by testing the first character in each block to see if it contains a 000 word. As shown later in the routine, a 000 word will be entered at that location whenever the operator has finished entering a series of names that will be sorted. It should be noted that whenever it is desired to initialize the name storage area so that it appears to the program that the storage area is empty, a subroutine that will place a 000 word at page 04, location 000 should be executed. (That simple subroutine is not shown above.) The above routine also makes a test each time the memory pointer is advanced to a new block, to determine whether the pointer is still in the alloted names storage area. For this example the storage area was planned to reside in locations from page 04 location 000 to page 07 location 377. Should the routine go beyond the designated storage area before an end of table marker is found, the routine would jump to a routine termed TOMUCH. TOMUCH might print out a message to the operator indicating that the storage area was already filled with names. (That routine is not included in the example above.) The reference to the routine TOMUCH is noted by an asterisk in the above program source listing.

When the routine has found the end of the names storage area, indicating where additional incoming names can be stored (provided the storage area has not been exhausted), the routine then proceeds to accept data from an input subroutine. The first character accepted at the start of a new name (block) is tested to see if it is a special code (an asterisk in this case) that the operator could use to signify to the program that all the desired names had been entered. If this code was received, then a 000 code would be placed in the first memory word for the block as the end of table marker mentioned above. The routine would then exit the above routine.

If the first character in a new block is not the special end code, a check is made to see if it is a carriage-return, comma, or period sign. Any one of those codes would be ignored as the first character in a block for the following reasons. The receipt of a carriage-return or comma would obviously be invalid at this point because no letters for a name have been entered. The acceptance of either of those operators would cause the last name to be completely filled with 000 words, including the first location. This action would result in an effective end of storage area marker being placed at the location of the current block. The receipt of a period sign would most likely be the period sign from the last initial field entered (which is to be ignored) and certainly would not be a valid letter for the beginning of a last name. The incorporation of these checks act as safeguards for human operator errors, and are another example of human engineering factors in the development of a program.

If the first character is not one of the above, it is stored in the first location in the last name field. After the first character has been stored, each character received from the input routine is tested to see if it is a carriage return or comma. If it is a comma, signifying the end of the last name field, any unfilled locations in the field are filled with zeros. The program then proceeds to the initial fields. However, if a carriage return is noted, the program fills the entire remainder of the current block, including the initial fields, with zero words. This is because a carriage return signifies the completion of a name entry. An additional safeguard is built into the routine in this section to prevent too many characters from being entered into the last name field. When the field has been filled, the pointer is not advanced until a carriage return or comma is received.

Once the last name field has been processed, the routine will accept characters as initials. However, it ignores the period signs after the initials. When an entire name has been processed, the program loops to accept another name block after checking to make sure the storage area is not filled. It then repeats the process described.

The above routine could be modified to include an operator convenience—the ability to erase a current entry if the operator made a mistake while typing in a name. This could be done by executing a routine immediately after the points designated in the program by a double asterisk (**). The routine could be used to check for a special erase code. If this code was detected, the program could reset the pointers to the start of the current name block, and allow re-entry of the name. Such a routine might be as shown here:

```
ERRORT,  CPI 377       Check for a rubout code
         JFZ AWAY      Exit routine if not a rubout
         LAL           If have a rubout then fetch pointer
         NDI 360       Remove 4 least significant bits
         LLA           And restore pointer to start of block
         JMP FNDEND    Jump to re-enter name
  AWAY,  ***           ***Next instruction in current sequence
```

While the previous routine seems a bit long at first glance, one must remember that it is doing quite a few functions, and is quite general purpose in over-all design. The program enables one to build up a list of names in a designated area of memory, place the data in formatted fields, check for selected operator errors, and bound or limit the storage area. The program, using the basic concepts presented, can be modified to serve as a basic structure for inputting a variety of types of data into justified fields. To provide a clear mental picture of how the list of names given several pages earlier would appear when inputted to memory using the program illustrated, a diagram showing memory locations and their contents is provided below. The diagram shows addresses (on page 04) with the contents of the memory location shown beneath it, followed by the alphabetical representation for the code where applicable.

```
ADDR: 000 001 002 003 004 005 006 007 010 011 012 013 014 015 016 017
CONT: 312 317 316 305 323 000 000 000 000 000 000 000 000 000 322 315
LETR:  J   O   N   E   S   -   -   -   -   -   -   -   -   -   R   M

ADDR: 020 021 022 023 024 025 026 027 030 031 032 033 034 035 036 037
CONT: 323 315 311 324 310 000 000 000 000 000 000 000 000 000 303 000
LETR:  S   M   I   T   H   -   -   -   -   -   -   -   -   -   C   -

ADDR: 040 041 042 043 044 045 046 047 050 051 052 053 054 055 056 057
CONT: 327 311 314 314 311 301 315 323 000 000 000 000 000 000 320 313
LETR:  W   I   L   L   I   A   M   S   -   -   -   -   -   -   P   K

ADDR: 060 061 062 063 064 065 066 067 070 071 072 073 074 075 076 077
CONT: 304 301 326 311 323 000 000 000 000 000 000 000 000 000 332 324
LETR:  D   A   V   I   S   -   -   -   -   -   -   -   -   -   Z   T
```

```
ADDR: 100 101 102 103 104 105 106 107 110 111 112 113 114 115 116 117
CONT: 324 310 317 315 320 323 317 316 000 000 000 000 000 000 301 322
LETR:  T   H   O   M   P   S   O   N   -   -   -   -   -   -   A   R

ADDR: 120 121 122 123 124 125 126 127 130 131 132 133 134 135 136 137
CONT: 324 310 317 315 301 323 000 000 000 000 000 000 000 000 306 000
LETR:  T   H   O   M   A   S   -   -   -   -   -   -   -   -   F   -

ADDR: 140 141 142 143 144 145 146 147 150 151 152 153 154 155 156 157
CONT: 301 314 314 311 323 317 316 000 000 000 000 000 000 000 301 302
LETR:  A   L   L   I   S   O   N   -   -   -   -   -   -   -   A   B

ADDR: 160 161 162 163 164 165 166 167 170 171 172 173 174 175 176 177
CONT: 323 315 311 324 310 000 000 000 000 000 000 000 000 000 324 320
LETR:  S   M   I   T   H   -   -   -   -   -   -   -   -   -   T   P

ADDR: 200 201 202 203 204 205 206 207 210 211 212 213 214 215 216 217
CONT: 000 *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
LETR:  -  ....DON'T CARE ABOUT MEMORY CONTENTS BEYOND HERE....
```

Once the data has been organized in a suitable manner in memory, one can proceed to develop a relatively simple sort routine to arrange the names in alphabetical order. The technique to be illustrated consists of comparing the letter, starting with the left-most position in a block (as seen in the memory diagram above) against the letter in the same position in the next block in memory. By letter what is actually meant is the ASCII code (in this example) for a letter. It so happens that the ASCII code is arranged such that the alphabet goes in ascending numerical order. The letter A is represented as 301, the letter B as 302, C as 303, and so forth on up to the letter Z which has an octal representation of 332. How convenient! This means that if the value in a memory word (representing a letter in ASCII format) is compared against another memory word containing an ASCII coded letter, that the lower valued entry contains a lower order letter in the alphabet.

With this information one may quickly discern that one can quite easily develop an algorithm to arrange names alphabetically. If the value of a memory location in the first position of say the first block (the Nth block) is compared against the value of the first position in the next block (N+1 block) and found to be greater in value, then the first (Nth) block has a name that is higher alphabetically than the name in the second (N+1) block. Thus one can immediately proceed to exchange the contents of the two blocks to arrange the names in ascending alphabetical order. If, however, the code in the first block is less in value than the second block, then the present order is correct, and the program can proceed to check the second block against the third one. If the letters in the first position checked are equal in value, then one cannot yet make a decision about the alphabetical order, but rather must go on to compare the values of the second letter within the two blocks!

To further complete the algorithm, one must also consider the possibility that when one exchanges the contents of blocks N and N + 1 that the new contents of N will now be of lesser order than that contained in block N - 1. Thus, whenever one performs an exchange of two blocks, one must have the program go back and do a comparison between the N and N-1 blocks. One can envision the algorithm as proceeding in a see-saw manner, comparing the Nth block against the N+1 block until an exchange is necessary. Then switching to compare between the Nth and N-1 block until an exchange is not necessary. At that point the process reverts back to comparing the Nth and N+1 blocks until another exchange is required. Looked at another way, the data blocks could be viewed as rippling upwards or downwards in memory as the process proceeds. Higher ordered names getting shoved to higher addressed blocks, lower ordered names being pushed to lower addressed blocks.

This type of algorithm is not the only way one could proceed to sort the data. There are other types of algorithms that can perform the same job, some of which are faster when large data bases are involved (but more complicated programming-wise). Such algorithms generally have considerable value on larger machines. However, the above algorithm is quite suitable for typical sorting jobs that a microcomputer might be called upon to perform. For those who might want to investigate other algorithms, they might consider the concept of having a program that immediately classifies a name into, say, the first, second, or third section of the alphabet.

A program for the ripple sorting algorithm discussed above is presented below.

```
SORT,     LHI 004        Initialize pointer to start
          LLI 000        Of names block storage area
INITBK,   LBI 020        Set block length counter
          LCM            Get 1st character from block N into C register
          LAL            Fetch N block pointer
          ADI 020        Advance pointer to block N + 1
          LLA            Restore pointer
          CPI 020        Check to see if going to new page
          CTC INCRH      Advance page pointer if required
          LAM            Get 1st character from block N+1 into accumulator
          NDA            Set flags after loading operation
          RTZ            End of storage - sort operations completed
          CPC            Compare N + 1 letter to N letter
          JTC XCHANG     N greater than N+1 so exchange block contents
          JTZ CKNEXT     N = N+1 so check next letter in block
          JMP FNDEND     N less than N+1 so order O.K., do next block
CKNEXT,   DCB            Decrement block length counter
          JFZ NOTFIN     Continue if not finished block
BACKER,   LAL            Fetch N+1 pointer to ACC
          NDI 360        Reset pointer to N block
          LLA            Restore pointer
          JMP INITBK     Go to compare next block
NOTFIN,   LAL            Fetch N+1 block pointer
          NDA            Clear the carry flag with this no-op
          SUI 017        Decrease pointer to N block
          LLA            Restore pointer
          CTC DECRH      If underflow then decrement page pointer
          LCM            Fetch character from N block to register C
          LAL            Fetch N block pointer
          ADI 020        Increase pointer to N+1 block
          LLA            Restore pointer
          CPI 020        Check to see if going to new page
          CTC INCRH      Advance page pointer if required
          LAM            Get character from N+1 block
```

```
            CPC            Compare N+1 letter to N letter
            JTC XCHANG     N greater than N+1 so exchange block contents
            JTZ CKNEXT     N = N+1 so check next letter in block
FINEND,     DCB            N less than N+1 so order O.K., do next block
            JTZ BACKER     At end of block N+1 reset pointer for N
            INL            Advance pointer
            JMP FINEND     And loop to look for end of block
XCHANG,     LAL            Fetch N+1 pointer
            NDI 360        Mask off LSB's to restore pointer
            LLA            To start of N+1 block
            LBI 020        Set block length counter
NOTYET,     LCM            Fetch N+1 into register C
            LAL            Fetch N+1 pointer to accumulator
            NDA            Clear the carry flag
            SUI 020        Decrease pointer to N block
            LLA            Restore pointer
            CTC DECRH      Decrement page pointer if required
            LDM            Fetch N into register D
            LMC            Place former N+1 into N
            LAL            Fetch N pointer to accumulator
            ADI 020        Increase pointer to N+1 block
            LLA            Restore pointer
            CPI 020        Check to see if going to new page
            CTC INCRH      Increment page pointer if required
            LMD            Place former N into N+1
            INL            Advance N+1 pointer
            DCB            Decrement block length counter
            JFZ NOTYET     Continue if not finished exchanging
            LAL            If finished exchanging, fetch N+1 pointer
            NDA            Clear carry flag
            SUI 060        Back pointer from N+1 to N-1 block
            LLA            Restore pointer
            CTC DECRH      Decrement page pointer if required
            LAH            Fetch current page
            CPI 003        Make sure still in storage area
            JFZ INITBK     Yes - do an effective N-1 to N test
            JMP SORT       Went back too far - go to starting block!
```

The INCRH referred to by the sort routine was presented earlier as part of the routine that accepted names into the storage area. The DECRH routine not shown should be a snap for anyone who has reached this point in the manual. (If it is not, for Heavens sake, go back and review!)

If one mentally proceeds through the sort routine while referring to the diagram given several pages earlier showing the names as originally stored in memory, one should be able to clearly discern the operation of the sort program. For example, for the first three names the program encounters in the original example setup, the program will only have to test the first letter in each block. When the name in the 4th block is examined, an exchange will have to be made with the name in the third block. Then the program will find when checking the N-1 block (which was the original second block) that the name

Davis, Z. T. has to be exchanged again. This will happen again until the name Davis, Z. T. arrives at the first block in the storage area. At this point the program goes back to checking against the N+1 block. The names would then appear in memory in the following order.

Block No. 1:	Davis, Z. T.
Block No. 2:	Jones, R. M.
Block No. 3:	Smith, C.
Block No. 4:	Williams, P. K.
Block No. 5:	Thompson, A. R.
Block No. 6:	Thomas, F.
Block No. 7:	Allison, A. B.
Block No. 8:	Smith, T. P.

Now the program would get down to block five before it found it necessary to exchange block five with block four. The next N-1 test would fail, however, and the program would proceed back up to block six. There it would find the name Thomas, F. and have to exchange it again with Thompson, A. R. At that point, the names storage area would appear as:

Block No. 1:	Davis, Z. T.
Block No. 2:	Jones, R. M.
Block No. 3:	Smith, C.
Block No. 4:	Thomas, F.
Block No. 5:	Thompson, A. R.
Block No. 6:	Williams, P. K.
Block No. 7:	Allison, A. B.
Block No. 8:	Smith, T. P.

At that point the program would get up to block number seven where it would find Allison, A. B. It would then have to exchange names all the way back down the line to get it into block number one. Finally, the program would find that Smith, T. P. had to be moved back ending up in block number five. All of the above would have happened in a mere fraction of a second as the CPU executed the instructions at micro-second speeds, resulting in the names organized in the following desired manner.

Block No. 1:	Allison, A. B.
Block No. 2:	Davis, Z. T.
Block No. 3:	Jones, R. M.
Block No. 4:	Smith, C.
Block No. 5:	Smith, T. P.
Block No. 6:	Thomas, F.
Block No. 7:	Thompson, A. R.
Block No. 8:	Williams, P. K.

Similar types of sorting or arranging operations can also be done with numbers in BCD or binary form, or with other types of data.

One could combine a control table, such as one of the types discussed earlier in this chapter, with the necessary input, formatting, and sort subroutines just discussed. Thus, one could make up a powerful, yet easy to use, operating package suited to the user's specific requirements.

By utilizing the concepts (as well as some of the specific routines) presented in this section, the reader should be able to see the way towards developing some sophisticated programs capable of performing functions tailored to the individual's own requirements.

For those interested in utilizing the mathematical capabilities of the digital computer (perhaps combining such operations with some of those already discussed) simply proceed on to study the next chapter.

The ability of a digital computer to handle mathematical operations combined with its ability to manipulate text gives the machine a unique combination of power that partially accounts for its growing popularity. Programming a computer using machine language to perform mathematical functions is perhaps a bit more complicated than having it perform routine text manipulations. But, it is not as difficult as some people tend to think before being introduced to the subject. Like most other programming tasks, the key to success is organization of the program into small routines that can be built upon to form more powerful functions.

The instruction set of the '8008' and similar CPU's contain a number of primary mathematical instructions that are the basis for developing mathematical programs. The groups used most often include the ADDITION, SUBTRACTION and ROTATE instructions. (Do you recall that rotating a binary number to the left effectively doubles, or multiplies the original value by two, and rotating it to the right essentially divides the original value in half?)

Dealing with numbers of small magnitude using a microprocessor is simplicity itself. For instance, if one wanted to add the numbers '2' and '7,' one could load one number into register B in the CPU and load the other into the accumulator. The simple directive:

```
ADB
```

would result in the value '011' (octal) being left in the accumulator. Subtraction is just as easy. If one placed '7' in the accumulator and '2' in register B and executed a:

```
SUB
```

instruction the value '5' would be left in the accumulator.

Multiplication of small numbers may be readily accomplished using a simple algorithm. That is to add the multiplicand to itself the number of times dictated by the multiplier. Suppose one desired to have the computer multiply '2' times '3.' Placing the value '2' in register B and '3' in register C and executing the following instruction sequence:

```
  START,  XRA
 MULTIP,  ADB
          DCC
          JFZ MULTIP
   STOP,  HLT
```

would result in the value '6' ending up in the accumulator. As shall be discussed further on, the above algorithm is not very efficient when the numbers become large. More efficient multiplication algorithms are based on rotate operations which effectively multiply a number by a power of two. For instance, multiplying a number by '32' (decimal) would require 32 loops through the above routine. It would only require five rotate operations! However, the above routine illustrates how a number can be multiplied even though the computer does not have a specific "multiply" instruction.

One may also divide small valued numbers that have integer results using a similarly simple algorithm that subtracts instead of adds. For instance, a reverse of the previous example would be to divide the number '6' by the value '2.' The subtraction algorithm could appear as:

```
  START,  LCI 000
 DIVIDE,  NDA
          JTZ STOP
          SUB
          INC
          JMP DIVIDE
   STOP,  HLT
```

In the algorithm just presented, the routine starts with the number '6' in the accumulator. The divisor is in register B. Register C is used as a counter to count how many times the value in B can be subtracted before the contents of the accumulator reaches zero. As pointed out previously, the algorithm only works properly if the result is an integer value. Division is perhaps the most difficult basic mathematical function to perform on a digital computer because of mathematical peculiarities (involving the manipulation of fractional values). However, as will be illustrated later, there are ways around the above limitation. The above illustration is merely to give the novice encouragement. It illustrates that such operations are possible even though a specific divide command is not part of a typical microprocessor's instruction set!

The discussion so far has been limited to numbers of relatively small magnitude. Specifically, numbers small enough to be contained in a single eight bit binary register or memory location in a microprocessor. Many user's who want to use the digital computer to perform mathematical operations seem to get stumped when first coming across a requirement to manipulate numbers that are too large in magnitude to fit in one memory word or CPU register. With an '8008' based machine, and indeed most microcomputers available at present, such a requirement typically arrives shortly after one has started operating their machine! The reason is simply that the largest valued number that can be placed in an 'N-bit' register is the value (2**N)-1. Since most microprocessors use but eight bits in a word, the largest number that can be represented in a single word if all the bits are used is a mere 255 (decimal). If one desires to maintain the "sign" (whether the value is greater or less than zero) and uses one bit in a register for that purpose, then the largest number that can be represented in a single word is a paltry 127 (decimal). That is hardly enough to bother using a computer to manipulate!

But, the secret to rapidly increasing the magnitudes of the numbers that can be handled by a digital computer is held in that formula just presented; (2**N)-1. That formula states that the size of the number that can be stored in a binary register doubles for every bit added to the register. Thus, if one were to store a number using the available bits in two registers or memory words in an 8-bit-per-word system, one would be able to represent numbers as large as (2**16)-1 or 65,535 (decimal). If one of those 16 bits was reserved for a sign indicator, the magnitude would be limited to (2**15)-1 which is 32,767 (decimal). That is certainly a lot more than the value of 127 that can be held in just one word! But, why stop at holding a number in two words? There is no need to, one may keep adding words to build up as many bits as desired. Three words of eight bits, leaving one bit out for a sign indicator, would leave 23 bits. That number of bits could represent numbers as large as (2**23)-1 which is about 8,388,607 (decimal). Four words would allow representing a signed number up to (2**31)-1 which is roughly 1,107,483,647! One could add still more words if required.. Generally, however, one selects the number of significant digits that will be important in the calculations that are to be performed and uses enough words to ensure that the "precision" or number of significant digits required for the operations, can be represented in the total number of bits available within the grouped words. The use of more than one computer word or normal sized register to store and manipulate numbers as though they were in one large continuous register is commonly referred to as "multiple-precision" arithmetic. One often hears computer technologists speaking of "double-precision" or "triple-precision" arithmetic. This simply means that the machine is using techniques (generally programming techniques) that enable it to handle numbers stored in two or three registers as though they were one number in a very large register.

The '8008' CPU is capable of multiple-precision arithmetic. In fact it does it quite nicely because the designers of the CPU took particular care to include some special instructions for just such operations. (Such as the ADD and SUBTRACT with CARRY instructions.) Multiple-precision arithmetic is not difficult. It takes a little extra consideration when organizing a program to handle and store numbers that are contained in multiple words in memory. But, with the use of effective subroutines (and "chaining") the task can be handled with relative ease.

In order to effectively deal with multiple-precision arithmetic one must establish a convention for storing the sections of one large number in several locations. For the purposes of the current discussion, it will be assumed that triple-precision arithmetic is to be performed. Numbers will be stored in three consecutive memory locations according to the following arrangement.

Location N	= Least significant 8 bits
Location N+1	= Next significant 8 bits
Location N+2	= Most significant 7 + sign bit

Thus, the three words in memory could be mentally viewed as being one continuous large register containing 23 binary bits plus a sign bit as illustrated in the diagram below.

```
   LOC N+2      LOC N+1      LOC N
  sx xxx xxx   xx xxx xxx   xx xxx xxx
    MSB's        NSB's        LSB's
```

Of course, one could reverse the above sequence, and store the least significant bits in memory location 'N,' the next group in 'N+1' and the most significant bits plus sign bit in memory location 'N+2.' It makes little difference as long as one remains consistent within a program. However, the convention illustrated will be the one used for the discussion in this section.

Also, as has been pointed out, it is not necessary to limit the storage to just three words. Additional words may be used if additional precision is required. For most of the discussion in this chapter, three words will be used for storing numbers. Using three words in the above fashion will allow numbers up to a value of 8,388,647 to be stored. This means that six to seven significant digits (decimal) can be maintained in calculations.

The first multiple-precision routine to be illustrated will be an addition routine that will add together two multiple-precision numbers and leave the result in the location formerly occupied by one of the numbers. The routine to be presented has been developed as a general purpose routine in that, by properly setting up memory address pointers and loading a CPU register with a precision value prior to calling the routine, the same routine may be used to handle multiple-precision addition of numbers varying in length from '1' to 'N' registers. (As long as the registers containing a number are in consecutive order in memory, and with the restriction that all the registers containing a number are on one page. That limits 'N' to 255 (decimal) words, which is a limitation few programmers will find cumbersome!)

The key element in the addition routine to be illustrated is the use of the ACM add with carry instruction. The essential difference between an add with carry (ACM) instruction and an ADM (add without carry) command is as follows.

An ADM instruction simply adds the contents of the accumulator and the contents of the memory location pointed to by the H and L registers. During the addition process, the status of the carry flag is ignored. However, if at the end of the process, an overflow has occured, the carry flag will be set to a logic one condition. For example, adding the following binary numbers would yield:

```
             10 101 010
             01 010 101
             ----------
CARRY = 0 :  11 111 111
```

An example illustrating a carry occuring is shown next.

```
             11 111 111
             00 000 001
             -------------------------
CARRY = 1 :  00 000 000
```

Remember in the above examples that the CARRY FLAG is only affected by an overflow condition after the operation has occured. The original condition of the carry flag will have no effect on the final results of the calculation.

An ACM command, on the other hand, examines the contents of the CARRY FLAG prior to the addition operation and considers it as an operator on the least significant bit position. At the end of the addition process, the carry flag is again set or cleared depending on whether or not an overflow occured. (As in the case for the ADM instruction discussed above.) For example, adding the following binary numbers yields results that differ depending on the INITIAL status of the carry flag.

```
CASE 1A:                     0 : 'C' FLAG initially cleared
             10 101 010
             01 010 101
             -------------------------
CARRY = 0 :  11 111 111

CASE 1B:                     1 : 'C' FLAG initially set
             10 101 010
             01 010 101
             -------------------------
CARRY = 1 :  00 000 000

CASE 2A:                     0 : 'C' FLAG initially cleared
             11 111 111
             00 000 001
             -------------------------
CARRY = 1 :  00 000 000

CASE 2B:                     1 : 'C' FLAG initially set
             11 111 111
             00 000 001
             -------------------------
CARRY = 1 :  00 000 001
```

In summary, one can see that an ACM type of instruction makes multiple-precision addition extremely easy. This is because the carry bit acts as a link between any carry from the most significant bit of one addition operation into the least significant bit of the next addition operation. This allows one to proceed just as though the addition operation was being performed in one long register instead of several short registers. To discern this clearly, examine the example provided next which first illustrates an addition operation being performed in a hypothetical 16 (decimal) bit register, then shows the same result when two ACM operations are performed on two eight bit registers "linked" by the special capabilities of the ACM instruction.

ADDITION IN HYPOTHETICAL 16 BIT REGISTER

```
                 11 111 111  10 101 010
                 00 000 000  11 010 101
                 ----------------------
     CARRY = 1 : 00 000 000  01 111 111
```

SAME OPERATION USING ACM INSTRUCTION & TWO 8 BIT REGISTERS

```
                                      0  : 'C' FLAG assumed cleared
FIRST ACM OPERATION:         10 101 010
                             11 010 101
                             ----------
                 CARRY = 1 : 01 111 111  : LSB's in memory location N

                                      1  : 'C' FLAG set by above add
SECOND ACM OPERATION:        11 111 111
                             00 000 000
                             ----------
                 CARRY = 1 : 00 000 000  : MSB's in location N+1
```

Placing the results of the two eight bit registers side-by-side after using the ACM type of instruction yields the same result as though the operation had been performed in a sixteen bit register. The concept can be applied to as many eight bit registers as desired.

Armed with the knowledge of how the powerful ACM type of instruction operates, one may proceed to develop a N'th precision addition subroutine. Examine the following routine.

```
  ADDER,  NDA          Always clear carry flag at routine entry
 ADDMOR,  LAM          Get first number into accumulator
          CAL SWITCH   Change pointers to second number
          ACM          Perform ADDITION with CARRY
          LMA          Place result back into memory
          DCB          Decrement the "precision" counter
          RTZ          Exit routine when counter reaches '0'
          INL          Advance second number pointer
          CAL SWITCH   Change pointer back to first number
          INL          Advance first number pointer
          JMP ADDMOR   Repeat process for next precision
```

Note that the above ADDER subroutine requires that a number of the CPU registers be setup prior to calling the routine. The H and L registers must contain the address of the least significant bits register (memory location) for the first multi-word number. Registers D and E similarly must be setup to contain the address of the least significant part of the second multi-word number that is to be added to the first. Finally, register B must be initialized to the "precision," or number of memory words used to hold a multi-word number. Suppose, for example, that a number in triple-precision format is stored in three words starting at location 100 on page 00 and that a second number in similar format is stored at location 200 on page 01. The following instructions would be used to setup the CPU registers prior to calling the ADDER subroutine just described.

```
INIT,  LHI 000      Set page for LSW of first number
       LLI 100      Set location on page for LSW of 1'st number
       LDI 001      Set page for LSW of second number
       LEI 200      Set location on page for LSW of 2'nd number
       LBI 003      Set "precision" value (three words this case)
       CAL ADDER    Call the N'th precision addition routine
```

Note too, that the ADDER subroutine is destructive to the original value of the second number because the answer is left in those locations. If, for some reason, the user wanted to save the original value of the second number, then it would have to be saved elsewhere in memory prior to performing the multi-precision addition operation.

Just as there are two classes of instructions for performing addition with an '8008' CPU, one of which (ACM category) is suited for multiple-precision arithmetic, there are two classes of subtract commands. The SUM (SUBTRACT WITHOUT CARRY) and the SBM (SUBTRACT WITH CARRY, or more appropriately, SUBTRACT WITH BORROW). The SBM type works similarly to the ACM type previously discussed. The CPU first checks the status of the carry flag before performing the subtraction operation. It is thus an easy matter to perform multiple-precision subtraction operations. In fact, one can set up aan almost identical routine to that just described for addition. As in the addition example, one would first setup CPU registers as pointers to the least significant portions of the multiple-precision numbers and load register B with the number of memory words (N) occupied by a N'th precision number.

The routines presented here only utilize the ACM or SBM instructions because the algorithms have been developed as general purpose routines to handle strings of numbers in memory. The reader is reminded that there are a whole group of instructions that have similar functions for working with data while it is in the various CPU registers (such as the ACB, ACC, ACD.... instructions). In addition, there is also the ACI instruction for performing an addition operation with an IMMEDIATE data word. The reader may review the appropriate section in Chapter 1 for a summary of the variations possible when using an '8008' CPU.

EXAMPLE OF AN N'th PRECISION SUBTRACTION SUBROUTINE

```
SUBBER,  NDA          Always clear carry flag at start of routine
SUBTRA,  LAM          Get first number into accumulator
         CAL SWITCH   Change pointers to second number
         SBM          Subtract second from first with borrow
         LMA          Place result back into memory
         DCB          Decrement the "precision" counter
         RTZ          Exit subroutine when counter is '0'
         INL          Advance second number pointer
         CAL SWITCH   Change pointer back to first number
         INL          Advance first number pointer
         JMP SUBTRA   Repeat process for next part of number
```

One thing a person dealing with mathematical functions on a computer will soon have to be concerned with is what happens when a larger number is subtracted from a smaller number. The answer is naturally a minus or negative number. As was initially discussed in the chapter on fundamental programming skills, most microprocessors handle negative numbers utilizing the "two's complement" convention. The reader may want to review the first few pages of that section at this time.

If, for instance, (using single-precision arithmetic) the number '8' (decimal) was subtracted from '6,' the result would appear in the accumulator as shown here:

```
6 decimal  =  0 0  0 0 0  1 1 0  binary
8 decimal  =  0 0  0 0 1  0 0 0  binary
subtracted    ------------------------
    is        1 1  1 1 1  1 1 0  binary
```

Note that the most significant bit in the register containing the minus answer is a '1.' By establishing a "two's complement" convention and always ensuring that the magnitude of any number handled does not interfere with the most significant bit, one may quickly determine whether a number in a register (or series of registers in the case of multiple-precision formatting) is positive or negative. This may be accomplished by testing to see if the most significant bit is a '1' (for a negative number) or '0' (for a positive) value. This is done in an '8008' or similar microprocessor by testing the SIGN FLAG with a JFS, CTS, or similar type CONDITIONAL instruction.

Remember too, that a number may be subtracted from another number by forming the two's complement of the number to be subtracted, then performing an addition operation. Thus:

```
+8 decimal  =  0 0  0 0 1  0 0 0  binary

  2's comp  =  1 1  1 1 1  0 0 0  binary

              consequently

   6 decimal  =  0 0  0 0 0  1 1 0  binary
2's comp of 8    1 1  1 1 1  0 0 0  binary
 when added      -----------------------
       is        1 1  1 1 1  1 1 0  binary
```

It is often desirable to perform a straight two's complement operation on a number in order to change it from a positive to a negative number (or the reverse). One easy way to accomplish this is to simply subtract the number from a value of zero. For multiple-precision work one could simply load one string of memory locations (the first number) with zeroes and place the number to be "negated" in the second string of memory locations. Then simply call the previously illustrated SUBBER subroutine. However, there may be cases where one does not want to disturb values in memory locations or perform the transfer operations necessary to setup the numbers for the SUBBER subroutine. What is needed is a two's complement routine that will operate on a value in the location(s) in which it resides. The following subroutine will accomplish that objective.

```
COMPLM,  LAM       Get least significant bits in first word
         XRI 377   Exclusive OR yields pure complement
         ADI 001   Now add '1' to form two's complement
MORCOM,  LMA       Return 2's complement value to memory
         RAR       Get the carry bit status
         LDA       And save the carry bit status
         DCB       Now decrement the precision counter
         RTZ       Finished subroutine when counter is zero
         INL       If not done, advance memory pointer
         LAM       And fetch the next group of bits
         XRI 377   Produce a pure complement
         LEA       Save pure complement temporarily
```

```
LAD          Get previous carry back into accumulator
RAL          And shift it back out to the carry flag
LAI 000      Do a load so does not disturb carry flag
ACE          Add complemented value with any carry
JMP MORCOM   Go on to do next word in string
```

Notice that in the above COMPLM subroutine it was necessary to save the status of the CARRY FLAG (carry bit) in a CPU register. This was because an XRI or any other BOOLEAN LOGIC instruction in an '8008' CPU automatically clears the carry flag to the zero state and would cause it to "lose" any previous logic one condition.

As in the ADDER and SUBBER subroutines it is also necessary to do some preliminary setting up before calling the COMPLM subroutine. The H and L registers must be set to the first word (least significant bits) of the multi-precision number. Register B must indicate how many words are used to hold the multi-word number.

It will also be pointed out here, that as the programmer gets into developing more and more complicated routines that utilize a lot of subroutines, the programmer must maintain strict control over which CPU registers are affected. The programmer must make sure that the use of selected CPU registers by one routine (especially when it CALLS another subroutine) do not interfere with the over-all operation of a program. The best rule of thumb is to try and leave a subroutine with all the CPU registers, except those transferring information to the next routine, in a FREE or "don't care" state. This is not always possible. When it is not, the programmer must keep track of which registers are being used for a specific purpose and not allow them to be unintentionally altered. For instance, the above COMPLM routine requires that three of the CPU registers be setup prior to entry. The H, L and B registers. When it leaves the subroutine those registers are essentially free for use by the next portion of the program. It also uses the A, D and E registers for operations that it performs. It does not care about the status of those registers when it starts operations because it loads them itself. It also leaves those registers essentially free when the routine is exited. (All the critical operations in the COMPLM subroutine are done with locations in memory.) However, the fact that the routine uses certain CPU registers, such as registers D and E, would be very important to remember if one was using other routines that maintained, say, memory pointers in registers D and E. The novice programmer (and a lot of times the not-so-novice ones) will often find some very strange operations occuring in a newly created program because of problems related to just this aspect!

The ADDER and SUBBER subroutines previously presented could be used by themselves to handle the addition and subtraction of large numbers. However, a restriction on the types of numbers they could handle would be that the numbers have to be whole numbers. Also, as the magnitudes of the numbers to be handled increased, the number of words used to store a value in multi-precision format would have to be increased. As was pointed out earlier, when one starts dealing with numbers of large magnitude, one is primarily concerned with a certain number of SIGNIFICANT DIGITS in a calculation. For instance, one could represent the value ONE MILLION as '1,000,000.' To store this number in multi-precision format requires the use of three memory words in an eight bit microprocessor. However, the number '1,000,000' only contains one significant digit. The number could just as easily be represented as '1' raised to the sixth power of ten. Or, 1 E+6 in what is often termed FLOATING POINT FORMAT. Note that if the number was stored in such format, one would only need to use one memory register in which to hold the single significant digit, plus a separate register in which to hold the power to which the significant digit was to be raised. Floating point format also makes it easy to handle the task of processing fractional numbers. Up to this point, no discussion on representing non-integer numbers has been presented. This will be done shortly. As an introduction, note that the decimal number '0.1' could be represented in floating point format as '1' raised to the 'minus one' power of ten, or 1 E-1.

The reader has now been introduced to multi-precision arithmetic. Hopefully the reader now has an understanding of how large numbers can be stored in several small registers. The term large numbers may be interpreted as meaning numbers containing more than a couple of significant digits. The reader should understand that increasing the number of significant digits requires an increase in the number of binary bits required to store a number. It thus increases the number of memory words required when the number is stored in multiple-precision format. Also, when the format described up to now is used, increasing the magnitude of a number (by adding zeros to the right of the significant digits) rapidly increases the number of words of memory required to hold a number. Finally, just storing a number in a register, without regard to a decimal point location, makes it impossible to properly manipulate fractional numbers.

However, the idea that numbers can be represented as a series of significant digits raised to a power presents a solution to the limitations mentioned. Handling numbers in such a fashion is generally termed "floating-point" arithmetic. The remainder of this chapter will be devoted to developing routines for a FLOATING POINT mathematical program for general purpose applications.

Before proceeding into the development of floating-point routines, it will be necessary to discuss a matter that has been left aside up to this point. That is how to represent fractional numbers utilizing the language of the digital computer, binary arithmetic.

In the decimal numbering system which virtually everyone has been educated in, fractions of a number are represented by digits placed to the right of a decimal point. Each position to the right of such a point represents units of decreasing powers of 10. Thus the number:

```
    0 . 1 2 5
```

actually represents:

```
        1 . .  Tenth (10 E-1)
Plus:     2 .  Hundredths (10 E-2)
Plus:       5  Thousandths (10 E-3)
```

The concept is exactly the same for binary arithmetic except that now each position to the right of the decimal point represents units of decreasing powers of two! Thus the number:

```
    0 . 1 1 1
```

represents:

```
        1 . .  Half (2**-1)
Plus:     1 .  Quarter (2**-2)
Plus:       1  Eighth (2**-3)
```

Thus the above binary number '0.111' represents a fractional number which when converted to decimal is equal to:

1/2 + 1/4 + 1/8 = 7/8 or 0.875 (decimal)

The manner in which fractional binary numbers are represented brings out an interesting point which many readers may have heard of, but not truly understood. That is the introduction of errors into calculations done on a digital computer due to the manipulation of fractions that can not be finalized. As an analogy, there are similar cases in decimal arithmetic. One such case occurs when the number '1' is divided by '3.' The answer is:

0.333333333333333............

which is a non-ending series of '3's after the decimal point. The accuracy or precision with which a calculation involving such a number can be carried out is determined by how many significant digits are used in further calculations involving the fraction. For instance, theoretically, if the number one is divided by three and then multiplied by three, one would get back one as a result. However, if the result of the division is actually multiplied by three, the answer is not actually one, but approaches that value as the number of significant bits used in the calculation is increased. Observe.

```
     0.3   (one significant digit used)
   X   3
   -----
      .9   (answer is off by 10%)

    0.33   (two significant digits used)
  X    3
  --------
     .99   (answer is off by 1%)

   0.333   (three significant digits used)
 X     3
 ---------
    .999   (answer is off by 0.1%)
```

A similar situation exists with binary arithmetic except there are now many more cases where the non-ending fraction situation can occur. For instance, the value '0.1' is truly represented in the decimal system. But, in the binary system, the decimal value '0.1' can only be approximated. As for the decimal case discussed above, the more binary digits used, the closer the value approaches the true value of '0.1.' Observe.

0.0001 (binary) = 1/16 =.0625 (decimal)
Which is off by 37.5%!

0.000110011 = 1/16 + 1/32 + 1/256
+ 1/512 = .0996
Off by just 0.4%!

Note too, that the binary representation is a non-ending series:

0.1 decimal = 0.0001100110011001100....
.... binary

and can not reach the theoretical true value of '0.1' as in the decimal system. Thus, if '0.1' as represented in the binary system is multiplied by, say, '10,' (which can be truly represented in the binary system) the theoretical value of '1.0' can only be approached. The more bits used to hold the binary equivalent, the closer one will approach the true answer. Thus, one may see another reason for using multiple-precision arithmetic in a digital computer, even if one does not want to handle large numbers. This is because, the more bits available to store a fractional number, the more precision one can maintain in performing calculations. One should now also realize, that the more complex a series of mathematical operations, in other words, the more times a number that can not be truly represented is multiplied or divided, the wider will become the margin of error in the final answer!

Now that one has a grasp of how binary digits can represent fractional numbers when placed to the right of a decimal point, one may proceed to investigate floating point arithmetic using a digital computer.

FLOATING POINT ARITHMETIC

Just as one can represent decimal numbers in floating point format, that is, by a string of significant digits raised to a power of ten, one may treat binary numbers in a similar manner as a string of binary digits raised to a power of two.

When handling numbers in floating point format the number is represented in two parts. The significant digits portion is referred to as the MANTISSA. The power to which the significant digits are raised is referred to as the EXPONENT. In decimal floating point format the number '5' could be expressed as:

```
        5.0 E+0  =  5 X 1  =  5

OR     50.0 E-1  =  50 X 1/10  =  5

OR      0.5 E+1  =  0.5 X 10  =  5
```

While in binary floating point format the number could be expressed as:

```
       101.0 E+0  =  5 X 1  =  5

OR  101000.0 E-3  =  40 X 1/8  =  5

OR       0.101 E+3  =  5/8 X 8  =  5
```

It should be remembered that in the decimal example above the EXPONENT represents a power of TEN. In the binary example it represents a power of TWO.

Note that the mechanics of the correspondence between the exponent and the location of the decimal point in the mantissa is the same for both numbering systems. If the significant digits in the mantissa are moved to the right of the decimal point then the exponent is increased one unit for each position the mantissa is shifted. If the digits in the mantissa are shifted to the left, then the exponent is decreased. The only difference between the two systems is that the exponent in the decimal system is specified for powers of ten, while in the binary system it is for powers of two.

The reader may now see that it can be quite a simple matter to handle binary numbers using floating point format if one register (or several registers) is used to hold the mantissa portion, and another register is used to maintain the exponent. Furthermore, a very simple relationship can be maintained between the mantissa and the exponent to facilitate keeping track of a decimal point. Once one has selected a given position as a reference in the mantissa portion, one has only to observe the following procedures for manipulating the number and keeping track of the decimal point:

Each time the MANTISSA is shifted RIGHT

INCREMENT the EXPONENT!

Each time the MANTISSA is shifted LEFT

DECREMENT the EXPONENT!

For the remainder of this chapter, a convention for storing numbers in floating point format will be established and maintained. Numbers will be stored in four consecutive words in memory. The first word in a group will be used to store the EXPONENT with the most significant bit in the word used to represent the SIGN of the EXPONENT. A '1' in the most significant bit position means the number is NEGATIVE. The next three words will hold the MANTISSA portion in triple-precision format. The first bit in the first (most significant word) of the mantissa will be used as the mantissa sign bit. The remaining bits in that word will be the most significant bits of the mantissa. The remaining two words in the mantissa group will hold the less significant bits of the mantissa. Furthermore, there will be an IMPLIED DECIMAL POINT immediately to the right of the sign bit in the mantissa. The format is illustrated here:

```
...EXPONENT...    .......... MSW ...................... MANTISSA ........................ LSW ..........
.            .    .                                                                                 .
S E E E E E E E   S.M M M M M M M    M M M M M M M M     M M M M M M M M

MEM LOC  N+3      MEM LOC  N+2      MEM LOC  N+1        MEM LOC  N
```

Note the order of the memory addresses assigned to the storage of a number. The order of storage is an arbitrary assignment. However, once it has been assigned it must be adhered to within a program. The order shown is the one that will be used in the discussion and program examples for the remainder of this section.

Note too, that a convention has been established that will consider a decimal point (actually, perhaps it should be termed a binary point) to be located to the right of the designated sign bit for the mantissa. This means that all numbers stored in floating point format will be represented as a fractional number! Also, the reader may observe that with one bit out of the three words used to hold the sign of the mantissa, that 23 (decimal) bits are left to hold the actual magnitude of the mantissa. Similarly, the exponent has seven bits in which to represent the magnitude of its value. The eighth bit being used to represent the sign of the exponent. Furthermore, an exponent must be an integer value as there is no implied decimal point in the exponent register.

FLOATING POINT NORMALIZATION

NORMALIZATION may be considered as a standardizing process that will place a number into a fixed position as a reference point from which to commence operations. For the purposes of this discussion, the term NORMALIZATION will mean to place a number into its storage registers so that the mantissa will have a value that is greater than or equal to ONE HALF (1/2) but less than ONE (1). Put another way, this means that any number to be manipulated by a floating point routine will first be shifted so that the most significant binary digit is next to the IMPLIED BINARY POINT in the most significant word of the MANTISSA storage registers. For instance, if a binary number such as:

101.0 E+0 (5 decimal)

was received by an input routine to a floating point program, the number would be NORMALIZED when it was placed in the form:

0.101 E+3 (5/8 X 5 = 5 decimal)

Similarly, if after, say, a binary division operation in which the number '1' had been divided by 10 (decimal) and one had the answer:

0.000110011001100... E+0 (0.1 decimal)

the number would be considered normalized when it was placed in the format:

0.110011001100110... E-3 (0.1 decimal)

Note that normalizing a number is a pretty easy matter. In the first example the number was normalized by shifting the original number to the right until the most significant bit was just to the right of the decimal point. During this procedure, the value of the exponent was incremented for each shifting operation in the mantissa. In the second example, the number is normalized by shifting the original value of the mantissa to the left while decrementing the exponent for each shifting operation in the mantissa.

There are several reasons for wanting to NORMALIZE a number when working with a floating point program. The first has to do with the fact that generally numbers will originate from a human who will be using the computer to manipulate numbers in decimal format. Therefore, the computer will have to convert numbers from say, decimal floating point format, to the binary floating point format used by the computer. There will be more discussion on this matter later in this chapter after a number of binary floating point operations have been presented. The second reason for normalizing numbers, and a very important one, is because the process will allow more significant digits to be retained in a fixed length register. This may be seen by observing in the above example (the case where '0.1' decimal is normalized) that shifting the binary number to the left three places would allow several more LSB's to be placed in a fixed length register for the non-ending binary series 0.110011001100..... and thus allow more accuracy in the binary calculations that might follow!

A routine for normalizing binary numbers will be presented next. In the routine for normalizing numbers, and various other mathematical routines in this chapter, various locations on PAGE 00 will be used for storing numbers that are to be manipulated by the routines as well as for holding COUNTERS and POINTERS used in the routines. A list of the locations reserved for such use on PAGE 00 will be provided later. Also, before getting into the actual binary floating point routines, the reader should be informed that in the following routines, references will be made to a FLOATING POINT ACCUMULATOR and a FLOATING POINT OPERAND. The floating point accumulator and operand will be separate groups of registers consisting of four consecutive memory words on PAGE 00 used to store the active numbers that are manipulated by the floating point routines. They will, of course, be arranged in the format described earlier. That is, a single-word EXPONENT and then a triple-word MANTISSA. The FLOATING POINT ACCUMULATOR will be the focal point for any floating point routine as all the results of floating point calculations will be placed there. The FLOATING POINT OPERAND will be used primarily for holding and manipulating the number that the floating point accumulator operates on. For abbreviation in further discussions, the floating point accumulator will be shortened to FPACC and the operand to FPOP.

```
FPNORM,  LAB           Check register B for special case
         NDA           Set flags after load operation
         JTZ NOEXCO    If B was '0' then do standard normalization
         LLI 127       Otherwise set EXPONENT of FPACC
         LMB           To value found in B at start of routine
NOEXCO,  LLI 126       Set pointer to MSW of FPACC MANTISSA
         LAM           And get MSW of FPACC MANTISSA into ACC
         LLI 100       Change pointer to SIGN storage address
         NDA           Set flags after previous LAM operation
         JTS ACCMIN    If MSB in MSW equals '1' then have minus number
         XRA           If MSB = '0' then have positive value mantissa
         LMA           So set SIGN storage to 000 value
         JMP ACZERT    Proceed to see if FPACC = zero
ACCMIN,  LMA           Original FPACC = negative number, set SIGN
         LBI 004       Set precision counter to four (using extra word)
         LLI 123       And pointer to FPACC LSW-1 (using extra word)
         CAL COMPLM    Two's complement FPACC (using extra word)
ACZERT,  LLI 126       Check to see if FPACC contains zero
         LBI 004       Set a counter
LOOK0,   LAM           Get a part of FPACC
         NDA           Set flags after load operation
         JFZ ACNONZ    If find anything then FPACC is not zero
         DCL           Otherwise move pointer to next part
         DCB           Decrement the loop counter
         JFZ LOOK0     And if not finished check next part
         LLI 127       If reach here FPACC was zero
         XRA           So make sure EXPONENT of FPACC is zero
         LMA           By placing zero in it
         RET           Can then exit NORMALIZATION routine
```

```
ACNONZ,   LLI 123       If FPACC has value, set up pointer and
          LBI 004       Precision value ('4' to handle special cases)
          CAL ROTATL    Then rotate FPACC to the LEFT
          LAM           Now get MSB of MSW from MANTISSA
          NDA           Set flags after load operation
          JTS ACCSET    If MSB = '1' then have found MSB in FPACC
          INL           If not, advance pointer to FPACC EXPONENT
          CAL CNTDWN    And decrement the value of the EXPONENT
          JMP ACNONZ    Then continue in the rotating left loop
ACCSET,   LLI 126       Compensate for last rotate left when MSB
          LBI 003       Found to leave room for SIGN in MSB of the
          CAL ROTATR    FPACC MANTISSA by doing one rotate RIGHT
          LLI 100       Set pointer to original SIGN storage
          LAM           Get original SIGN indicator value
          NDA           Set flags after load operation
          RFS           Finished if value in FPACC is POSITIVE
          LLI 124       Original SIGN is negative, so set pointer to
          LBI 003       LSW of FPACC and also set precision counter
          CAL COMPLM    Now two's complement the NORMALIZED FPACC
          RET           That is all for the FP NORMALIZATION ROUTINE
```

There are several items in the above routine that might confuse the reader if not explained. First of all, the routine checks CPU register B when it is entered. If B contains '0' then the routine will proceed directly on to a new section in the program. If B contains some value, then the value it contains will be placed in the EXPONENT portion of the FPACC. This is done so that the FPNORM subroutine can process numbers that are not initially in floating point form. For instance, when a number is first received from an INPUT device it will generally be in a form such as shown in the example below depicting the binary equivalent of 5 decimal:

00 000 000 00 000 000 00 000 101

As it would appear in standard triple-precision format. Now, the above standard format could be converted to floating point format by assuming that a BINARY POINT existed to the right of the least significant bit, and shifting the entire number to the right while incrementing the binary exponent register. However, the technique would cause a slight problem. How could one tell where the most significant bit of the binary number resided? A way around that problem is to simply shift the registers to the LEFT until the first '1' (MSB) is in the desired position. If this is done, one must first set the EXPONENT portion of the floating point number to the highest possible value that could be contained in the registers. Then, that value is decremented each time the magnitude portion of the number is shifted to the LEFT. In the example presentation, there are 23 decimal bits available for storing the mantissa when triple-precision formatting is being used (24 bits less one which is used to represent the sign of the number). Thus, one would simply load register B with the octal equivalent of 23 decimal which is 27 before calling the FPNORM subroutine whenever one wanted to convert a number in standard form to floating point format! The following illustrations should help clarify the presentation:

ORIGINAL BINARY NUMBER WHEN IT IS IN STANDARD FORMAT

00 000 000 00 000 000 00 000 101

DESIRED FLOATING POINT FORMAT

SE EEE EEE
(exponent followed by mantissa):
S.I III III II III III II III III

NOW ORIGINAL NUMBER PLACED IN FPACC and EXPONENT SET TO 27 (OCTAL)

00 010 111
(exponent followed by mantissa):
0.0 000 000 00 000 000 00 000 101

ORIGINAL NUMBER IS THEN NORMALIZED BY ROTATING LEFT

00 000 011
(exponent followed by mantissa):
0.1 010 000 00 000 000 00 000 000

Since the exponent was decremented each time the number was rotated left the final exponent value is the same as if the number had been rotated to the right to accomplish the normalization while incrementing the exponent from a value of zero!

The reader should also note that the FPNORM subroutine checks to see if the number to be normalized is negative. If it is, the routine keeps track of that fact and makes the number positive in order to accomplish the normalization procedure. If it did not, the normalization routine would not work as may be seen when one recalls what a number such as minus five appears like in its two's complement form:

11 111 111 11 111 111 11 111 011

After the number has been normalized in its positive form, it is converted back to the negative form so that the number minus five would appear when normalized as:

00 000 011
(exponent followed by mantissa):
1.0 110 000 00 000 000 00 000 000

The reader should work through the procedure using pencil and paper to make sure the process is understood when processing negative numbers as it may be confusing at first glance. Note that the normalized minus value has the most significant bit position in the mantissa set to a '1' to indicate a negative value!

Another point of interest in the FPNORM subroutine is that the routine tests to see if the FPACC contains zero. Note that if this test was not made and appropriate action taken to exit the subroutine on such a condition, that the program could become trapped in the rotate left loop as it would fail to ever see a '1' appear in the most significant bit position! When a zero condition is found in the mantissa, the routine sets the exponent part of the FPACC to zero as an additional safety measure.

Finally, the reader may note that the first part of the normalization routine assumes the mantissa uses four memory words. This was done so that the subroutine could handle some special cases that can occur after operations such as multiplication where it may be necessary to have some additional precision. In cases where the feature is not needed, the extra memory word should be set to zero before using the FPNORM subroutine.

The ROTATL and ROTATR subroutines called by FPNORM are short routines that have been set up for N'th-precision operation as with other algorithms discussed in this chapter. Before entering the routines the calling program sets the starting address of the string of memory words to be processed in the H and L CPU registers. It should also set the number of words in the string in register B. The two subroutines are shown next.

```
ROTATL, NDA        Clear carry flag at this entry point
  ROTL, LAM        Fetch word from memory
        RAL        Rotate LEFT (with carry)
        LMA        Restore rotated word to memory
        DCB        Decrement precision counter
        RTZ        Return to calling routine when done
        INL        Otherwise advance pointer to next word
        JMP ROTL   And rotate across the memory word string

ROTATR, NDA        Clear carry flag at this entry point
  ROTR, LAM        Fetch word from memory
        RAR        Rotate RIGHT (with carry)
        LMA        Restore rotated word to memory
        DCB        Decrement precision counter
        RTZ        Return to calling routine when done
        DCL        Going other way so decrement memory pointer
        JMP ROTR   And rotate across the memory word string
```

FLOATING POINT ADDITION

Floating point addition is quite straight forward. In fact, one may use the ADDER subroutine already developed earlier in this chapter for the mantissa portion of a set of floating point numbers. However, there are a few other parameters that must be considered in developing the overall routine.

When two numbers are to be added it will be assumed that they have been positioned in the FPACC and the FPOP memory storage areas. A few items that should be considered in developing the basic floating point addition routine include the following.

Suppose either the FPOP or FPACC contain zero? Or, they both contain zero? In the latter case the routine could be immediately exited as the answer is sitting in the FPACC. If the FPACC is zero, but the FPOP is not, then one has merely to place the contents of the FPOP into the FPACC (as the convention was established earlier that the result of an operation would always be left in the FPACC). For the case where the FPACC contains a value, but the FPOP is zero, one may immediately exit the routine.

But, as will more likely be the case when the floating point ADD routine is called, both the FPACC and the FPOP will contain some non-zero value. Thus one could immediately proceed to perform the addition operation, right? WRONG! Since floating point operations allow the manipulation of numbers with large magnitudes, because of the exponent method of maintaining magnitudes, it is quite possible that an operator might ask for an addition of a very small number to a very large number. (This also might occur in the middle of a complex calculation where an operator was not monitoring the intermediate results.) Readers know that if the difference between the two numbers to be added is so great that there can be no change in the significant digits during the calculation then there is no need to perform the addition process. So, the next step in the floating point addition routine would be to check to see whether or not the magnitudes of the numbers are within significant range of one another. If they are not, then the largest value should be placed in the FPACC as the answer!

If the magnitudes of the two numbers are within significant range then the two numbers may be added. Before this can be done, they must first be ALIGNED by shifting one of the numbers until the exponent is equal in value to that of the second number. The alignment is accomplished by finding out which exponent is the smallest and shifting the mantissa of that number to the right (while incrementing the exponent for each shift) until it is properly aligned. The shifting procedure is quite straightforward since it can be handled by a N'th-precision register rotate subroutine. However, there is one special consideration for the case of a negative number being shifted to the right. One must insert a '1' into the most significant bit position each time such a shift is made in order to maintain the minus value properly (to keep the sign bit in its proper state). This can be accomplished easily as the reader may observe in the following FPADD subroutine by inserting a '1' into the carry bit, then calling the ROTR subroutine. (This is simply another entry point to the ROTATR subroutine presented earlier. The entry point at ROTR avoids the NDA instruction which would cause the carry bit to be cleared to a '0' condition if executed.)

One more consideration that the reader may note in the following FPACC subroutine is that the two numbers to be added are shifted to the right once before the addition is performed so that any overflow from the addition will stay within the FPACC. This will allow normalization to be handled by the previously presented routine instead of having to be concerned with the status of the carry flag at the end of the operation. Because of this shifting operation, an additional memory word is used by both the FPACC and FPOP and the addition is performed using quad-precision. At the end of the addition process the result is normalized and left in the FPACC.

```
  FPADD,   LLI 126       Set pointer to MSW of FPACC
           LBI 003       Set loop counter
 CKZACC,   LAM           Fetch part of FPACC
           NDA           Set flags after loading operation
           JFZ NONZAC    Finding anything means FPACC not zero
           DCB           If that part equals zero, decrement loop counter
           JTZ MOVOP     If FPACC equals zero, move FPOP into FPACC
           DCL           Not finished checking, decrement pointer
           JMP CKZACC    And test next part of FPACC
  MOVOP,   CAL SWITCH    Save pointer to LSW of FPACC
           LHD           Set H equal to zero for sure
           LLI 134       Set pointer to LSW of FPOP
           LBI 004       Set a loop counter
           CAL MOVEIT    Move FPOP into FPACC as answer
           RET           Exit FPADD subroutine
 NONZAC,   LLI 136       Set pointer to MSW of FPOP
           LBI 003       Set loop counter
  CKZOP,   LAM           Get MSW of FPOP
           NDA           Set flags after load operation
           JFZ CKEQEX    If not zero then have a number
           DCB           If zero, decrement loop counter
           RTZ           Exit subroutine if FPOP equals zero
           DCL           Else decrement pointer to next part of FPOP
           JMP CKZOP     And continue testing for zero FPOP
 CKEQEX,   LLI 127       Check for equal exponents
           LAM           Get FPACC exponent
           LLI 137       Change pointer to FPOP exponent
           CPM           Compare exponents
           JTZ SHACOP    If same can setup for ADD operation
```

```
            XRI 377       If not same, then two's complement
            ADI 001       The value of the FPACC exponent
            ADM           And add in FPOP exponent
            JFS SKPNEG    If + then go directly to alignment test
            XRI 377       If negative perform two's complement
            ADI 001       On the result
SKPNEG,     CPI 030       Now see if result greater than 27 octal
            JTS LINEUP    If not can perform alignment
            LAM           If not alignable, get FPOP exponent
            LLI 127       Set pointer to FPACC exponent
            SUM           Subtract FPACC exponent from FPOP exponent
            RTS           FPACC exponent greater so just exit routine
            LLI 124       FPOP was greater, set pointer to FPACC LSW
            JMP MOVOP     Go put FPOP into FPACC and then exit routine
LINEUP,     LAM           Align FPACC and FPOP, get FPOP exponent
            LLI 127       Change pointer to FPACC exponent
            SUM           Subtract FPACC exponent from FPOP exponent
            JTS SHIFTO    FPACC greater so go to shift operand
            LCA           FPOP greater so save difference
MORACC,     LLI 127       Pointer to FPACC exponent
            CAL SHLOOP    Call shift loop subroutine
            DCC           Decrement difference counter
            JFZ MORACC    Continue aligning if not done
            JMP SHACOP    Setup for ADD operation
SHIFTO,     LCA           Shift FPOP routine, save difference count (negative)
MOROP,      LLI 137       Set pointer to FPOP exponent
            CAL SHLOOP    Call shift loop subroutine
            INC           Increment difference counter
            JFZ MOROP     Shift again if not done
SHACOP,     LLI 123       First clear out extra room, setup pointer
            LMI 000       to FPACC LSW+1 and set it to zero
            LLI 127       Now prepare to shift FPACC right once
            CAL SHLOOP    Set pointer and then call shift loop routine
            LLI 137       Shift FPOP right once, first set pointer
            CAL SHLOOP    Call shift loop subroutine
            LDH           Setup pointers, set D equal to zero for sure
            LEI 123       Pointer to LSW of FPACC
            LBI 004       Set precision counter
            CAL ADDER     Add FPACC to FPOP using quad-precision
            LBI 000       Set B for standard normalization procedure
            CAL FPNORM    Normalize the result of the addition
            RET           Exit FPADD subroutine with result in FPACC
SHLOOP,     LBM           Shifting loop for alignment
            INB           Fetch exponent into B and increment
            LMB           Return increment value to memory
            DCL           Decrement the pointer
            LBI 004       Set a counter
FSHIFT,     LAM           Get MSW of floating point number
            NDA           Set flags after loading operation
            JTS BRING1    If number is minus, need to shift in a '1'
            CAL ROTATR    Otherwise perform N'th-precision rotate
```

```
          RET            Exit FSHIFT subroutine
BRING1,   RAL            Save '1' in carry bit
          CAL ROTR       Do ROTATE RIGHT without clearing carry bit
          RET            Exit FSHIFT subroutine
MOVEIT,   LAM            Fetch a word from memory string 'A'
          INL            Advance 'A' string pointer
          CAL SWITCH     Switch pointer to string 'B'
          LMA            Put word from string 'A' into string 'B'
          INL            Advance B string pointer
          CAL SWITCH     Switch pointer back to string 'A'
          DCB            Decrement counter
          RTZ            Return to calling routine when counter is zero
          JMP MOVEIT     Otherwise continue moving operation
```

FLOATING POINT SUBTRACTION

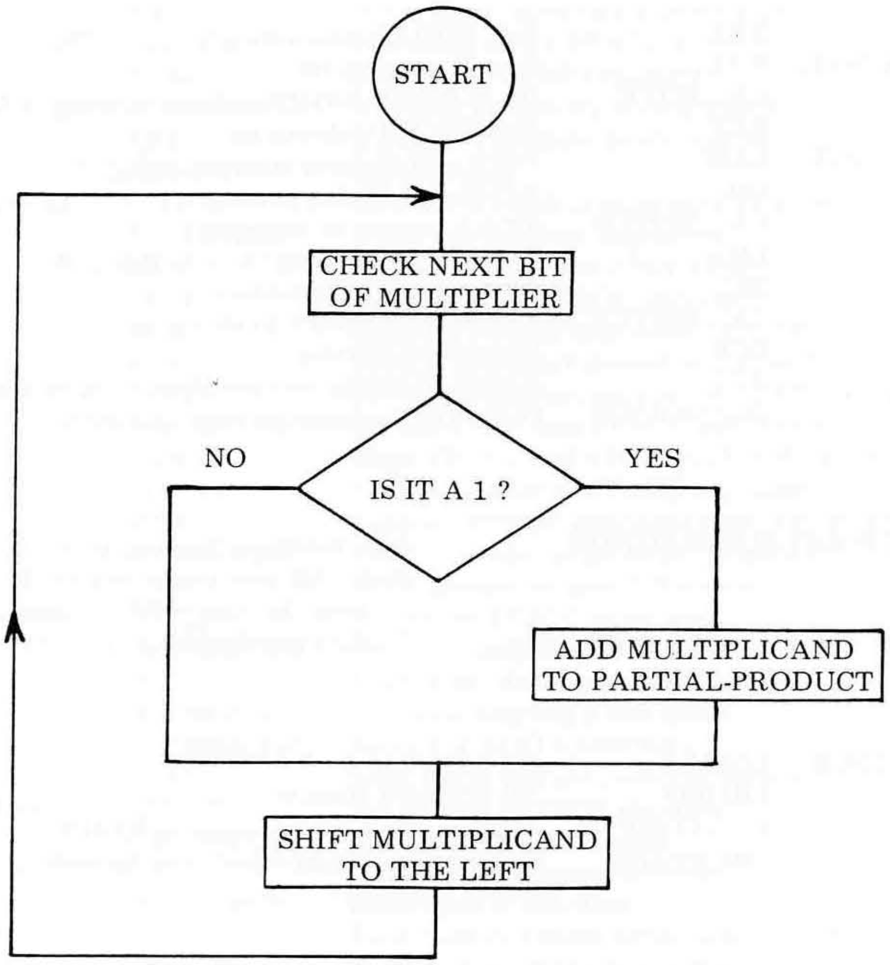
Now that one has a floating point addition routine, floating point subtraction is a snap. All one really has to do is negate the number in the FPACC and jump to the floating point addition routine!

```
FSUB,     LLI 124        Set pointer to LSW of FPACC
          LBI 003        Set precision counter
          CAL COMPLM     Perform two's complement on FPACC
          JMP FPADD      Subtraction accomplished now by adding!
```

FLOATING POINT MULTIPLICATION

Floating point multiplication can be accomplished by utilizing a shifting and adding algorithm for the mantissa portion of the numbers. As pointed out earlier, shifting a binary number to the LEFT serves to essentially DOUBLE its value. An algorithm that takes advantage of that fact can be described as follows.

Consider the two numbers as a MULTIPLIER and a MULTIPLICAND. Examine the least significant bit of the MULTIPLIER. If it is a one, add the current value of the MULTIPLICAND to a third register (which initially starts with a value of zero). Now, shift the MULTIPLICAND one position to the LEFT. Examine the next bit to the LEFT of the least significant bit in the MULTIPLIER. If it is a one, add the current value of the MULTIPLICAND to the third register (which could be called the PARTIAL-PRODUCT register). Shift the MULTIPLICAND to the LEFT again. Continue the process by examining all the bits in the MULTIPLIER for a one condition. Whenever the MULTIPLIER contains a ONE add the current value of the MULTIPLICAND to the PARTIAL-PRODUCT register. After each examination of a bit in the multiplier (and addition of the multiplier to the partial-product register if a '1' was observed) shift the multiplicand LEFT. Continue until all bits in the multiplier have been examined. The result of the multiplication will be in the partial-product register at the completion of the above process. The algorithm can perhaps be seen a little more clearly by studying the flow chart presented next.


START
CHECK NEXT BIT OF MULTIPLIER
IS IT A 1 ?
NO
YES
ADD MULTIPLICAND TO PARTIAL-PRODUCT
SHIFT MULTIPLICAND TO THE LEFT

The reader may verify the algorithm by following the example below for two small numbers, the number '3' as the multiplicand and the number '5' as the multiplier.

```
00 000 011      (Multiplicand at start of operations.)

00 000 101      (Multiplier.)
-----------------
00 000 000      (Partial-product before operations start.)


00 000 011      (Multiplicand when first bit of multiplier
                                  is examined.)
00 000 101      (Least significant bit of multiplier = '1.')
-----------------
00 000 011      (Multiplicand is added to partial-product.)
```

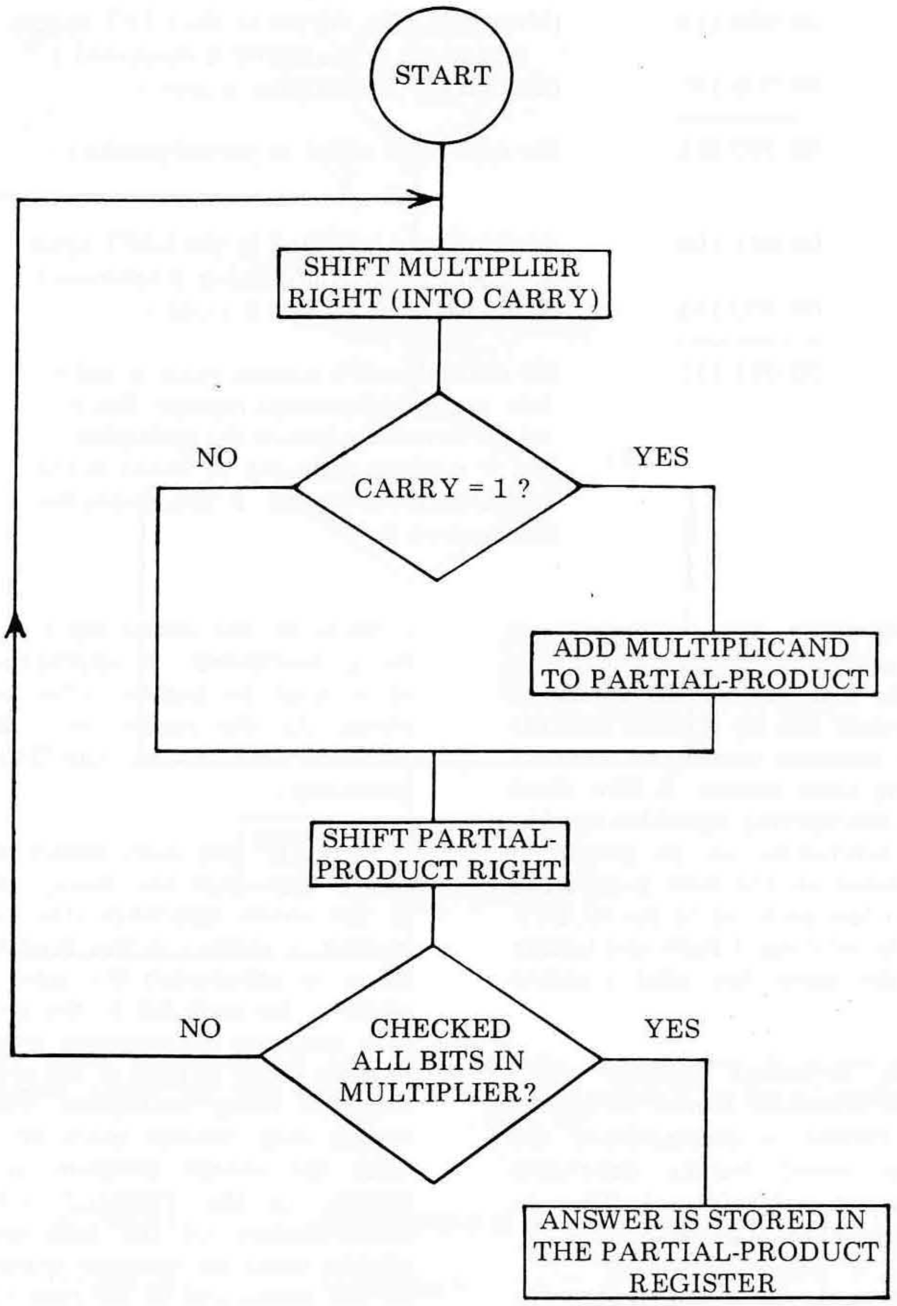
```
00 000 110          (Multiplicand is shifted to the LEFT before
                        second bit of multiplier is examined.)
00 000 101          (Second bit of multiplier is zero.)
-----------------
00 000 011          (So nothing is added to partial-product.)


00 001 100          (Multiplicand is shifted to the LEFT again
                     before next bit of multiplier is examined.)
00 000 101          (Third bit of multiplier is a one.)
-----------------
00 001 111          (So multiplicand's current value is added
                     into the partial-product register. Since
                     all the remaining bits in the multiplier
                     are '0' nothing more will be added to the
                     partial-product register. It thus holds the
                     final answer!)
```

While the algorithm just presented was designed for multiplying numbers that are in standard format, with just a little variation, the basic procedure can by applied towards multiplying the mantissa portion of numbers stored in floating point format. A flow chart of the mantissa multiplying algorithm used in the FPMULT subroutine to be presented shortly is illustrated on the next page. Note that it is easy to test each bit of the MULTIPLIER by simply rotating it right and testing the status of the carry flag after a rotate operation.

Handling the exponent portion when multiplying two numbers stored in binary floating point format is accomplished the same way one would handle exponents in decimal floating point format. The exponents are simply added together.

There are several other parameters to consider when multiplying numbers. First, the algorithm presented may only be used when the numbers are positive in value. Thus, any negative numbers must first be negated before using the algorithm. Furthermore, the reader knows that if two numbers of the same sign are multiplied together the answer will be a positive value, but, if the signs are different, the answer will be a negative number. Therefore, one must take account of the initial signs of the numbers being multiplied. If appropriate, the final value must be negated after using the algorithm. As the reader may observe in the FPMULT subroutine, handling this task is quite easy.

Secondly, the alert reader may have observed that since the multiplicand is shifted in the above algorithm (the partial-product register is shifted in the floating point algorithm to accomplish the same purpose) one position for each bit in the multiplier, then it is necessary to maintain working registers that are twice as long as the original numbers that are being multiplied. Thus, the final answer may contain more bits of precision than the overall program is designed to handle. In the FPMULT subroutine, the multiplication of the mantissas is accomplished using six memory words per register. At the conclusion of the routine, the twenty-third binary bit is rounded off (depending on the status of the twenty-fourth least significant bit) and the answer is normalized back to a 23 bit binary number which is the largest number of bits the package being discussed is designed to normally manipulate. The method allows maximum precision to be maintained during the multiplication process without over-burdening the rest of the floating point routines.


START
SHIFT MULTIPLIER RIGHT (INTO CARRY)
CARRY = 1 ?
NO
YES
ADD MULTIPLICAND TO PARTIAL-PRODUCT
SHIFT PARTIAL-PRODUCT RIGHT
CHECKED ALL BITS IN MULTIPLIER?
NO
YES
ANSWER IS STORED IN THE PARTIAL-PRODUCT REGISTER

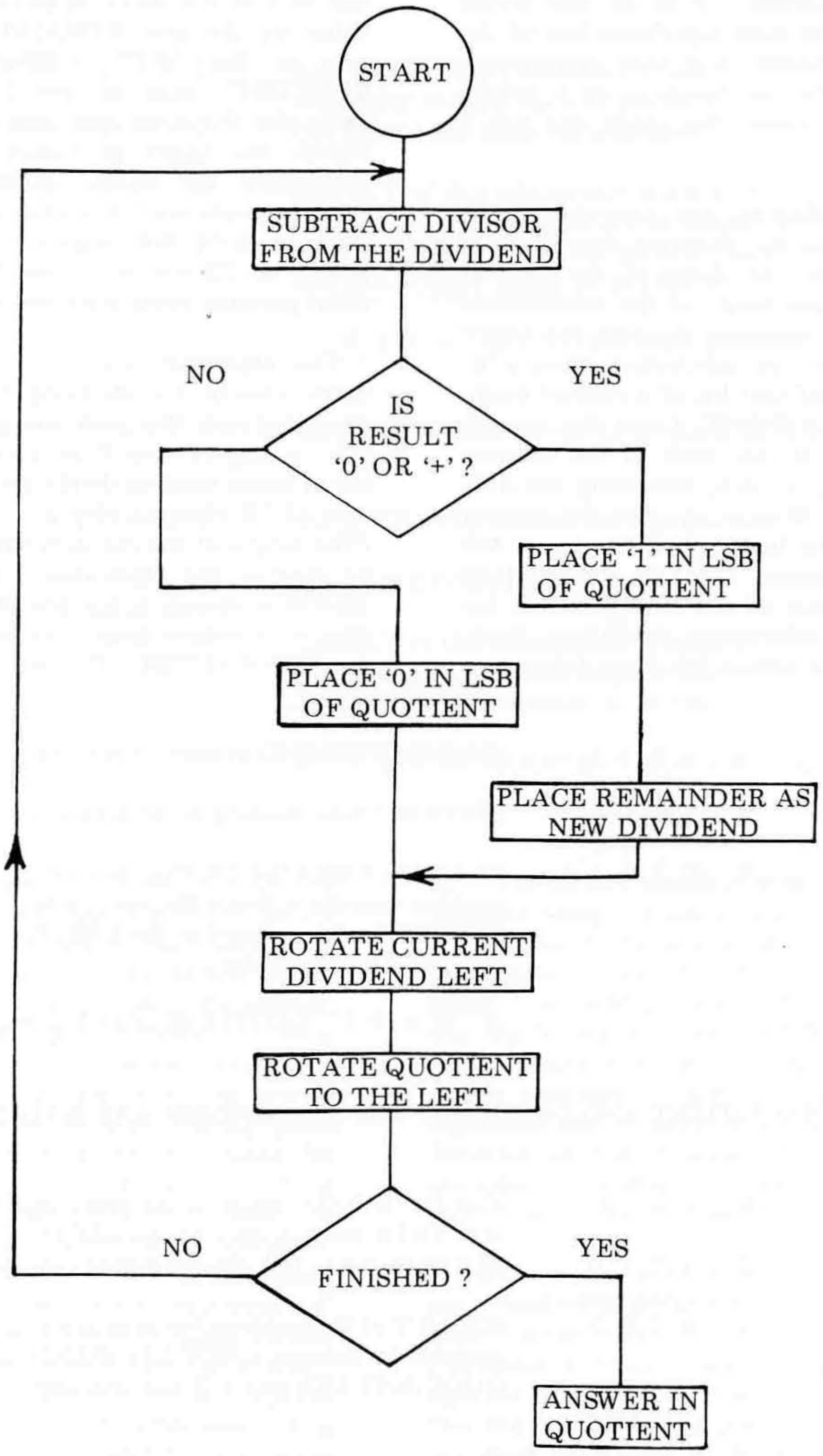
```
FPMULT,   CAL CKSIGN    Setup routine and check sign of numbers
ADDEXP,   LLI 137       Set pointer to FPOP exponent
          LAM           Fetch FPOP exponent into accumulator
          LLI 127       Set pointer to FPACC exponent
          ADM           Add FPACC exponent to FPOP exponent
          ADI 001       Add one for algorithm compensation
          LMA           Store result in FPACC exponent
```

```
SETMCT,   LLI 102        Set bit counter storage pointer
          LMI 027        Set bit counter to 23 decimal (27 octal)
MULTIP,   LLI 126        Basic multiply algorithm, set pntr to MSW of FPACC
          LBI 003        Set precision counter
          CAL ROTATR     Rotate multiplier RIGHT into carry flag
          CTC ADOPPP     If carry equals one, add multiplicand to partial-product
          LLI 146        Set pointer to partial-product MSW
          LBI 006        Set precision counter
          CAL ROTATR     Shift partial-product RIGHT
          LLI 102        Set pointer to bit counter
          CAL CNTDWN     Decrement value in bit counter
          JFZ MULTIP     If bit counter not zero, repeat algorithm
          LLI 146        Set pointer to partial-product MSW
          LBI 006        Set precision counter, now rotate partial-product
          CAL ROTATR     Once more to make room for possible rounding
          LLI 143        Set pointer to access 24'th bit in partial-product
          LAM            Fetch 24'th bit
          RAL            Position it to MSB position
          LAA            NOP inserted to correct algorithm
          NDA            Set flags after rotate operation
          CTS MROUND     If 24'th bit is a '1' then do rounding process
          LLI 123        Now set pointer to FPACC
          CAL SWITCH     Save FPACC pointer
          LHD            Ensure that H is '000'
          LLI 143        Set pointer to partial-product
          LBI 004        Set precision counter
EXMLDV,   CAL MOVEIT     Move answer from partial-product into FPACC
          LBI 000        Set B for standard normalization
          CAL FPNORM     Normalize the answer
          LLI 101        Set pointer to SIGN indicator
          LAM            Fetch SIGN indicator
          NDA            Set flags after load operation
          RFZ            If SIGN has value, result is positive, exit subroutine
          LLI 124        But if SIGN is zero, set FPACC LSW pointer
          LBI 003        And set precision counter
          CAL COMPLM     And negate the answer
          RET            Before exiting the FPMULT subroutine
CKSIGN,   CAL CLRWRK     Clear working locations for multiplication
          LLI 101        Set pointer to SIGN storage
          LMI 001        Place the initial value of '1' into SIGN storage
          LLI 126        Set pointer to MSW of FPACC
          LAM            Fetch MSW of FPACC
          NDA            Set flags after load operation
          JTS NEGFPA     If number is minus, need to do two's complement
OPSGNT,   LLI 136        Set pointer to MSW of FPOP
          LAM            Fetch MSW of FPOP
          NDA            Set flags after load operation
          RFS            If number is positive, return to calling routine
          LLI 101        If number is minus, set pointer to SIGN storage
          CAL CNTDWN     Decrement value of SIGN indicator
          LLI 134        Set pointer to LSW of FPOP
```

```
           LBI 003      Set precision counter
           CAL COMPLM   Perform two's complement of number in FPOP
           RET          Go back to calling routine
NEGFPA,    LLI 101      Set pointer to SIGN storage
           CAL CNTDWN   Decrement value of SIGN indicator
           LLI 124      Set pointer to LSW of FPACC
           LBI 003      Set precision counter
           CAL COMPLM   Negate the value in the FPACC
           JMP OPSGNT   Go check sign of FPOP
CLRWRK,    LLI 140      Clear partial-products work area (140 - 147)
           LBI 010      Set pointer and counter
           XRA          Set accumulator to zero
CLRNEX,    LMA          Deposit accumulator contents into memory
           DCB          Decrement counter
           JTZ CLROPL   When done go to next area
           INL          Else continue clearing partial-product working area
           JMP CLRNEX   By stuffing zeroes in next memory location
CLROPL,    LBI 004      Clear additional room for multiplicand
           LLI 130      At 130 to 133, first set counter and pointer
CLRNX1,    LMA          Put '000' in memory
           DCB          Decrement counter
           RTZ          Return to calling program when done
           INL          Else advance pointer
           JMP CLRNX1   And continue clearing operation
ADOPPP,    LEI 141      Pointer to LSW of partial-product
           LDH          On PAGE 00 in D & E pointer
           LLI 131      Pointer to LSW of multiplicand
           LBI 006      Set precision counter
           CAL ADDER    Perform addition
           RET          Exit subroutine
MROUND,    LBI 003      Set precision counter
           LAI 100      Add '1' to 23'rd bit of partial-product
           ADM          Here
CROUND,    LMA          Restore to memory
           INL          Advance pointer
           LAI 000      Clear ACC without disturbing CARRY FLAG
           ACM          And propogate rounding
           DCB          In partial-product
           JFZ CROUND   Finished when counter equals zero
           LMA          Restore last word of partial-product
           RET          Exit subroutine
```

FLOATING POINT DIVISION

In a manner that is sort of the reverse of multiplication (which uses ADDITION and ROTATE operations) one can perform division using an algorithm that utilizes SUBTRACTION and ROTATE operations. An algorithm will be presented directly in the form used in floating point operations because in this case it is simpler than describing it for numbers that are not in floating point form. The alert reader should have little difficulty observing that the algorithm


START
SUBTRACT DIVISOR FROM THE DIVIDEND
NO
YES
IS RESULT '0' OR '+' ?
PLACE '1' IN LSB OF QUOTIENT
PLACE '0' IN LSB OF QUOTIENT
PLACE REMAINDER AS NEW DIVIDEND
ROTATE CURRENT DIVIDEND LEFT
ROTATE QUOTIENT TO THE LEFT
NO
YES
FINISHED ?
ANSWER IN QUOTIENT

could be used for numbers that are not in floating point format. To do so, one would have to align the most significant bits of the divisor and dividend, and take appropriate action to handle the location of a binary point in cases where the result was not a pure integer.

In rambling English, the algorithm could be stated as follows. Subtract the value of the divisor from the value of the original dividend. Test the result of the subtraction. If the result is negative, meaning the entire divisor could not be subtracted, place a '0' in the least significant bit of a register designated as the QUOTIENT. Leave the current dividend alone. If the result of the subtraction is positive, or zero, indicating the dividend was larger than or equal to the divisor, place a '1' in the least significant bit of the QUOTIENT register, then set the dividend equal to the value of the REMAINDER (or result) of the subtraction operation. Next, once appropriate action has been taken as a result of the subtraction operation, rotate the contents of the dividend (whether its original value or the new REMAINDER) one position to the LEFT. Similarly, rotate the QUOTIENT once to the LEFT to allow room for the next least significant bit. Now repeat the entire procedure until one has performed the above operations as many times as there are bit positions in the register used to hold the original dividend! (That would be 23 decimal times for the floating point package being discussed.)

The algorithm may be visualized a little more clearly by studying the flow chart presented on the previous page. Additionally, a step-by-step illustration of the algorithm being used to divide the binary equivalent of 15 (decimal) by 5 is presented next. (The length of the registers have been reduced to shorten the illustration.) Remember, the algorithm shown is for the MANTISSA portion of numbers once they have been stored in NORMALIZED floating point format!

0 . 1 1 1 1	Original DIVIDEND at start of routine.
0 . 1 0 1 0	DIVISOR (Note floating point format.)
---------------------	
0 . 0 1 0 1	This is the REMAINDER from the subtraction operation. Since the result was POSITIVE a '1' is placed in the LSB of the QUOTIENT register.
	0 . 0 0 0 1 QUOTIENT after 1'st loop.

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

0 . 1 0 1 0	New DIVIDEND (which is the previous REMAINDER rotated once to the LEFT).
0 . 1 0 1 0	DIVISOR (Does not change during routine).
---------------------	
0 . 0 0 0 0	RESULT of this subtraction is zero and thus qualifies to become a NEW DIVIDEND. Also, QUOTIENT LSB gets a '1' for this case!
	0 . 0 0 1 1 QUOTIENT after 2'nd loop.

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

0 . 0 0 0 0	New DIVIDEND (which is the last remainder rotated once to the left).
0 . 1 0 1 0	DIVISOR (still same old number).
---------------------	
1 . 0 1 1 0	RESULT of this subtraction is a minus number (note that the SIGN bit changed). Thus, old DIVIDEND stays in place and QUOTIENT gets a '0' in LSB!

0 . 0 1 1 0 QUOTIENT after 3'rd loop.

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

0 . 0 0 0 0	Old DIVIDEND rotated once to the left.
0 . 1 0 1 0	Same old DIVISOR.
---------------------	
1 . 0 1 1 0	RESULT of this subtraction is again a minus. Old DIVIDEND stays in place. QUOTIENT gets another '0' in LSB.

0 . 1 1 0 0 QUOTIENT after 4'th loop.

Since there were just four bits in the multiplicand register, the algorithm would be completed at the end of the fourth loop in the illustration above. The answer would be that shown in the quotient. Remember, that since floating point format is being used, there would be binary exponents involved. Similar to the way one would handle exponents in decimal floating point notation, one subtracts the exponents for the two numbers (DIVISOR exponent from the DIVIDEND exponent) to obtain the exponent value for a division operation. In the above example, the multiplicand would have had the binary exponent '4' (decimal) to represent the normalized storing of 15 and the divisor would have had a binary exponent of '3.' The above algorithm requires a compensation factor of +1 after subtracting the exponents (can the reader think of ways in which this could be avoided?) in order to have the correct floating point result. In the example being discussed here, (4 - 3) + 1 = 2, and indeed if the answer shown was moved two places to the left (of the implied binary point) one could quickly verify that the result was the binary equivalent of 3 decimal! The reader might want to try using other small valued numbers to test the validity of the algorithm and to develop a thorough understanding of the process. A good case to examine is one where the result is non-ending such as when the number '1' is divided by '3.'

Just as in the multiplication routine, there are several other parameters that must be considered when developing the division routine. For instance, there is again the matter of the signs of the numbers. The algorithm requires that the numbers be in positive format. Again one must keep track of the signs of the original numbers and convert any negative ones to positive values for the routine. If the signs of the two numbers involved are identical, the result must be a positive value. If they are different then the program must negate the answer obtained from the actual division process. And, because some calculations could result in a non-ending series for an answer, some rounding capability must be included in the routine. Then, there is a special case in division that one must check for and take appropriate action upon finding. That is the case of an attempted divide by zero! In such a situation, the program should branch off to notify the operator of an error condition. The floating point routine shown next considers these matters as the reader may observe.

```
     FPDIV,  CAL CKSIGN    Setup registers and check sign of numbers
             LLI 126       Set pointer to MSW of FPACC (DIVISOR)
             LAI 000       Clear accumulator
             CPM           See if MSW of FPACC is zero
             JFZ SUBEXP    If find anything proceed to divide
             DCL           Decrement pointer
             CPM           See if NSW of DIVISOR is zero
             JFZ SUBEXP    If find anything proceed to divide
             DCL           Decrement pointer
             CPM           See if LSW of DIVISOR is zero
             JTZ DERROR    If DIVISOR equals zero, have error condition!
    SUBEXP,  LLI 137       Set pointer to DIVIDEND (FPOP) exponent
             LAM           Fetch DIVIDEND exponent
             LLI 127       Set pointer to DIVISOR (FPACC) exponent
             SUM           Subtract DIVISOR exp from DIVIDEND exp
             ADI 001       Compensate for division algorithm
             LMA           Store exponent result in FPACC exponent
    SETDCT,  LLI 102       Set pointer to bit counter storage
             LMI 027       Set it to 27 octal (23 decimal)
    DIVIDE,  CAL SETSUB    Main division subroutine, subtract DIVIS from DIVID
             JTS NOGO      If result is negative then put '0' in QUOTIENT
             LEI 134       If '+' or '0' then move REMAINDER into DIVIDEND
             LLI 131       Set pointers
             LBI 003       And precision counter
             CAL MOVEIT    And move REMAINDER into DIVIDEND
             LAI 001       Put a '1' into accumulator
             RAR           And move it into the CARRY BIT
             JMP QUOROT    Proceed to ROTATE it into the QUOTIENT
      NOGO,  LAI 000       When RESULT is NEGATIVE, put '0' into ACC
             RAR           And move it into CARRY BIT
    QUOROT,  LLI 144       Set pointer to LSW of QUOTIENT
             LBI 003       Set precision counter
             CAL ROTL      Move CARRY BIT into LSB of QUOTIENT
             LLI 134       Set pointer to DIVIDEND LSW
             LBI 003       Set precision counter
             CAL ROTATL    Rotate DIVIDEND left
             LLI 102       Set pointer to bits counter
             CAL CNTDWN    Decrement bits counter
             JFZ DIVIDE    If not finished then continue algorithm
             CAL SETSUB    Do one more divide for rounding operations
```

```
            JFS DVEXIT    If 24'th bit equal zero then no rounding
            LLI 144       When 24'th bit is '1' set pntr to QUOTIENT LSW
            LAM           Fetch LSW of QUOTIENT
            ADI 001       Add '1' to 23'rd bit
            LMA           Restore LSW
            LAI 000       Clear accumulator while saving CARRY FLAG
            INL           Advance pointer to NSW of QUOTIENT
            ACM           Add with carry
            LMA           Restore NSW
            LAI 000       Clear accumulator while saving CARRY FLAG
            INL           Advance pointer to MSW of QUOTIENT
            ACM           Add with carry
            LMA           Restore MSW
            JFS DVEXIT    If MSB of MSW is zero prepare to exit
            LBI 003       Otherwise set precision counter
            CAL ROTATR    Move QUOTIENT to the RIGHT to clear SIGN BIT
            LLI 127       Set pointer to FPACC exponent
            LBM           Fetch exponent
            INL           Increment it for ROTATE RIGHT operation above
            LMB           Restore exponent
DVEXIT,     LLI 144       Set pointers to transfer
            LEI 124       QUOTIENT to FPACC
            LBI 003       Set precision counter
            JMP EXMLDV    Exit through FPMULT routine at EXMLDV
SETSUB,     LLI 131       Set pointer to LSW of working register
            CAL SWITCH    Save pointer
            LHD           Set H = '0' for sure
            LLI 124       Set pointer to LSW of FPACC
            LBI 003       Set precision counter
            CAL MOVEIT    Move FPACC value to working register
            LEI 131       Reset pointer to working register LSW (DIVISOR)
            LLI 134       Set pointer to LSW of FPOP (DIVIDEND)
            LBI 003       Set precision counter
            CAL SUBBER    Subtract DIVISOR from DIVIDEND
            LAM           Get MSW of RESULT from subtraction operations
            NDA           And set flags after load operation
            RET           Before returning to calling routine

DERROR,     CAL DERMSG    **User defined ERROR routine for handling
            JMP USERDF    Attempted divide by zero, exit as directed**
```

The five fundamental floating point subroutines, FPNORM, FPADD, FPSUB, FPMULT and FPDIV when assembled into object code will fit within three pages of memory in an '8008' system. Additionally, the routines as presented in this chapter use some space on PAGE 00 for storing data and counters. Needless to say, the programs as developed for discussion could be modified to use other memory locations with little difficulty. For reference purposes, the locations used on PAGE 00 by the fundamental floating point routines just presented are listed on the next page.

100	SIGN indicator
101	SIGNS indicator (multiply & divide)
102	Bits counter
123	FPACC extension
124	FPACC least significant word (LSW)
125	FPACC next significant word (NSW)
126	FPACC most significant word (MSW)
127	FPACC exponent
130 - 133	Working area
134	FPOP least significant word
135	FPOP next significant word
136	FPOP most significant word
137	FPOP exponent
140 - 147	Working area

The fundamental floating point routines which have been presented and discussed are extremely powerful routines which should be of considerable value to anyone desiring to manipulate mathematical data in an '8008' or similar system. The routines in the form presented for illustrative purposes are capable of handling binary numbers that are the decimal equivalent of six to seven digits raised to approximately the plus or minus 38'th power of ten! The routines may be used to solve a wide variety of mathematical formulas by simply calling the appropriate subroutines after loading the FPOP and FPACC registers with the values that are to be manipulated (when they are in normalized floating point format). Furthermore, the basic routines illustrated can become the fundamental routines in more sophisticated programs. Such programs might be developed to calculate functions such as SINES and COSINES using numerical techniques such as expansion series formulas.

The interested programmer should have little difficulty in modifying the routines illustrated to upgrade their capability to provide more significant digits (by increasing the length of the mantissa). Or, to extend the exponents capability by providing double or even triple-precision registers for the exponent. For many applications, however, the user will be well satisfied with the capability provided by the routines as they have been presented for educational purposes.

The floating point routines which have been presented are used to manipulate numbers once they are in binary format. In some applications, such as when formulas are being solved by a computer to control the operation of a machine, or applications where there is little or no need to communicate with humans, the above routines coupled with I/O routines and whatever operating programs are dictated by the application, would be sufficient for handling the mathematical operations. However, in probably the majority of applications, at some time or other it will be desirable for humans to communicate with the computer. Or, for the computer to at least present information to humans. It seems that the vast majority of people prefer to manipulate mathematical data using decimal notation. Most people would not want to change their ways by working in floating point binary notation! So, most programmers would find it beneficial to have some conversion routines that would convert numbers from decimal floating point notation to binary floating point notation as well as the reverse. The next section of this chapter is devoted to discussing and developing routines that accomplish such a worthwile objective.

CONVERTING FLOATING POINT DECIMAL TO FLOATING POINT BINARY

Most people using a digital computer for handling mathematical functions would like to input data in the form:

1234.567

OR

1.234 E+3

Using an input device such as a keyboard or electronic typing machine. In order to accept data in such format one needs to develop a program that will first convert the information from the decimal mantissa and exponent form to the binary equivalent. The process is fairly straightforward conceptually.

First, one needs to develop a method for breaking down the mantissa portion into a DECIMAL NORMALIZED format. This may be done quite readily because:

1234.567 = 1234567.0 E-3

AND

1.234 E+3 = 1234.0 E+0

Thus, to effectively normalize a decimal number one has to simply keep track of where the decimal point is placed by the operator in the mantissa. Then one needs to compensate for that factor by removing the decimal point (making the mantissa an integer value) and changing the exponent value to compensate for the removal of the decimal point!

Next, one needs to convert the mantissa portion of the number from decimal to binary. That conversion process can actually be accomplished as each decimal number is inputted by the operator using the algorithm described below.

DECIMAL TO BINARY CONVERSION

Each time a digit is received in decimal form, immediately convert it to its binary equivalent. In many cases this consists of simply MASKING OFF extra bits to leave a value in BCD format. Next, in order to compensate for the powers of ten denoted by the positional weight of decimal numbers, multiply any previous number(s) that are already stored in binary form by multiplying them by ten (decimal). Then add in the binary equivalent of the number that has just been received.

The algorithm can be illustrated by considering the following example. An operator enters the decimal number 63 by first entering the number '6' and then '3' from an input device such as an ASCII encoded keyboard:

```
00 000 000        Input register initially cleared.
```

Operator initially types in the character for a '6.' This is immeidately converted to 1 1 0 as its binary equivalent. Since it is the first character received it is not necessary to multiply the present value of the storage register by ten. The binary value 1 1 0 can simply be placed in the INPUT register giving:

00 000 110	Input register after 1'st number.

The operator then enters the character for a '3.' Once again this is immediately converted to 0 1 1 as its binary equivalent. But, before this new digit is added to the binary storage register, the contents of the register must be multiplied by ten to account for the positional value of the previous digit. A simple way to multiply a binary register by ten is to perform the following steps:

00 000 110	Input register initially contains '6'
00 001 100	Rotate left = multiply by 2
00 011 000	Rotate left = multiply by 4
00 011 110	Add in original value = times 5
00 111 100	Rotate left = multiply by 10

With the previous value of '6' now multiplied by ten to represent 60 (decimal) in the binary register, the new value of '3' can now be added in to yield:

00 111 111	Binary equivalent of 63 (decimal)

The above algorithm is repeated each time an additional decimal character is received to maintain the binary equivalent. The algorithm is valid for multiple-precision storage of numbers.

Finally, it is necessary to convert the decimal exponent value (which again is immediately converted to a binary number as it is received from the input device) to represent a binary number raised to an equivalent value. Conversion at this point may be accomplished by first converting the binary representation of the mantissa to its normalized format (using the special capability of the FPNORM routine). Then multiplying the normalized floating point binary number by 10 (decimal) for each unit of a positive decimal exponent. This can be accomplished by using the FPMULT routine previously described!

The decimal to binary input program to be presented next handles the above considerations plus several other functions. The routine will allow an operator to specify the sign of the decimal mantissa and exponent and takes appropriate action to negate numbers designated as being minus in value. It also allows for erasure of the current input string by typing a special character. The routine assumes that characters are received from an input device that uses ASCII code and that an output device using ASCII code is used to ECHO (repeat back) information as it is received from the input. Neither the actual input or output subroutines are shown in the sample program that follows. (Information on typical I/O routines will be presented in another chapter.) The program also assumes that certain locations on PAGE 00 will be used for storage of numbers received and for maintaining counters and indicators. A listing of the locations used will be provided later. The program calls on other routines previously detailed in this chapter such as FPNORM and FPMULT.

```
 DINPUT,   LHI 000        Set pointer to INPUT
           LLI 150        Storage registers
           XRA            Clear accumulator
           LBI 010        Set a counter
 CLRNX2,   LMA            And clear memory locations 150 - 157
           INL            By depositing zeroes and advancing pointer
           DCB            And decrementing loop counter
           JFZ CLRNX2     Until finished
           LLI 103        Set pointers to counter/indicator storage
           LBI 004        Set a counter
 CLRNX3,   LMA            And clear memory locations 103 - 106
           INL            In a similar fashion by depositing zeroes
           DCB            And decrementing loop counter
           JFZ CLRNX3     Until finished
           CAL INPUT      Now bring in a character from I/O device
           CPI 253        Test to see if it is a '+' sign
           JTZ SECHO      If yes, go to ECHO and continue
           CPI 255        If not '+' see if '-' sign
           JFZ NOTPLM     If not '+' or '-' test for valid character
           LLI 103        If minus, set pointer to INPUT SIGN
           LMA            And make it non-zero by depositing character
 SECHO,    CAL ECHO       Output character in ACC as ECHO to operator
 NINPUT,   CAL INPUT      Fetch a new character from I/O device
 NOTPLM,   CPI 377        See if character is code for RUBOUT
           JTZ ERASE      If yes, prepare to start over
           CPI 256        If not, see if character is a period (.)
           JTZ PERIOD     If '.' process as decimal point
           CPI 305        If not, see if character is 'E' for exponent
           JTZ FNDEXP     If 'E' process as exponent indicator
           CPI 260        If not, see if character is a valid number
           JTS  ENDINP    If none of above, terminate input string
           CPI 272        Still checking for valid number
           JFS ENDINP     If not, terminate input string
           LLI 156        Have a number, set pntr to MSW of INPUT register
           LBA            Save character in register B
           LAI 370        Form a mask and check to see if input
           NDM            Registers can accept larger number
           JFZ NINPUT     If not, ignore present input
           LAB            If O.K., restore character to accumulator
           CAL ECHO       And ECHO number back to operator
           LLI 105        Set pointer to digit counter
           LCM            Fetch digit counter
           INC            Increment its value
           LMC            And restore it to storage
           CAL DECBIN     Perform decimal to binary conversion
           JMP NINPUT     Get next character for mantissa
 PERIOD,   LBA            Subroutine to process '.' - save in B
```

```
          LLI 106         Set pointer to '.' storage indicator
          LAM             Fetch contents
          NDA             Set flags after load operation
          JFZ ENDINP      If '.' already present, end input string
          LLI 105         Otherwise set pointer to digit counter
          LMA             And reset digit counter to zero
          INL             Advance pointer back to '.' storage
          LMB             And put a '.' there
          LAB             Restore '.' to accumulator
          CAL ECHO        And echo it back to operator
          JMP NINPUT      Get next character in number string
ERASE,    LAI 274         Put ASCII code for < in accumulator
          CAL ECHO        Display it
          LAI 240         Put ASCII code for SPACE in ACC
          CAL ECHO        And leave a couple of spaces
          CAL ECHO        Before going back to
          JMP DINPUT      Start the input string over
FNDEXP,   CAL ECHO        Subroutine to process exponent, echo 'E'
          CAL INPUT       Get next part of exponent
          CPI 253         Test for a '+' sign
          JTZ EXECHO      If yes, proceed to echo it
          CPI 255         If not, test for a '-' sign
          JFZ NOEXPS      If not, see if a valid character
          LLI 104         If have '-' then set pointer to EXPONENT SIGN
          LMA             Set EXPONENT SIGN minus indicator
EXECHO,   CAL ECHO        Echo character back to operator
EXPINP,   CAL INPUT       Get next character for exponent portion
NOEXPS,   CPI 377         See if code for RUBOUT
          JTZ ERASE       If yes, prepare to re-enter entire string
          CPI 260         Otherwise check for valid decimal number
          JTS ENDINP      If not, end input string
          CPI 272         Still testing for valid number
          JFS ENDINP      If not, end input string
          NDI 017         Have valid number, form mask and strip ASCII
          LBA             Character to pure BCD, save in register B
          LLI 157         Set pointer to input exponent storage location
          LAI 003         Set accumulator = '3'
          CPM             See if 1'st exponent number was greater than three
          JTS EXPINP      If yes, ignore input (limits exponent to less than 40)
          LCM             If O.K., save previous exponent value in register C
          LAM             And also place it in accumulator
          NDA             Clear the carry bit
          RAL             Multiply times ten algorithm: 1'st multiply by two
          RAL             Multiply by two again
          ADC             Add in original value
          RAL             Multiply by two once more
          ADB             Add in new number to complete the decimal to
          LMA             Binary conversion for exponent and restore to memory
          LAI 260         Restore ASCII code by adding 260
          ADB             To BCD value of the number
          JMP EXECHO      And echo number, then look for next input
```

```
ENDINP,  LLI 103       Set pointer to mantissa SIGN indicator
         LAM           Fetch SIGN indicator
         NDA           Set flags after load operation
         JTZ FININP    If nothing in indicator, number is positive
         LLI 154       Set pointer to LSW of input mantissa
         LBI 003       Set precision
         CAL COMPLM    Perform two's complement to negate number
 FININP, LLI 153       Set pointer to input storage LSW-1
         XRA           Clear accumulator
         LDA           Clear register D
         LMA           Clear input storage location LSW-1
         LEI 123       Set pointer to FPACC LSW-1
         LBI 004       Set precision counter
         CAL MOVEIT    Move input + LSW-1 to FPACC + LSW-1
         LBI 027       Set special FPNORM mode by setting bit count
         CAL FPNORM    In register B and then call normalization routine
         LLI 104       Set pointer to EXPONENT SIGN indicator
         LAM           Fetch EXPONENT SIGN indicator to ACC
         NDA           Set flags after load operation
         LLI 157       Set pointer to decimal exponent storage
         JTZ POSEXP    If exponent positive, jump ahead
         LAM           If exponent negative, fetch it into accumulator
         XRI 377       And perform two's
         ADI 001       complement
         LMA           Then restore to storage location
POSEXP,  LLI 106       Set pointer to period indicator
         LAM           Fetch contents to accumulator
         NDA           Set flags after load operation
         JTZ EXPOK     If nothing, no decimal point involved
         LLI 105       If have decimal point, set pointer to digit
         XRA           Counter then clear accumulator
         SUM           Subtract digit counter from '0' to give negative
 EXPOK,  LLI 157       Set pointer to decimal exponent storage
         ADM           Add in compensation for decimal point
         LMA           Restore compensated value to storage
         JTS MINEXP    If compensated value minus, jump ahead
         RTZ           If compensated value zero, finished!
EXPFIX,  CAL FPX10     Compensated decimal exponent is positive, multiply
         JFZ EXPFIX    FPACC by 10, loop until decimal exponent is zero
         RET           Exit with converted value in FPACC
 FPX10,  LEI 134       Multiply FPACC by 10 subroutine, set pointer to
         LDH           FPOP LSW, then set D = zero for sure
         LLI 124       Set pointer to FPACC LSW
         LBI 004       Set precision counter
         CAL MOVEIT    Move FPACC to FPOP (including exponents)
         LLI 127       Set pointer to FPACC exponent
         LMI 004       Place FP form of 10 (decimal) in FPACC
         DCL           Place FP form of 10 (decimal) in FPACC
         LMI 120       Place FP form of 10 (decimal) in FPACC
         DCL           Place FP form of 10 (decimal) in FPACC
         XRA           Place FP form of 10 (decimal) in FPACC
```

```
          LMA          Place FP form of 10 (decimal) in FPACC
          DCL          Place FP form of 10 (decimal) in FPACC
          LMA          Place FP form of 10 (decimal) in FPACC
          CAL FPMULT   Now multiply original binary number (in FPOP) by ten
          LLI 157      Set pointer to decimal exponent storage
          CAL CNTDWN   Decrement decimal exponent value
          RET          Return to calling program
MINEXP,   CAL FPD10    Compensated decimal exponent is minus, multiply
          JFZ MINEXP   FPACC by 0.1, loop until decimal exponent is zero
          RET          Exit with converted value in FPACC
 FPD10,   LEI 134      Multiply FPACC by 0.1 routine, pointer to FPOP LSW
          LDH          Set D = '0' for sure
          LLI 124      Set pointer to FPACC
          LBI 004      Set precision counter
          CAL MOVEIT   Move FPACC to FPOP (including exponent)
          LLI 127      Set pointer to FPACC exponent
          LMI 375      Place FP form of 0.1 (decimal) in FPACC
          DCL          Place FP form of 0.1 (decimal) in FPACC
          LMI 146      Place FP form of 0.1 (decimal) in FPACC
          DCL          Place FP form of 0.1 (decimal) in FPACC
          LMI 146      Place FP form of 0.1 (decimal) in FPACC
          DCL          Place FP form of 0.1 (decimal) in FPACC
          LMI 147      Place FP form of 0.1 (decimal) in FPACC
          CAL FPMULT   Now multiply original binary number (in FPOP) by 0.1
          LLI 157      Set pointer to decimal exponent storage
          LBM          Fetch value
          INB          Increment it
          LMB          Restore it to memory
          RET          Return to calling program
DECBIN,   LLI 153      Decimal to binary conversion, set pntr to temp storage
          LAB          Restore character to accumulator
          NDI 017      Mask off ASCII bits to leave pure BCD number
          LMA          Place current BCD number in temporary storage
          LEI 150      Set pointer to working area LSW
          LLI 154      Set another pointer to LSB of input registers
          LDH          Set D = '0' for sure
          LBI 003      Set precision counter
          CAL MOVEIT   Move original value to working area
          LLI 154      Set pointer to LSW of INPUT storage
          LBI 003      Set precision counter
          CAL ROTATL   Rotate LEFT (X 2)              (Total = X 2)
          LLI 154      Set pointer to LSW again
          LBI 003      Set precision counter
          CAL ROTATL   Rotate LEFT (X 2)              (Total = X 4)
          LEI 154      Set pointer to LSW of rotated value
          LLI 150      And another to LSW of original value
          LBI 003      Set precision counter
          CAL ADDER    Add original to rotated      (Total now = X 5)
          LLI 154      Set pointer to LSW again
          LBI 003      Set precision counter
          CAL ROTATL   Rotate LEFT (X 2)           (Total now = X 10)
```

```
           LLI 152        Set pointer to clear working area
           XRA            Clear accumulator
           LMA            Deposit in MSW of working area
           DCL            Decrement pointer to MSW
           LMA            Put zero there too
           LLI 153        Set pointer to current digit storage
           LAM            Fetch latest BCD number
           LLI 150        Set pointer to LSW of working area
           LMA            Deposit latest BCD number in LSW
           LEI 154        Seup pointer
           LBI 003        Set precision counter
           CAL ADDER      Add in latest number to complete DECBIN conversion
           RET            Return to calling program
```

CONVERTING FLOATING POINT BINARY TO FLOATING POINT DECIMAL

The following program will convert binary numbers stored in floating point format to decimal floating point format and display them on an output device such as an electronic printer (using ASCII code) in the following format:

+0.1234567 E+07

The routine operates essentially in the reverse manner to the input routine just described. First the binary floating point number is converted to a regularly formatted binary number. Then the number is converted to a decimal number using a multiply by ten algorithm. Since the reader should now be quite adept at following the operation of a program from the commented source listing, the floating point binary to floating point decimal conversion routine will be presented without further discussion. Remember that the routine illustrated assumes an ASCII encoded output device is being utilized. In addition, several subroutines used by the previously illustrated DINPUT program are called by the routine.

```
FPOUT,     LLI 157        Set pointer to decimal exponent storage
           LMI 000        Clear decimal exponent storage location
           LLI 126        Set pointer to MSW FPACC MANTISSA
           LAM            Fetch MSW FPACC MANTISSA to accumulator
           NDA            Set flags after load operation
           JTS OUTNEG     If MSB = 1 have a negative number
           LAI 253        Otherwise number is positive, set ASCII code for '+'
           JMP AHEAD1     Go to display '+' sign
OUTNEG,    LLI 124        Have a negative number, set pntr to LSW FPACC
           LBI 003        Set precision counter
           CAL COMPLM     Perform two's complement on FPACC
           LAI 255        Set ASCII code for '-' sign
AHEAD1,    CAL ECHO       Display sign of MANTISSA
           LAI 260        Set ASCII code for '0'
           CAL ECHO       Display '0'
           LAI 256        Set ASCII code for '.'
           CAL ECHO       Display '.'
           LLI 127        Set pointer to FPACC exponent
           LAI 377        Put '-1' in accumulator
```

```
             ADM            Effectively subtract one from exponent
             LMA            Restore compensated exponent
DECEXT,      JFS DECEXD     If compen exp is zero or positive, multip MANT by 0.1
             LAI 004        If compensated exponent is negative
             ADM            Add '4' (decimal) to exponent value
             JFS DECOUT     If exponent now zero or positive, output MANTISSA
             CAL FPX10      Otherwise, multiply MANTISSA by 10
DECREP,      LLI 127        Set pointer to FPACC exponent
             LAM            Get exponent after multiplication routine
             NDA            Set flags after load operation
             JMP DECEXT     Repeat above test for zero or positive condition
DECEXD,      CAL FPD10      Multiply FPACC by 0.1
             JMP DECREP     Check status of FPACC exponent after multiplication
DECOUT,      LEI 164        Set pointer to LSW of OUTPUT registers
             LDH            Make D = zero for sure
             LLI 124        Set pointers to LSW of FPACC
             LBI 003        Set precision counter
             CAL MOVEIT     Move FPACC to OUTPUT registers
             LLI 167        Set pointer to MSW+1 of OUTPUT register
             LMI 000        And clear that location
             LLI 164        Now set pointer to LSW of OUTPUT register
             LBI 003        Set precision counter, perform one
             CAL ROTATL     Rotate operation to compensate for space of sign bit
             CAL OUTX10     Multiply OUTPUT register by 10, overflow into MSW+1
COMPEN,      LLI 127        Set pointer to FPACC exponent
             LBM            Compensate for any remainder in binary
             INB            Exponent by performing a ROTATE RIGHT on
             LMB            OUTPUT registers until binary exponent becomes zero
             JTZ OUTDIG     Go to output digits when compensation done
             LLI 167        Binary exponent compensation rotate loop
             LBI 004        Set pointer to OUTPUT MSW+1 and set counter
             CAL ROTATR     Perform compensating ROTATE RIGHT operation
             JMP COMPEN     Repeat loop until binary exponent equals zero
OUTDIG,      LLI 107        Set pointer to output digit counter
             LMI 007        Set digit counter to '7' to initialize
             LLI 167        Set pointer to MSD in OUTPUT register MSW+1
             LAM            Fetch BCD form of digit to be displayed
             NDA            Set flags after load operation
             JTZ ZERODG     See if 1'st digit is a '0'
OUTDGS,      LLI 167        If not, set pointer to MSW+1 (BCD code)
             LAI 260        Form ASCII number code by adding 260 (octal)
             ADM            To the BCD code
             CAL ECHO       And display the ASCII encoded decimal number
DECRDG,      LLI 107        Set pointer to output digit counter
             CAL CNTDWN     Decrement value of output digit counter
             JTZ EXPOUT     When it is = '0' go do exponent output routine
             CAL OUTX10     Otherwise multiply OUTPUT register by 10
             JMP OUTDGS     And output next decimal digit
ZERODG,      LLI 157        If 1'st digit, then set pointer to MSW
             CAL CNTDWN     Decrement value to compensate for skipping display
             LLI 166        Of first digit, then set pointer to MSW
```

```
           LAM           Of output registers, fetch contents
           NDA           Set flags after load operations
           JFZ DECRDG    Check to see if entire mantissa is '0'
           DCL           Check to see if entire mantissa is '0'
           LAM           Check to see if entire mantissa is '0'
           NDA           Check to see if entire mantissa is '0'
           JFZ DECRDG    Check to see if entire mantissa is '0'
           DCL           Check to see if entire mantissa is '0'
           LAM           Check to see if entire mantissa is '0'
           NDA           Check to see if entire mantissa is '0'
           JFZ DECRDG    Check to see if entire mantissa is '0'
           LLI 157       If entire mantissa is zero, set pointer to
           LMA           Decimal exponent storage and set it to '0'
           JMP DECRDG    Before proceeding to finish display
OUTX10,    LLI 167       Multiply output registers by 10 to push out
           LMI 000       BCD code of MSD, first clear output MSW+1
           LLI 164       Set pointer to LSW of output registers
           LDH           Make sure D equals zero
           LEI 160       Set another pointer to working area
           LBI 004       Set precision counter
           CAL MOVEIT    Move original value to working area
           LLI 164       Set pointer to original value LSW
           LBI 004       Set precision counter
           CAL ROTATL    Start multiply by 10 routine      (Total = X 2)
           LLI 164       Reset pointer
           LBI 004       And counter
           CAL ROTATL    Multiply by two again             (Total = X 4)
           LLI 160       Set pointer to LSW of original value
           LEI 164       And another to LSW of rotated value
           LBI 004       Set precision counter
           CAL ADDER     Add original value to rotated     (Total = X 5)
           LLI 164       Reset pointer
           LBI 004       And counter
           CAL ROTATL    Multiply by two once more       (Total = X 10)
           RET           Finished multiplying output registers by ten
EXPOUT,    LAI 305       Set ASCII code for letter E
           CAL ECHO      Display E for Exponent
           LLI 157       Set pointer to decimal exponent storage location
           LAM           Fetch decimal exponent to accumulator
           NDA           Set flags after load operation
           JTS EXOUTN    If MSB equals one, value is negative
           LAI 253       If value is positive, set ASCII code for '+' sign
           JMP AHEAD2    Go to display the sign
EXOUTN,    XRI 377       For negative exponent, perform two's comp
           ADI 001       In standard manner
           LMA           And restore to storage location
           LAI 255       Set ASCII code for '-' sign
AHEAD2,    CAL ECHO      Display sign of the exponent
           LBI 000       Clear register B for use as a counter
           LAM           Fetch decimal exponent value
 SUB12,    SUI 012       Subtract 10 (decimal)
```

```
           JTS TOMUCH    Look for negative result
           LMA           Restore positive result, maintain count of how
           INB           Many times 10 (decimal) can be subtracted
           JMP SUB12     to obtain most significant digit of exponent
TOMUCH,    LAI 260       Form ASCII character for MSD of exponent by
           ADB           Adding 260 to count in register B
           CAL ECHO      And display most significant digit of exponent
           LAM           Fetch remainder from decimal exponent storage
           ADI 260       And form ASCII character for LSD of exponent
           CAL ECHO      Display least significant digit of exponent
           RET           Exit FPOUT routine
```

Once one has a decimal to binary INPUT routine, and binary to decimal OUTPUT routine to work with the fundamental floating point routines, it is a relatively simple matter to tie them all together. By doing so, one may form an OPERATING PACKAGE that will allow an operator to specify numerical values in decimal floating point notation, indicate whether addition, subtraction, multiplication, or division was desired, and then obtain an answer from the computer. An illustrative operating program that utilizes all the demonstration routines presented in this section is shown below. The program will allow an operator to make entries and receive results in the format illustrated here:

```
+33.0E+3 X -4 = -0.1320000E+6
```

```
FPCONT,    CAL CRLF2     Display a few Cr's & LF's for I/O device
           CAL DINPUT    Let operator enter a FP decimal number
           CAL SPACES    Display a few spaces after number
           LLI 124       Set pointer to LSW of FPACC
           LDH           Set D = 0 for sure
           LEI 170       Set pointer to temp number storage area
           LBI 004       Set precision counter
           CAL MOVEIT    Move FPACC to temporary storage area
NVALID,    CAL INPUT     Fetch OPERATOR from input device
           LBI 000       Clear register B
           CPI 253       Test for '+' sign
           JTZ OPERA1    Go setup for '+' sign
           CPI 255       If not '+' then test for '-' sign
           JTZ OPERA2    Go set up for '-' sign
           CPI 330       If not above, test for X (multiply) sign
           JTZ OPERA3    Go set up for X sign
           CPI 257       If not above, test for / (divide) sign
           JTZ OPERA4    Go set up for / sign
           CPI 377       If none of above, test for RUBOUT
           JFZ NVALID    If none of above then ignore current input
           JMP FPCONT    If ROBOUT then start a new input sequence
OPERA1,    DCB           Setup register B based on above tests
           DCB           Setup register B based on above tests
OPERA2,    DCB           Setup register B based on above tests
           DCB           Setup register B based on above tests
OPERA3,    DCB           Setup register B based on above tests
```

```
                     DCB           Setup register B based on above tests
          OPERA4,    LCA           Save OPERATOR character in register C
                     LAI ***       *** = Next to last location in LOOK-UP table
                     ADB           Modify  ***  by contents of register B
                     LLI 110       Set pointer to LOOK-UP table address storage
                     LMA           Place LOOK-UP address in storage location
                     LAC           Restore OPERATOR character to ACC
                     CAL ECHO      Display the OPERATOR sign
                     CAL SPACES    Display a few spaces after OPERATOR sign
                     CAL DINPUT    Let operator enter 2'nd FP decimal number
                     CAL SPACES    Provide a few spaces after 2'nd number
                     LAI 275       Place ASCII code for = in accumulator
                     CAL ECHO      Display '=' sign
                     CAL SPACES    Display a few spaces after the '=' sign
                     LLI 170       Set pointer to temporary number storage area
                     LDH           Set D = 000 for sure
                     LEI 134       Set another pointer to LSW of FPOP
                     LBI 004       Set precision counter
                     CAL MOVEIT    Move 1'st number inputted to FPOP
                     LLI 110       Set pointer to LOOK-UP table address storage
                     LLM           Bring in LOW order address of LOOK-UP table
                     LHI XXX       XXX = PAGE this routine located on!
                     LEM           Bring in an address stored in LOOK-UP table
                     INL           Residing on this PAGE (XXX) at LOCATIONS
                     LDM           '*** + B' and '*** + B + 1' and place it
                     LLI Z+1       In registers D and E then change pointer to address
                     LME           Part of instruction labeled RESULT below
                     INL           And transfer the LOOK-UP table contents so that it
                     LMD           Becomes the address portion of  the instruction
                     LHI 000       Labeled RESULT, then restore
                     LDH           registers H and D to zero
                     JMP RESULT    Now JUMP to command labeled RESULT
           CRLF2,    LAI 215       Subroutine to provide CR & LF's
                     CAL ECHO      Place ASCII code for CR in ACC then display
                     LAI 212       Place ASCII code for LF in ACC
                     CAL ECHO      Then display
                     LAI 215       Do it again, first setup code for CR in ACC
                     CAL ECHO      Display it
                     LAI 212       Setup code for LF
                     CAL ECHO      Display it
                     RET           Return to calling routine
          SPACES,    LAI 240       Setup code for SPACE in accumulator
                     CAL ECHO      Display a SPACE
                     LAI 240       Do it again, place code for SPACE in ACC
                     CAL ECHO      Display a SPACE
                     RET           Return to calling routine
   * Z *  RESULT,    CAL DUMMY     CALL the subroutine indicated by current address here
                     CAL FPOUT     Display results of the calculation
                     JMP FPCONT    Go back and wait for next problem input!
LOOK-UP TABLE        AAA           LOW address for start of FPADD subroutine
                     BBB           PAGE address for start of FPADD subroutine
```

CCC	LOW address for start of FPSUB subroutine
DDD	PAGE address for start of FPSUB subroutine
EEE	LOW address for start of FPMULT subroutine
FFF	PAGE address for start of FPMULT subroutine
GGG	LOW address for start of FPDIV subroutine
HHH	PAGE address for start of FPDIV subroutine

The three subroutines, FPINP, FPOUT, and FPCONT as presented would require about three pages of memory for storage. However, as will be discussed in the next chapter, the subroutines could be modified to fit into considerably less memory. The demonstration routines used certain locations on PAGE 00 for storage of transient data and these are listed below for reference. Naturally, the routines could be easily altered to use other temporary storage locations.

LOCATIONS	USAGE
103	Input MANTISSA sign storage
104	Input EXPONENT sign storage
105	Input DIGIT COUNTER
107	Output DIGIT COUNTER
110	Temporary storage for control OPERATOR
150 - 153	Input working area
154 - 156	Input storage registers (for DECBIN conv)
157	Input EXPONENT (decimal equivalent)
160 - 163	Output working area
164 - 167	Output storage registers (for BINDEC conv)
170 - 173	Temporary number storage

ASSEMBLED LISTING OF THE DESCRIBED FLOATING POINT PROGRAM

The following is an assembled listing of the floating point package just described in this chapter as it would appear for an '8008' system. The order in which the major routines appear in the following assembled version is different than the order in which they were presented for explanation. The routines were presented for explanation in a manner related to the increasing complexities of the various portions of the package. The assembled version is arranged more along the logical lines of order of usage. As a guide to the assembled version which is presented next, a memory map shown below gives the starting and ending addresses of the major routines. It may be noted, however, that while the order of the routines have been changed in the assembled version, all of the actual instructions in the routines themselves have been left unchanged. (The assembled version has the comments portion of the listing deleted in order to save space.)

FLOATING POINT PROGRAM MEMORY MAP

ROUTINE	STARTING ADDRESS	ENDING ADDRESS
SCRATCH PAD AREA	PG 00 LOC 100	PG 00 LOC 177
FPCONT	PG 01 LOC 000	PG 01 LOC 243
FPOUT	PG 01 LOC 244	PG 02 LOC 263
DINPUT	PG 02 LOC 264	PG 04 LOC 107
FPNORM	PG 04 LOC 110	PG 04 LOC 237
FPADD	PG 04 LOC 240	PG 05 LOC 114
FSUB	PG 05 LOC 115	PG 05 LOC 126
FPMULT	PG 05 LOC 127	PG 06 LOC 021
FPDIV	PG 06 LOC 022	PG 06 LOC 254
UTILITY ROUTINES	PG 06 LOC 255	PG 07 LOC 004

The assembled version assumes that user defined routines for INPUT and OUTPUT to an I/O device, as well as user defined routines for displaying an attempted divide by zero operation as well as re-directing program operation after such an error, will reside at the locations indicated below.

ROUTINE	STARTING ADDRESS	DEFINITION
DERMSG	PG 07 LOC 100	Attempted divide by zero error message
USERDF	PG 07 LOC 160	Direct program flow after above error
INPUT	PG 07 LOC 200	ASCII input routine
ECHO	PG 07 LOC 300	ASCII Output routine

ASSEMBLED LISTING OF THE FLOATING POINT PROGRAM FOR AN '8 0 0 8' SYSTEM

```
ADDRESS        MACHINE CODE                    MNEMONICS
-------        ------------                    ---------
001 000        106 163 001         FPCONT,     CAL CRLF2
001 003        106 264 002                     CAL DINPUT
001 006        106 210 001                     CAL SPACES
001 011        066 124                         LLI 124
001 013        335                             LDH
001 014        046 170                         LEI 170
001 016        016 004                         LBI 004
001 020        106 076 005                     CAL MOVEIT
001 023        106 200 007         NVALID,     CAL INPUT
001 026        016 000                         LBI 000
001 030        074 253                         CPI 253
```

```
001 032          150 064 001                 JTZ OPERA1
001 035          074 255                     CPI 255
001 037          150 066 001                 JTZ OPERA2
001 042          074 330                     CPI 330
001 044          150 070 001                 JTZ OPERA3
001 047          074 257                     CPI 257
001 051          150 072 001                 JTZ OPERA4
001 054          074 377                     CPI 377
001 056          110 023 001                 JFZ NVALID
001 061          104 000 001                 JMP FPCONT
001 064          011                OPERA1,  DCB
001 065          011                         DCB
001 066          011                OPERA2,  DCB
001 067          011                         DCB
001 070          011                OPERA3,  DCB
001 071          011                         DCB
001 072          320                OPERA4,  LCA
001 073          006 242                     LAI 242
001 075          201                         ADB
001 076          066 110                     LLI 110
001 100          370                         LMA
001 101          302                         LAC
001 102          106 300 007                 CAL ECHO
001 105          106 210 001                 CAL SPACES
001 110          106 264 002                 CAL DINPUT
001 113          106 210 001                 CAL SPACES
001 116          006 275                     LAI 275
001 120          106 300 007                 CAL ECHO
001 123          106 210 001                 CAL SPACES
001 126          066 170                     LLI 170
001 130          335                         LDH
001 131          046 134                     LEI 134
001 133          016 004                     LBI 004
001 135          106 076 005                 CAL MOVEIT
001 140          066 110                     LLI 110
001 142          367                         LLM
001 143          056 001                     LHI 001
001 145          347                         LEM
001 146          060                         INL
001 147          337                         LDM
001 150          066 224                     LLI 224
001 152          374                         LME
001 153          060                         INL
001 154          373                         LMD
001 155          056 000                     LHI 000
001 157          335                         LDH
001 160          104 223 001                 JMP RESULT
001 163          006 215            CRLF2,   LAI 215
001 165          106 300 007                 CAL ECHO
001 170          006 212                     LAI 212
001 172          106 300 007                 CAL ECHO
```

```
001 175         006 215                      LAI 215
001 177         106 300 007                  CAL ECHO
001 202         006 212                      LAI 212
001 204         106 300 007                  CAL ECHO
001 207         007                          RET
001 210         006 240           SPACES,    LAI 240
001 212         106 300 007                  CAL ECHO
001 215         006 240                      LAI 240
001 217         106 300 007                  CAL ECHO
001 222         007                          RET
001 223         106 000 000       RESULT,    CAL DUMMY
001 226         106 244 001                  CAL FPOUT
001 231         104 000 001                  JMP FPCONT
001 234         240                          240
001 235         004                          004
001 236         115                          115
001 237         005                          005
001 240         127                          127
001 241         005                          005
001 242         022                          022
001 243         006                          006

001 244         066 157           FPOUT,     LLI 157
001 246         076 000                      LMI 000
001 250         066 126                      LLI 126
001 252         307                          LAM
001 253         240                          NDA
001 254         160 264 001                  JTS OUTNEG
001 257         006 253                      LAI 253
001 261         104 275 001                  JMP AHEAD1
001 264         066 124           OUTNEG,    LLI 124
001 266         016 003                      LBI 003
001 270         106 311 006                  CAL COMPLM
001 273         006 255                      LAI 255
001 275         106 300 007       AHEAD1,    CAL ECHO
001 300         006 260                      LAI 260
001 302         106 300 007                  CAL ECHO
001 305         006 256                      LAI 256
001 307         106 300 007                  CAL ECHO
001 312         066 127                      LLI 127
001 314         006 377                      LAI 377
001 316         207                          ADM
001 317         370                          LMA
001 320         120 343 001       DECEXT,    JFS DECEXD
001 323         006 004                      LAI 004
001 325         207                          ADM
001 326         120 351 001                  JFS DECOUT
001 331         106 300 003                  CAL FPX10
001 334         066 127           DECREP,    LLI 127
001 336         307                          LAM
001 337         240                          NDA
```

```
001 340          104 320 001              JMP DECEXT
001 343          106 346 003    DECEXD,   CAL FPD10
001 346          104 334 001              JMP DECREP
001 351          046 164        DECOUT,   LEI 164
001 353          335                      LDH
001 354          066 124                  LLI 124
001 356          016 003                  LBI 003
001 360          106 076 005              CAL MOVEIT
001 363          066 167                  LLI 167
001 365          076 000                  LMI 000
001 367          066 164                  LLI 164
001 371          016 003                  LBI 003
001 373          106 340 006              CAL ROTATL
001 376          106 122 002              CAL OUTX10
002 001          066 127        COMPEN,   LLI 127
002 003          317                      LBM
002 004          010                      INB
002 005          371                      LMB
002 006          150 023 002              JTZ OUTDIG
002 011          066 167                  LLI 167
002 013          016 004                  LBI 004
002 015          106 352 006              CAL ROTATR
002 020          104 001 002              JMP COMPEN
002 023          066 107        OUTDIG,   LLI 107
002 025          076 007                  LMI 007
002 027          066 167                  LLI 167
002 031          307                      LAM
002 032          240                      NDA
002 033          150 064 002              JTZ ZERODG
002 036          066 167        OUTDGS,   LLI 167
002 040          006 260                  LAI 260
002 042          207                      ADM
002 043          106 300 007              CAL ECHO
002 046          066 107        DECRDG,   LLI 107
002 050          106 305 006              CAL CNTDWN
002 053          150 177 002              JTZ EXPOUT
002 056          106 122 002              CAL OUTX10
002 061          104 036 002              JMP OUTDGS
002 064          066 157        ZERODG,   LLI 157
002 066          106 305 006              CAL CNTDWN
002 071          066 166                  LLI 166
002 073          307                      LAM
002 074          240                      NDA
002 075          110 046 002              JFZ DECRDG
002 100          061                      DCL
002 101          307                      LAM
002 102          240                      NDA
002 103          110 046 002              JFZ DECRDG
002 106          061                      DCL
002 107          307                      LAM
002 110          240                      NDA
```

```
002 111          110 046 002             JFZ DECRDG
002 114          066 157                 LLI 157
002 116          370                     LMA
002 117          104 046 002             JMP DECRDG

002 122          066 167       OUTX10,   LLI 167
002 124          076 000                 LMI 000
002 126          066 164                 LLI 164
002 130          335                     LDH
002 131          046 160                 LEI 160
002 133          016 004                 LBI 004
002 135          106 076 005             CAL MOVEIT
002 140          066 164                 LLI 164
002 142          016 004                 LBI 004
002 144          106 340 006             CAL ROTATL
002 147          066 164                 LLI 164
002 151          016 004                 LBI 004
002 153          106 340 006             CAL ROTATL
002 156          066 160                 LLI 160
002 160          046 164                 LEI 164
002 162          016 004                 LBI 004
002 164          106 255 006             CAL ADDER
002 167          066 164                 LLI 164
002 171          016 004                 LBI 004
002 173          106 340 006             CAL ROTATL
002 176          007                     RET
002 177          006 305       EXPOUT,   LAI 305
002 201          106 300 007             CAL ECHO
002 204          066 157                 LLI 157
002 206          307                     LAM
002 207          240                     NDA
002 210          160 220 002             JTS EXOUTN
002 213          006 253                 LAI 253
002 215          104 227 002             JMP AHEAD2
002 220          054 377       EXOUTN,   XRI 377
002 222          004 001                 ADI 001
002 224          370                     LMA
002 225          006 255                 LAI 255
002 227          106 300 007   AHEAD2,   CAL ECHO
002 232          016 000                 LBI 000
002 234          307                     LAM
002 235          024 012       SUB12,    SUI 012
002 237          160 247 002             JTS TOMUCH
002 242          370                     LMA
002 243          010                     INB
002 244          104 235 002             JMP SUB12
002 247          006 260       TOMUCH,   LAI 260
002 251          201                     ADB
002 252          106 300 007             CAL ECHO
002 255          307                     LAM
002 256          004 260                 ADI 260
```

```
002 260          106 300 007                  CAL ECHO
002 263          007                          RET

002 264          056 000          DINPUT,     LHI 000
002 266          066 150                      LLI 150
002 270          250                          XRA
002 271          016 010                      LBI 010
002 273          370              CLRNX2,     LMA
002 274          060                          INL
002 275          011                          DCB
002 276          110 273 002                  JFZ CLRNX2
002 301          066 103                      LLI 103
002 303          016 004                      LBI 004
002 305          370              CLRNX3,     LMA
002 306          060                          INL
002 307          011                          DCB
002 310          110 305 002                  JFZ CLRNX3
002 313          106 200 007                  CAL INPUT
002 316          074 253                      CPI 253
002 320          150 333 002                  JTZ SECHO
002 323          074 255                      CPI 255
002 325          110 341 002                  JFZ NOTPLM
002 330          066 103                      LLI 103
002 332          370                          LMA
002 333          106 300 007      SECHO,      CAL ECHO
002 336          106 200 007      NINPUT,     CAL INPUT
002 341          074 377          NOTPLM,     CPI 377
002 343          150 046 003                  JTZ ERASE
002 346          074 256                      CPI 256
002 350          150 022 003                  JTZ PERIOD
002 353          074 305                      CPI 305
002 355          150 066 003                  JTZ FNDEXP
002 360          074 260                      CPI 260
002 362          160 170 003                  JTS ENDINP
002 365          074 272                      CPI 272
002 367          120 170 003                  JFS ENDINP
002 372          066 156                      LLI 156
002 374          310                          LBA
002 375          006 370                      LAI 370
002 377          247                          NDM
003 000          110 336 002                  JFZ NINPUT
003 003          301                          LAB
003 004          106 300 007                  CAL ECHO
003 007          066 105                      LLI 105
003 011          327                          LCM
003 012          020                          INC
003 013          372                          LMC
003 014          106 006 004                  CAL DECBIN
003 017          104 336 002                  JMP NINPUT
003 022          310              PERIOD,     LBA
003 023          066 106                      LLI 106
```

```
003 025           307                                  LAM
003 026           240                                  NDA
003 027           110 170 003                          JFZ ENDINP
003 032           066 105                              LLI 105
003 034           370                                  LMA
003 035           060                                  INL
003 036           371                                  LMB
003 037           301                                  LAB
003 040           106 300 007                          CAL ECHO
003 043           104 336 002                          JMP NINPUT
003 046           006 274                   ERASE,     LAI 274
003 050           106 300 007                          CAL ECHO
003 053           006 240                              LAI 240
003 055           106 300 007                          CAL ECHO
003 060           106 300 007                          CAL ECHO
003 063           104 264 002                          JMP DINPUT
003 066           106 300 007               FNDEXP,    CAL ECHO
003 071           106 200 007                          CAL INPUT
003 074           074 253                              CPI 253
003 076           150 111 003                          JTZ EXECHO
003 101           074 255                              CPI 255
003 103           110 117 003                          JFZ NOEXPS
003 106           066 104                              LLI 104
003 110           370                                  LMA
003 111           106 300 007               EXECHO,    CAL ECHO
003 114           106 200 007               EXPINP,    CAL INPUT
003 117           074 377                   NOEXPS,    CPI 377
003 121           150 046 003                          JTZ ERASE
003 124           074 260                              CPI 260
003 126           160 170 003                          JTS ENDINP
003 131           074 272                              CPI 272
003 133           120 170 003                          JFS ENDINP
003 136           044 017                              NDI 017
003 140           310                                  LBA
003 141           066 157                              LLI 157
003 143           006 003                              LAI 003
003 145           277                                  CPM
003 146           160 114 003                          JTS EXPINP
003 151           327                                  LCM
003 152           307                                  LAM
003 153           240                                  NDA
003 154           022                                  RAL
003 155           022                                  RAL
003 156           202                                  ADC
003 157           022                                  RAL
003 160           201                                  ADB
003 161           370                                  LMA
003 162           006 260                              LAI 260
003 164           201                                  ADB
003 165           104 111 003                          JMP EXECHO
003 170           066 103                   ENDINP,    LLI 103
```

```
003 172         307                         LAM
003 173         240                         NDA
003 174         150 206 003                 JTZ FININP
003 177         066 154                     LLI 154
003 201         016 003                     LBI 003
003 203         106 311 006                 CAL COMPLM
003 206         066 153           FININP,   LLI 153
003 210         250                         XRA
003 211         330                         LDA
003 212         370                         LMA
003 213         046 123                     LEI 123
003 215         016 004                     LBI 004
003 217         106 076 005                 CAL MOVEIT
003 222         016 027                     LBI 027
003 224         106 110 004                 CAL FPNORM
003 227         066 104                     LLI 104
003 231         307                         LAM
003 232         240                         NDA
003 233         066 157                     LLI 157
003 235         150 246 003                 JTZ POSEXP
003 240         307                         LAM
003 241         054 377                     XRI 377
003 243         004 001                     ADI 001
003 245         370                         LMA
003 246         066 106           POSEXP,   LLI 106
003 250         307                         LAM
003 251         240                         NDA
003 252         150 261 003                 JTZ EXPOK
003 255         066 105                     LLI 105
003 257         250                         XRA
003 260         227                         SUM
003 261         066 157           EXPOK,    LLI 157
003 263         207                         ADM
003 264         370                         LMA
003 265         160 337 003                 JTS MINEXP
003 270         053                         RTZ
003 271         106 300 003       EXPFIX,   CAL FPX10
003 274         110 271 003                 JFZ EXPFIX
003 277         007                         RET
003 300         046 134           FPX10,    LEI 134
003 302         335                         LDH
003 303         066 124                     LLI 124
003 305         016 004                     LBI 004
003 307         106 076 005                 CAL MOVEIT
003 312         066 127                     LLI 127
003 314         076 004                     LMI 004
003 316         061                         DCL
003 317         076 120                     LMI 120
003 321         061                         DCL
003 322         250                         XRA
003 323         370                         LMA
```

```
003 324          061                              DCL
003 325          370                              LMA
003 326          106 127 005                      CAL FPMULT
003 331          066 157                          LLI 157
003 333          106 305 006                      CAL CNTDWN
003 336          007                              RET
003 337          106 346 003            MINEXP,   CAL FPD10
003 342          110 337 003                      JFZ MINEXP
003 345          007                              RET
003 346          046 134                FPD10,    LEI 134
003 350          335                              LDH
003 351          066 124                          LLI 124
003 353          016 004                          LBI 004
003 355          106 076 005                      CAL MOVEIT
003 360          066 127                          LLI 127
003 362          076 375                          LMI 375
003 364          061                              DCL
003 365          076 146                          LMI 146
003 367          061                              DCL
003 370          076 146                          LMI 146
003 372          061                              DCL
003 373          076 147                          LMI 147
003 375          106 127 005                      CAL FPMULT
004 000          006 157                          LLI 157
004 002          317                              LBM
004 003          010                              INB
004 004          371                              LMB
004 005          007                              RET

004 006          066 153                DECBIN,   LLI 153
004 010          301                              LAB
004 011          044 017                          NDI 017
004 013          370                              LMA
004 014          046 150                          LEI 150
004 016          066 154                          LLI 154
004 020          335                              LDH
004 021          016 003                          LBI 003
004 023          106 076 005                      CAL MOVEIT
004 026          066 154                          LLI 154
004 030          016 003                          LBI 003
004 032          106 340 006                      CAL ROTATL
004 035          066 154                          LLI 154
004 037          016 003                          LBI 003
004 041          106 340 006                      CAL ROTATL
004 044          046 154                          LEI 154
004 046          066 150                          LLI 150
004 050          016 003                          LBI 003
004 052          106 255 006                      CAL ADDER
004 055          066 154                          LLI 154
004 057          016 003                          LBI 003
004 061          106 340 006                      CAL ROTATL
```

```
004 064         066 152                       LLI 152
004 066         250                           XRA
004 067         370                           LMA
004 070         061                           DCL
004 071         370                           LMA
004 072         066 153                       LLI 153
004 074         307                           LAM
004 075         066 150                       LLI 150
004 077         370                           LMA
004 100         046 154                       LEI 154
004 102         016 003                       LBI 003
004 104         106 255 006                   CAL ADDER
004 107         007                           RET

004 110         301                 FPNORM,   LAB
004 111         240                           NDA
004 112         150 120 004                   JTZ NOEXCO
004 115         066 127                       LLI 127
004 117         371                           LMB
004 120         066 126             NOEXCO,   LLI 126
004 122         307                           LAM
004 123         066 100                       LLI 100
004 125         240                           NDA
004 126         160 136 004                   JTS ACCMIN
004 131         250                           XRA
004 132         370                           LMA
004 133         104 146 004                   JMP ACZERT
004 136         370                 ACCMIN,   LMA
004 137         016 004                       LBI 004
004 141         066 123                       LLI 123
004 143         106 311 006                   CAL COMPLM
004 146         066 126             ACZERT,   LLI 126
004 150         016 004                       LBI 004
004 152         307                 LOOK0,    LAM
004 153         240                           NDA
004 154         110 171 004                   JFZ ACNONZ
004 157         061                           DCL
004 160         011                           DCB
004 161         110 152 004                   JFZ LOOK0
004 164         066 127                       LLI 127
004 166         250                           XRA
004 167         370                           LMA
004 170         007                           RET
004 171         066 123             ACNONZ,   LLI 123
004 173         016 004                       LBI 004
004 175         106 340 006                   CAL ROTATL
004 200         307                           LAM
004 201         240                           NDA
004 202         160 214 004                   JTS ACCSET
004 205         060                           INL
004 206         106 305 006                   CAL CNTDWN
```

```
004 211          104 171 004              JMP ACNONZ
004 214          066 126        ACCSET,   LLI 126
004 216          016 003                  LBI 003
004 220          106 352 006              CAL ROTATR
004 223          006 100                  LLI 100
004 225          307                      LAM
004 226          240                      NDA
004 227          023                      RFS
004 230          066 124                  LLI 124
004 232          016 003                  LBI 003
004 234          106 311 006              CAL COMPLM
004 237          007                      RET

004 240          066 126        FPADD,    LLI 126
004 242          016 003                  LBI 003
004 244          307            CKZACC,   LAM
004 245          240                      NDA
004 246          110 275 004              JFZ NONZAC
004 251          011                      DCB
004 252          150 261 004              JTZ MOVOP
004 255          061                      DCL
004 256          104 244 004              JMP CKZACC
004 261          106 276 006    MOVOP,    CAL SWITCH
004 264          353                      LHD
004 265          066 134                  LLI 134
004 267          016 004                  LBI 004
004 271          106 076 005              CAL MOVEIT
004 274          007                      RET
004 275          066 136        NONZAC,   LLI 136
004 277          016 003                  LBI 003
004 301          307            CKZOP,    LAM
004 302          240                      NDA
004 303          110 314 004              JFZ CKEQEX
004 306          011                      DCB
004 307          053                      RTZ
004 310          061                      DCL
004 311          104 301 004              JMP CKZOP
004 314          066 127        CKEQEX,   LLI 127
004 316          307                      LAM
004 317          066 137                  LLI 137
004 321          277                      CPM
004 322          150 016 005              JTZ SHACOP
004 325          054 377                  XRI 377
004 327          004 001                  ADI 001
004 331          207                      ADM
004 332          120 341 004              JFS SKPNEG
004 335          054 377                  XRI 377
004 337          004 001                  ADI 001
004 341          074 030        SKPNEG,   CPI 030
004 343          160 360 004              JTS LINEUP
004 346          307                      LAM
```

```
004 347         066 127                   LLI 127
004 351         227                       SUM
004 352         063                       RTS
004 353         066 124                   LLI 124
004 355         104 261 004               JMP MOVOP
004 360         307             LINEUP,   LAM
004 361         066 127                   LLI 127
004 363         227                       SUM
004 364         160 004 005               JTS SHIFTO
004 367         320                       LCA
004 370         066 127         MORACC,   LLI 127
004 372         106 046 005               CAL SHLOOP
004 375         021                       DCC
004 376         110 370 004               JFZ MORACC
005 001         104 016 005               JMP SHACOP
005 004         320             SHIFTO,   LCA
005 005         066 137         MOROP,    LLI 137
005 007         106 046 005               CAL SHLOOP
005 012         020                       INC
005 013         110 005 005               JFZ MOROP
005 016         066 123         SHACOP,   LLI 123
005 020         076 000                   LMI 000
005 022         066 127                   LLI 127
005 024         106 052 005               CAL SHLOOP
005 027         066 137                   LLI 137
005 031         106 052 005               CAL SHLOOP
005 034         335                       LDH
005 035         046 123                   LEI 123
005 037         016 004                   LBI 004
005 041         106 255 006               CAL ADDER
005 044         016 000                   LBI 000
005 046         106 110 004               CAL FPNORM
005 051         007                       RET
005 052         317             SHLOOP,   LBM
005 053         010                       INB
005 054         371                       LMB
005 055         061                       DCL
005 056         016 004                   LBI 004
005 060         307             FSHIFT,   LAM
005 061         240                       NDA
005 062         160 071 005               JTS BRING1
005 065         106 352 006               CAL ROTATR
005 070         007                       RET
005 071         022             BRING1,   RAL
005 072         106 353 006               CAL ROTR
005 075         007                       RET
005 076         307             MOVEIT,   LAM
005 077         060                       INL
005 100         106 276 006               CAL SWITCH
005 103         370                       LMA
005 104         060                       INL
```

```
005 105    106 276 006              CAL SWITCH
005 110    011                      DCB
005 111    053                      RTZ
005 112    104 076 005              JMP MOVEIT

005 115    066 124         FSUB,    LLI 124
005 117    016 003                  LBI 003
005 121    106 311 006              CAL COMPLM
005 124    104 240 004              JMP FPADD

005 127    106 257 005     FPMULT,  CAL CKSIGN
005 132    066 137         ADDEXP,  LLI 137
005 134    307                      LAM
005 135    066 127                  LLI 127
005 137    207                      ADM
005 140    004 001                  ADI 001
005 142    370                      LMA
005 143    066 102         SETMCT,  LLI 102
005 145    076 027                  LMI 027
005 147    066 126         MULTIP,  LLI 126
005 151    016 003                  LBI 003
005 153    106 352 006              CAL ROTATR
005 156    142 367 005              CTC ADOPPP
005 161    066 146                  LLI 146
005 163    016 006                  LBI 006
005 165    106 352 006              CAL ROTATR
005 170    066 102                  LLI 102
005 172    106 305 006              CAL CNTDWN
005 175    110 147 005              JFZ MULTIP
005 200    066 146                  LLI 146
005 202    016 006                  LBI 006
005 204    106 352 006              CAL ROTATR
005 207    066 143                  LLI 143
005 211    307                      LAM
005 212    022                      RAL
005 213    300                      LAA
005 214    240                      NDA
005 215    162 002 006              CTS MROUND
005 220    066 123                  LLI 123
005 222    106 276 006              CAL SWITCH
005 225    353                      LHD
005 226    066 143                  LLI 143
005 230    016 004                  LBI 004

005 232    106 076 005     EXMLDV,  CAL MOVEIT
005 235    016 000                  LBI 000
005 237    106 110 004              CAL FPNORM
005 242    066 101                  LLI 101
005 244    307                      LAM
005 245    240                      NDA
005 246    013                      RFZ
```

```
005 247         066 124                 LLI 124
005 251         016 003                 LBI 003
005 253         106 311 006             CAL COMPLM
005 256         007                     RET
005 257         106 336 005   CKSIGN,   CAL CLRWRK
005 262         066 101                 LLI 101
005 264         076 001                 LMI 001
005 266         066 126                 LLI 126
005 270         307                     LAM
005 271         240                     NDA
005 272         160 317 005             JTS NEGFPA
005 275         066 136       OPSGNT,   LLI 136
005 277         307                     LAM
005 300         240                     NDA
005 301         023                     RFS
005 302         066 101                 LLI 101
005 304         106 305 006             CAL CNTDWN
005 307         066 134                 LLI 134
005 311         016 003                 LBI 003
005 313         106 311 006             CAL COMPLM
006 316         007                     RET

005 317         066 101       NEGFPA,   LLI 101
005 321         106 305 006             CAL CNTDWN
005 324         066 124                 LLI 124
005 326         016 003                 LBI 003
005 330         106 311 006             CAL COMPLM
005 333         104 275 005             JMP OPSGNT
005 336         066 140       CLRWRK,   LLI 140
005 340         016 010                 LBI 010
005 342         250                     XRA
005 343         370           CLRNEX,   LMA
005 344         011                     DCB
005 345         150 354 005             JTZ CLROPL
005 350         060                     INL
005 351         104 343 005             JMP CLRNEX
005 354         016 004       CLROPL,   LBI 004
005 356         066 130                 LLI 130
005 360         370           CLRNX1,   LMA
005 361         011                     DCB
005 362         053                     RTZ
005 363         060                     INL
005 364         104 360 005             JMP CLRNX1
005 367         046 141       ADOPPP,   LEI 141
005 371         335                     LDH
005 372         066 131                 LLI 131
005 374         016 006                 LBI 006
005 376         106 255 006             CAL ADDER
006 001         007                     RET
006 002         016 003       MROUND,   LBI 003
006 004         006 100                 LAI 100
```

```
006 006        207                      ADM
006 007        370            CROUND,  LMA
006 010        060                      INL
006 011        006 000                  LAI 000
006 013        217                      ACM
006 014        011                      DCB
006 015        110 007 006              JFZ CROUND
006 020        370                      LMA
006 021        007                      RET

006 022        106 257 005    FPDIV,   CAL CKSIGN
006 025        066 126                  LLI 126
006 027        006 000                  LAI 000
006 031        277                      CPM
006 032        110 047 006              JFZ SUBEXP
006 035        061                      DCL
006 036        277                      CPM
006 037        110 047 006              JFZ SUBEXP
006 042        061                      DCL
006 043        277                      CPM
006 044        150 247 006              JTZ DERROR
006 047        066 137        SUBEXP,  LLI 137
006 051        307                      LAM
006 052        066 127                  LLI 127
006 054        227                      SUM
006 055        004 001                  ADI 001
006 057        370                      LMA
006 060        066 102        SETDCT,  LLI 102
006 062        076 027                  LMI 027
006 064        106 216 006    DIVIDE,  CAL SETSUB
006 067        160 111 006              JTS NOGO
006 072        046 134                  LEI 134
006 074        066 131                  LLI 131
006 076        016 003                  LBI 003
006 100        106 076 005              CAL MOVEIT
006 103        006 001                  LAI 001
006 105        032                      RAR
006 106        104 114 006              JMP QUOROT
006 111        006 000        NOGO,    LAI 000
006 113        032                      RAR
006 114        066 144        QUOROT,  LLI 144
006 116        016 003                  LBI 003
006 120        106 341 006              CAL ROTL
006 123        066 134                  LLI 134
006 125        016 003                  LBI 003
006 127        106 340 006              CAL ROTATL
006 132        066 102                  LLI 102
006 134        106 305 006              CAL CNTDWN
006 137        110 064 006              JFZ DIVIDE
006 142        106 216 006              CAL SETSUB
006 145        120 205 006              JFS DVEXIT
```

```
006 150        066 144                     LLI 144
006 152        307                         LAM
006 153        004 001                     ADI 001
006 155        370                         LMA
006 156        006 000                     LAI 000
006 160        060                         INL
006 161        217                         ACM
006 162        370                         LMA
006 163        006 000                     LAI 000
006 165        060                         INL
006 166        217                         ACM
006 167        370                         LMA
006 170        120 205 006                 JFS DVEXIT
006 173        016 003                     LBI 003
006 175        106 352 006                 CAL ROTATR
006 200        066 127                     LLI 127
006 202        317                         LBM
006 203        060                         INL
006 204        371                         LMB
006 205        066 144         DVEXIT,     LLI 144
006 207        046 124                     LEI 124
006 211        016 003                     LBI 003
006 213        104 232 005                 JMP EXMLDV
006 216        066 131         SETSUB,     LLI 131
006 220        106 276 006                 CAL SWITCH
006 223        353                         LHD
006 224        066 124                     LLI 124
006 226        016 003                     LBI 003
006 230        106 076 005                 CAL MOVEIT
007 233        046 131                     LEI 131
006 235        066 134                     LLI 134
006 237        016 003                     LBI 003
006 241        106 364 006                 CAL SUBBER
006 244        307                         LAM
006 245        240                         NDA
006 246        007                         RET
006 247        106 100 007     DERROR,     CAL DERMSG
006 252        104 160 007                 JMP USERDF

006 255        240             ADDER,      NDA
006 256        307             ADDMOR,     LAM
006 257        106 276 006                 CAL SWITCH
006 262        217                         ACM
006 263        370                         LMA
006 264        011                         DCB
006 265        053                         RTZ
006 266        060                         INL
006 267        106 276 006                 CAL SWITCH
006 272        060                         INL
006 273        104 256 006                 JMP ADDMOR
```

```
006 276          325                        SWITCH,  LCH
006 277          353                                 LHD
006 300          332                                 LDC
006 301          326                                 LCL
006 302          364                                 LLE
006 303          342                                 LEC
006 304          007                                 RET

006 305          327                        CNTDWN,  LCM
006 306          021                                 DCC
006 307          372                                 LMC
006 310          007                                 RET

006 311          307                        COMPLM,  LAM
006 312          054  377                            XRI 377
006 314          004  001                            ADI 001
006 316          370                        MORCOM,  LMA
006 317          032                                 RAR
006 320          330                                 LDA
006 321          011                                 DCB
006 322          053                                 RTZ
006 323          060                                 INL
006 324          307                                 LAM
006 325          054  377                            XRI 377
006 327          340                                 LEA
006 330          303                                 LAD
006 331          022                                 RAL
006 332          006  000                            LAI 000
006 334          214                                 ACE
006 335          104  316  006                       JMP MORCOM

006 340          240                        ROTATL,  NDA
006 341          307                        ROTL,    LAM
006 342          022                                 RAL
006 343          370                                 LMA
006 344          011                                 DCB
006 345          053                                 RTZ
006 346          060                                 INL
006 347          104  341  006                       JMP ROTL

006 352          240                        ROTATR,  NDA
006 353          307                        ROTR,    LAM
006 354          032                                 RAR
006 355          370                                 LMA
006 356          011                                 DCB
006 357          053                                 RTZ
006 360          061                                 DCL
006 361          104  353  006                       JMP ROTR

006 364          240                        SUBBER,  NDA
006 365          307                        SUBTRA,  LAM
```

```
006 366        106 276 006                    CAL SWITCH
006 371        237                            SBM
006 372        370                            LMA
006 373        011                            DCB
006 374        053                            RTZ
006 375        060                            INL
006 376        106 276 006                    CAL SWITCH
007 001        060                            INL
007 002        104 365 006                    JMP SUBTRA


007 100                            DERMSG,


007 160                            USERDF,


007 200                            INPUT,


007 300                            ECHO,
```

USING MEMORY SPACE EFFECTIVELY

The FPINP, FPOUT, FPCONT and other routines presented in the floating point program in the previous chapter might all appear somewhat lengthy to the reader. Indeed, they are all somewhat longer than necessary because they were developed in a manner that would enable one to follow the logic of the program rather than to save memory space in a computer system. As readers know, however, it is often desirable to reduce programs to forms that use a minimum amount of memory. But, there are trade-offs to consider. Designing a program to minimize the amount of memory used generally requires significantly more program development time. It also tends to make the program more complex or difficult for someone else to understand. This is because one of the fundamental techniques to reduce a program's length is to capitalize on making as many subroutines as possible out of different sections of the program. There is another parameter that may be affected by designing a program to use a minimum amount of memory. That is the speed at which the program will execute. As a general rule of thumb, the execution speed will decrease because lots of extra time will be spent executing time consuming CALL instructions. (Note that this is contradictory to what one might initially presume!) More discussion on the considerations of a program's operating speed will be presented in another chapter.

Perhaps the first rule to apply towards reducing the amount of memory a program requires is to maximize the amount of subroutining utilized, provided that the subroutining meets the following simple mathematical relationship (when utilizing an '8008' based or similar machine):

$$B \times N \quad 3 \times N + B + 1$$

where: B = the number of bytes in a repeated instruction sequence.

and: N = the number of times the sequence is used in the program

Examining the formula above will show that it does no good in terms of conserving memory space to call a subroutine that utilizes only three bytes of memory. This is because a CAL instruction itself requires three bytes of memory. (A BYTE is equal to eight binary bits of information and is thus equal to one memory word in an '8008' or similar microcomputer system.) However, once an instruction sequence exceeds three bytes of memory, the point at which subroutining becomes profitable for conserving memory space is a function of 'N' where 'N' is the number of times the instruction sequence needs to be repeated in a program. For example, if B = 4, one starts saving memory space by subroutining when N = 6. The above formula shows that the value of 'N' required to meet the condition where memory space is saved by subroutining drops quite rapidly as B is increased. By the time one is dealing with instructional sequences which use eight or more bytes of memory, one can save memory space by forming a subroutine if that same sequence is used more than once in the program! A summary of the minimum values of B and N that will result in memory space being saved by subroutining based on the above formula is provided here:

B = 4 and N = 6
B = 5 and N = 5
B = 6 and N = 3
B = 8 and N = 2

The amount of memory space that one saves by appropriate subroutining can be ascertained by rearranging the above formula.

$$B \times N - (3 \times N + B + 1) = Z$$

and solving for 'Z' which is the number of bytes saved. For example, if B is 8 and N is 3, then Z is equal to:

$$8 \times 3 - (3 \times 3 + 8 + 1) = 6$$

When developing subroutines, one can often use one routine to serve several functions by allowing for multiple entry points to the subroutine. An example of this method was used in the floating point package discussed. There, two entry points to the rotate subroutines were provided. The ROTATL subroutine, for example, had a second entry point labeled ROTL which allowed one to enter the subroutine after the NDA instruction which resided in the location labeled ROTATL.

Another way to often save significant amounts of memory is by careful organization of the program and assignment of data storage areas in memory. For example, the reader may have noted that all the numerical data storage areas used in the floating point routines along with the counters and indicators were located on PAGE 00. This was done to minimize the resetting of the page pointer (register H). Scattering data on different pages of memory in a large program can result in quite a bit of wasted memory because register H (or other pointers) must be frequently altered. Careful organization of data storage can even be helpful in minimizing the amount of times that register L (or similar pointers) must be loaded with a new address by locating storage areas in accordance with how they are accessed in a program sequence. Then an INL or DCL (one byte commands) may be used to access a storage location rather than a LLI XXX or similar instruction.

In line with the above is the simple rule of maintaining pointers, counters, and other frequently used indicators in CPU registers as much as possible. This considerably reduces the number of times that the memory pointer registers have to be changed to point to locations that contain such information, then changed back to handle the current data that is being manipulated.

Another general rule of thumb for reducing program memory usage is to capitalize on LOOPS. A formula for determining when one can save memory space by using a loop (assuming the loop counter is stored in a CPU register) is presented here:

$$B \times N \qquad B + 6$$

where: B = the number of bytes forming the repeated portion of the sequence that must be repeated.

and: N = the number of times the sequence must be consecutively repeated.

By using the formula, one may verify that if a programmer has a four byte instruction that must be consecutively repeated the programmer can save memory by setting up a loop when the sequence must be repeated three or more times. If B is only two, then a loop conserves memory if it must be consecutively performed five or more times. (The above formula is derived from the fact that it requires six bytes to set up a counter, increment or decrement the counter each time a loop is completed, and make a conditional branching test in an '8008' or similar CPU.)

A subtle concept that can save memory space involves the possibility of including a few carefully chosen instructions in subroutines to increase their general usefulness. For example, consider the subroutine illustrated below:

```
SAMPLE, LCH       Save value of H in C
        LHI XXX   Set pntr to data page
        LAM       Fetch a byte of data
        LHC       Restore orig value of H
        NDA       Set flags for ACC conts
        RET
```

Such a subroutine might be extremely valuable in a large program where data was stored on one page, but counters and indicators had to be stored on another. Before calling the above routine, the program would have to set register L to the appropriate address on the page where data was to be obtained. Suppose that sometimes the main program needed to simply transfer data from one location to another, and at other times it made tests on the data it obtained. The simple inclusion of a NDA instruction in the above routine does no harm in cases where data is to be simply transferred, but it can save valuable memory storage if there are two or more times in which the data must be tested in the main program. For, the NDA sets up the flags allowing one to immediately execute a conditional branching instruction upon return from the subroutine. To push the point being made one step further, adding one more instruction to the above subroutine, an INL placed just before the NDA instruction, could make the routine even more general purpose. For instance, in a typical data manipulating program one might be sequentially accessing locations in the data storage area while possibly searching for a certain code. At other times one might branch off to perform work in another area of memory. In the latter case one would probably have to perform an LLI XXX instruction. Thus, the inclusion of the INL command in the subroutine takes care of all the times that one needs to access the next location in the data area, yet it does no harm if the program will be directed to a different memory area! (Note, however, that one would have to examine carefully just how often the main program might be required to access the exact same location again, thus requiring a compensating DCL instruction in the main portion of the program.)

One of the most powerful memory saving techniques for '8008' systems is based on the use of a class of instructions that many novice programmers completely overlook! This class of instructions is the RESTART (RST XXX) group. For, while the mnemonic for a RESTART instruction is shown as consisting of two parts, the actual machine code results in an effective one byte CALL instruction. While the RST commands were included in the instruction set for the '8008' to facilitate implementing start-up operations in conjunction with the INTERRUPT facility on typical systems, they may also be put to extremely effective usage in general programming applications. The reason is easy to understand once it has been pointed out. Being able to CALL a subroutine with a one byte instruction instead of a three byte instruction can save a large amount of memory space if a routine is called frequently in a program.

The reader may want to review the material in the first chapter which explained the restart instructions. There are eight restart locations on PAGE 00 in an '8008' system. That means that one may have up to eight different subroutines in a program that can be accessed with a single byte CALL instruction. While the restart locations are spaced just eight locations apart, one can still use the restart locations for saving memory space even if the desired subroutines will not fit in eight locations. This may be accomplished by simply placing a JUMP instruction at a RESTART location to direct the program to the actual location of a subroutine!

To see the importance of using RST commands in large programs consider the fact that it may often be necessary to call a particular subroutine 30 or 40 (decimal) times in a program. Using a one byte RST command instead of a three byte CAL instruction can thus save 60 to 80 memory locations. That is roughly one-fourth of a PAGE of memory. Multiply that by a factor of eight, the number of RST locations available, and one can see a very considerable savings in memory usage. The person who has developed a fairly decent sized program for an '8008' system without taking advantage of the RST command to conserve memory is often amazed when such programs are rewritten to utilize the technique.

As a challenge to the reader who is interested in doing a little creative trimming of a program, why not try reducing the size of the FPINP, FPOUT, and FPCONT routines presented in the previous chapter? Using the techniques described in this chapter one should be able to work those routines down from the roughly three pages of memory they required to about two pages!

INPUT/OUTPUT PROGRAMMING

This chapter will be concerned with discussing programming techniques for transferring information to and from the computer and external devices. External devices are connected to the computer via physical connections which carry electronic signals. Since it is often desirable to have a number of different devices connected to a system at one time, a hardware arrangement is generally provided that enables a number of devices to be connected at one time. However, only one such device may actually communicate with the computer at any given instant of time. To allow control of which device will communicate with the computer at any given instant, an electronic arrangement is normally provided that will allow software selection of input and output ports. As far as a programmer is concerned, a port consists of eight parallel electronic signals that may be in the '1' or '0' states. The eight signals correspond to the eight bit positions available in the accumulator of the CPU. An input port accepts information from an external device and presents it to the accumulator. An output port takes information from the accumulator and passes it to an output device. The selection of a particular input or output port is specified by the programmer when utilizing an I/O command. The reader may desire to review the discussion of the I/O instructions presented on page 15 of the chapter describing the instruction set for the 8008 CPU at this time.

> NOTE: For the purposes of the discussion in this chapter, all I/O operations will be assumed to take place between the I/O ports and the accumulator of the CPU. Some readers may be aware that it is possible to communicate with a computer via techniques known as direct memory access, whereby an external device places data directly into areas in memory, or vice-versa. Such transfer techniques are essentially hardware controlled and are outside the pure programming realm to which this manual is devoted.

The basic concept behind communicating with a computer lies in providing some form of systematic system for encoding information from an external device that will allow a program to decode the information and take appropriate action. And, to allow a program to send codes to an external device that will direct it to perform in a desired manner.

Such a system may be created entirely by the programmer. Indeed, in many special applications, such as controlling a unique piece of machinery, that is just the approach taken. For example, suppose some manufacturer had a machine that was to be controlled by the computer. The machine could be constructed so that when it was performing a certain type of function it would close a particular electrical switch. There might be a number of such switches on the machine and each one could be connected to an input line, representing one bit on an input port. For the sake of discussion, suppose a machine had eight such input switches, one connected to each possible line making up an input port. When the switch was closed, a '1' condition would be placed on the line, and when it was open the line would represent a '0' condition. For the sake of simplicity, it could also be assumed that only one switch could be closed at any given time.

Now, assume the computer was to monitor the status of the switches by periodically executing an input instruction for the input port to which the switches were attached. Then, depending on which switch was in the closed condition, the computer would direct information to be outputted on an output port, say, to direct another part of the machine to perform a specific operation. A programmer might make up an input program in the following manner.

```
INCTRL,  INP X          Read data from port X into accumulator
         NDA            Set flags after input operation
         JTZ INCTRL     No switches closed - keep looking
         CPI 001        Is it switch No. 1?
         JTZ START1     Yes, do required routine
         CPI 002        Is it switch No. 2?
         JTZ START2     Yes, do required routine
         CPI 004        Is it switch No. 3?
         JTZ START3     Yes, do required routine
         CPI 010        Is it switch No. 4?
         JTZ START4     Yes, do required routine
          ---
          ---
         CPI 200        Is it switch No. 8?
         JTZ START8     Yes, do required routine
         JMP ERROR      If program ever gets here, something is wrong
```

The above input routine is quite simple and lacks a technical consideration that might be necessary in a real system (how can the routine tell whether a reading indicates a new switch closure or a previous condition still present?). However, it does illustrate the concept of inputting information and having the computer interpret that information.

In a similar manner to the input routine, one could connect, say, the coils of electronic relays to the output lines of a specific output port. Each of the eight possible lines connected to an output port could activate the associated relay when a '1' condition was present, but not when a '0' condition existed. Since each line corresponds to one bit in the accumulator, one could easily develop a program to control the operation of the relays by placing appropriate codes in the accumulator of the CPU, and then executing an OUT Z instruction where Z represented the output port whose lines were connected to the relays.

In the above example input program to monitor the status of a set of switches it was assumed that only one switch could be closed at a given time. Thus, there were only nine possible signal conditions that could be received by the computer - any one of the eight switches, each represented by the status of a particular bit in the accumulator, could be on, or none of them were activated. Thus, the particular coding technique for the example was really quite limited. Had it been stated that any number of the switches could be on at any given time, then there would be 256 different codes possible on the 8 input lines at any given time! Such an encoding scheme would allow quite a lot more information to be conveyed to the computer on one input port. One could readily envision coming up with a system whereby an external machine could use the 256 possible states available on one input port to provide a lot of information to the computer. By assigning different codes to represent different artifacts, one could easily come up with a device that could essentially encode all the letters of the alphabet, the numbers 0 - 9, and a lot of special symbols, and still have unused states! Well, as the reader undoubtably knows, people developed such encoding systems quite some time ago. In fact, a number of different standardized encoding systems have been developed over the years. One of the most popular encoding systems, one that is used on many kinds of machines such as electronic keyboards, typewriters, numerical control machines, and a variety of communication devices, is commonly abbreviated and referred to as the ASCII code. ASCII is the abbreviation for American Standard Code for Information Interchange. ASCII code itself is actually designed to use just 7 bits of information (thus allowing for the encoding of 128 different symbols), however, ASCII code is often used in devices that use 8 bits because the last bit of data can be used to test for transmission errors by serving as a parity indicator. More will be said about parity a little later.

While the entire ASCII code is based on the different patterns that will fit in seven bits of a register, thus yielding 128 (decimal) different codes, a commonly used subset of the ASCII code is often utilized. The subset does not use every possible pattern but only those patterns desired. The subset referred to is frequently used in ASCII coded keyboards, teletype machines, and other devices. In the listing shown below, the 8th bit not used by the ASCII code will be shown as a '1' condition, and the codes will be pre-

CHARACTER	BINARY	OCTAL
A	11 000 001	301
B	11 000 010	302
C	11 000 011	303
D	11 000 100	304
E	11 000 101	305
F	11 000 110	306
G	11 000 111	307
H	11 001 000	310
I	11 001 001	311
J	11 001 010	312
K	11 001 011	313
L	11 001 100	314
M	11 001 101	315
N	11 001 110	316
O	11 001 111	317
P	11 010 000	320
Q	11 010 001	321
R	11 010 010	322
S	11 010 011	323
T	11 010 100	324
U	11 010 101	325
V	11 010 110	326
W	11 010 111	327
X	11 011 000	330
Y	11 011 001	331
Z	11 011 010	332
[	11 011 011	333
\	11 011 100	334
]	11 011 101	335
↑	11 011 110	336
←	11 011 111	337
SPACE	11 100 000	240

CHARACTER	BINARY	OCTAL
!	10 100 001	241
"	10 100 010	242
#	10 100 011	243
$	10 100 100	244
%	10 100 101	245
&	10 100 110	246
'	10 100 111	247
(	10 101 000	250
)	10 101 001	251
*	10 101 010	252
+	10 101 011	253
,	10 101 100	254
-	10 101 101	255
.	10 101 110	256
/	10 101 111	257
0	10 110 000	260
1	10 110 001	261
2	10 110 010	262
3	10 110 011	263
4	10 110 100	264
5	10 110 101	265
6	10 110 110	266
7	10 110 111	267
8	10 111 000	270
9	10 111 001	271
:	10 111 010	272
;	10 111 011	273
<	10 111 100	274
=	10 111 101	275
>	10 111 110	276
?	10 111 111	277
@	11 000 000	300

sented as they could appear in the registers of an 8008 CPU.

The subset of the ASCII code just presented has several nice features worth noting. For instance, the 26 letters of the alphabet are all encoded in a sequence starting with 301 (octal) and ending with 332 (octal). Thus one can easily check data, for example, being inputted by an operator to see if the code being received represents a letter of the alphabet by performing a range test as illustrated below.

```
CKALFA,   INP X         Accept a character from input device
          CPI 301       See if input in range from 301
          JTS CKALFA    To 332, if it is not, ignore the
          CPI 333       Input, if it is within the range
          JFS CKALFA    Then have an alphabetical character
ISALFA,   . . .         To process as desired
```

The reader may note that the numbers 0 through 9 are also grouped together in the sequence from 260 to 271, and the programmer can thus readily perform a similar range test to only accept numbers.

There are several other characters that are used by many machines that operate with ASCII code that will be mentioned for reference. The functions carriage return (215), line-feed (212), bell (207), and RUBOUT (377) are most often found on automatic typing machines which make very nice I/O devices for a computer.

When an input instruction is executed, the computer will receive eight bits of information simultaneously, corresponding to the eight possible lines of an input port which are fed into the accumulator. In other words, the data is accepted in parallel. Likewise, when an output instruction is executed, the computer will send all eight bits in the accumulator out to the appropriate output port simultaneously. However, some devices which one desires to operate with the computer may not be parallel devices. They may instead be serially operated, which means they do not transmit information over a group of wires, but rather send the information one bit at a time over a single wire. Such devices may, however, still be connected to an 8008 system since one may simply discard the unused bits corresponding to unused lines of an I/O port. In such cases, the programmer must know which line of a port is the active line and take care to ensure that the program manipulates bits of information so that they appear on that line at the proper time. Whether a particular device connected to a computer is serial or parallel in operation (as far as the computer is concerned) is often a function of the type of hardware interface provided for the external device. For instance, electric typing machines are essentially serial devices since they act on information one bit at a time. However, when actually connected to a computer, one can elect to have a hardware interface that converts information received from the machine in serial form and places it in a parallel register before passing the data to the computer. Going in the other direction one may have the computer send data in parallel form to the interface which will then pass it on to the machine in bit-serial fashion. Such an interface can save a lot of computer time because the external hardware interface is able to handle the time consuming serial to parallel and parallel to serial tasks. However, such hardware costs money, and in many applications one may desire to have the computer do the serial to parallel conversion and vice-versa. This can be accomplished quite readily with a suitable program that actually utilizes the computer's own timing to determine when to look or sample for the next bit of information from the serial device, or when to send the next bit of information to the serial device. While the details of carefully controlling the timing for such a program will be discussed in the next chapter, the concept of having the computer perform parallel to serial or serial to parallel conversion will be demonstrated with several routines at this point. The technique consists of using accumulator rotate instructions to shift the serial data in or out of the computer.

In the parallel to serial routine shown next, it will be assumed that a device that accepts serial data is connected to the least significant bit line of output port X, and that the remaining lines available on the port are unused. The device will be assumed to be a unit that operates with ASCII code, and before the illustrated routine is called, that the code for a character has been placed in the accumulator.

```
PARSER,   LCI 010        Set up register C as a bit counter
NEXOUT,   OUT X          Output data in ACC to port X, only the
          RRC            Data in LSB used, now rotate ACC right
          DCC            Ignore carry, then decrement bit counter
          JFZ NEXOUT     Do next bit if counter not zero
          RET            Exit routine when all 8 bits transmitted
```

In the following serial to parallel routine it is assumed that data is arriving at the most significant bit position of an input port, and that it is to be assembled into an eight bit format.

```
SERPAR,   XRA            Clear accumulator and also clear
          LBA            Register B at start of routine
          LCI 010        Set a bit counter
NEXTIN,   INP X          Bring in data from input port X
          NDI 200        Since only MSB has important data, mask
          RAL            Off other bits and clear carry, now rotate
          ADB            Left to save new bit, then add in any
          RAR            Previous bits from B and rotate right
          LBA            To add on latest bit, store in B
          DCC            Decrement bit counter
          JFZ NEXTIN     If not finished, get next bit
          RET            Exit routine when 8 bits received and stored
```

Another popular standardized code for operating I/O devices is known as BAUDOT code. BAUDOT code is a 5 level code in that it requires five bits to specify a particular character. Thus, there are theoretically 32 different patterns that can be represented when using BAUDOT code. Now, BAUDOT code has long been used in a variety of electro-mechanical typing machines and other communication devices, and the code is of interest to many computer owners because older model machines, paper tape punches, and paper tape readers can often be obtained from second-hand sources at quite reasonable prices, and used as an I/O device for a computer. While BAUDOT code can only represent 32 different bit patterns, these machines can print all the letters of the alphabet, the numbers 0 through 9, and a variety of punctuation symbols! That is a lot more than 32 different characters! How is it done?

Well, the designers of those machines used a little ingenuity to enable the machines to handle almost double the number of characters that could be represented by a five bit code by using several of the codes to shift the machine between two modes, so that in one mode it would interpret the codes to mean one set of characters, and in the other mode it would interpret the codes to represent a different set of characters. In one mode, termed the letters mode, all the letters of the alphabet may be printed. In the figures mode, numbers and punctuation are printed. The BAUDOT code is presented below.

LC	UC	BIT POSITION	CODES
A	-	0 0 0 1 1	003
B	?	1 1 0 0 1	031
C	:	0 1 1 1 0	016
D	$	0 1 0 0 1	011
E	3	0 0 0 0 1	001
F	!	0 1 1 0 1	015
G	&	1 1 0 1 0	032
H	#	1 0 1 0 0	024
I	8	0 0 1 1 0	006
J	'	0 1 0 1 1	013
K	(	0 1 1 1 1	017
L	)	1 0 0 1 0	022
M	.	1 1 1 0 0	034
N	,	0 1 1 0 0	014
O	9	1 1 0 0 0	030
P	0	1 0 1 1 0	026
Q	1	1 0 1 1 1	027
R	4	0 1 0 1 0	012
S	Bell	0 0 1 0 1	005
T	5	1 0 0 0 0	020
U	7	0 0 1 1 1	007
V	;	1 1 1 1 0	036
W	2	1 0 0 1 1	023
X	/	1 1 1 0 1	035
Y	6	1 0 1 0 1	025
Z	"	1 0 0 0 1	021
SPACE		0 0 1 0 0	004
CAR. RET.		0 1 0 0 0	010
LINE FEED		0 0 0 1 0	002
NULL		0 0 0 0 0	000
FIGURES		1 1 0 1 1	033
LETTERS		1 1 1 1 1	037

In the BAUDOT table shown above, the octal codes column was shown assuming that the codes were stored in the least significant bit positions of an 8008 register with the three most significant bits set to 0. The reader can now see that 26 of the possible 32 codes can represent two different characters depending on which mode the machine is in. The functions SPACE, CARRIAGE-RETURN, LINE-FEED, and NULL mean the same regardless of which mode the machine is in, and two codes FIGURES and LETTERS are used to switch the mode of the machine. While everything may seem fine at this point, it is important to discuss handling the code as part of an I/O routine because there is a subtle factor that can be over-looked by some beginning programmers!

In actual operation, a BAUDOT machine

operates in the mode that it was last placed in by a figures or letters key, and remains in that mode until the opposite mode code is received. Thus, a mechanical arrangement actually serves to remember a bit of information. The fact that an external mechanical linkage is used to hold a bit of information must be taken in account if a computer program is to process the code with practical results!

For instance, if one had an input routine that simply looked for a five bit pattern from a BAUDOT device, one could get that pattern in many instances from two possible conditions of the machine. For instance, if the operator typed an 'A' or an '-' mark, and the program was designed to perform a certain function on receipt of the letter 'A,' it would also perform it if the punctuation '-' was received! To avoid that happening, one might inform the human operator to always enter information during that part of the program with the machine in the letters mode, but that is not the safest way in which to design a program.

Instead, one would be better off to add a bit to the BAUDOT code when it was manipulated in the computer that would serve to differentiate between letters and figures. For instance, the code 000011 could be used to indicate the letter 'A,' and 100011 to indicate the punctuation '-' mark. In order to institute this method, one would have to have a program that kept track of which mode the machine was operating in whenever it was receiving data from the machine by remembering the last letters or figures code received. Furthermore, in order to ensure that the mode was properly received (such as when the program was first started, or power turned on the machine), it would be wise to have the computer output a command that would place the machine in a known state such as would be accomplished by outputting a letters or figures code at the start of such operations. Then, for storage and manipulation in the computer, the input routine could set a sixth bit to a '1' condition whenever a code was received while the machine was in, say, the figures mode, and leave the sixth bit as a '0' when codes were received in the letters mode. The six bit codes could then be manipulated and stored by the program in much the same manner as one might process ASCII codes with the ability to immediately recognize the close to 60 different characters. When it was desired to output information, the sixth bit would be used to indicate whether it was necessary to first output a figures or letters code to set the machine in the proper mode. (It would not be necessary to output a figures or letters mode command before every character was sent because one could use an algorithm that would only send a mode command when the sixth bit was noted to have changed from that present when the previous character was transmitted.)

Two sample routines for performing such a function, one for inputting data from a BAUDOT machine, and one for outputting data to such a machine, will be illustrated here.

```
BAUDIN,   LAI 037       Load letters code into accumulator
          CAL OUTPUT    Call routine to send BAUDOT character
          CAL LETCOD    Initialize register B to letters
INBAUD,   CAL INPUT     Now accept BAUDOT characters from machine
          CPI 033       See if figures code
          CTZ FIGCOD    Go set up '1' as sixth position bit
          CPI 037       See if letters code
          CTZ LETCOD    Go set up '0' as sixth position bit
          ADB           Add in status of sixth bit position
STORBD,   CAL MANIP     User subroutine to process data
          JMP INBAUD    Get next character in sequence if applicable
```

```
    FIGCOD,  LBI 040        Set sixth bit in B = 1
             RET            Return to main subroutine
    LETCOD,  LBI 000        Set sixth bit in B = 0
             RET            Return to main routine
```

The reader should note that there are actually two entry points to the routine just presented. The subroutine BAUDIN should be called to initialize the condition of the BAUDOT machine whenever the program is first started, or at other times when the mode of the machine is not certain. Once the machine and routine have been initialized, then the program may be called at INBAUD as long as some other routine does not interfere with the status of register B. The reader who is interested in logic might note that register B in the above program acts as a flip-flop to remember the mode in which the typewriter is operating.

The routine shown next also has two entry points. The first termed BAUDOT is used when the first character of a string of characters is to be outputted in order to initialize the BAUDOT machine, and set up register C. The entry point OTBAUD may then be used until the mode memory register (C) is interfered with by any other external routine. Note too, that the routine below expects the character to be outputted to be residing in register B when the subroutine is called!

```
    BAUDOT,  LAI 037        Load letters code into accumulator
             CAL OUTPUT     Call routine to send BAUDOT character
             LCI 000        Set indicator for letters in C
    OTBAUD,  LAB            Move character from B to accumulator
             NDI 040        See if sixth bit = 1, if yes = figures
             JTZ LTCHAR     Character, if not = letters character
             NDC            If figure see if last out also figure
             JTZ LASLET     If 0 here then last was a letters
    OUTCOD,  LAB            Put present character in accumulator
             CAL OUTPUT     Send the BAUDOT character
             RET            Return to calling routine
     LASLET, LAI 033        Since last was letter put figures code
     LASFIG, CAL OUTPUT     Send code
             LCB            Save latest in register C for comparison
             JMP OUTCOD     Send current character
    LTCHAR,  LAI 040        Set mask and see if last was letters
             NDC            By comparison of sixth bit position
             JTZ OUTCOD     If 0 here, last was also letters
             LAI 037        If not, send letters code first
             JMP LASFIG     By using above routine to send letters code
```

It is often desirable to have I/O routines that will convert between one type of I/O code and another, such as between ASCII and BAUDOT. This may be desired for a number of reasons. For instance, because one has had one type of input device using one code, and a different output device using another code. Or, one might desire to use a particular program that was written to use one kind of code, with a machine that used a different kind of code, without having to modify a lot of locations in the original program that might have been testing for specific I/O codes from an external device. In such cases, the computer's capability to perform conversion functions is readily capitalized upon by constructing a lookup table and using a suitable program to convert from one code to another.

For example, suppose it was desired to use a BAUDOT machine with a program that was developed originally to operate with a machine that used ASCII code. One could proceed to first construct a lookup table similar in format to that shown here:

ADDRESS	CONTENTS	COMMENTS
10 000	301	A (ASCII)
10 001	003	A (BAUDOT)
10 002	302	B (ASCII)
10 003	031	B (BAUDOT)
. .	.	. .
. .	.	. .
. .	.	. .
. .	.	. .
10 076	240	SPACE (ASCII)
10 077	004	SPACE (BAUDOT)
10 100	241	"!" (ASCII)
10 101	015	"!" (BAUDOT)
. .	.	. .
. .	.	. .
. .	.	. .
. .	.	. .
. .	.	. .
10 174	277	"?" (ASCII)
10 175	071	"?" (BAUDOT)
10 176	300	"@" (ASCII)
10 177	000	Substitute null (BAUDOT)

In constructing the table, one could elect to leave out or ignore characters that were not represented by both codes, or to substitute a substitute character when one code does not have an equivalent character. Either method requires consideration when the search routine is developed. The former method leaves the possibility that a human operator might type in a character that did not exist in the table, and so the programmer would have to be careful to limit the table search routine. Note that if every possible entry existed in the table, then the table search routine will be self limiting in that a match will always be found. On the other hand, the latter choice of using a substitute character requires that the table be organized so that the preferred character for cases of multiple substitution will be the one found first by the table lookup routine. For instance, there are several characters besides the @ mark, such as ']' and '[' which could be included in the above table which are represented by ASCII codes but not BAUDOT codes. If one decided to include them in the table, but have NULL characters as their conversion equivalent, one can see that a problem arises when one uses the same table to convert from BAUDOT to ASCII, as now there are several places in the table that have the NULL code. As will be clear shortly, the routine that converts from BAUDOT to ASCII will always represent a NULL character in BAUDOT as a '@' symbol in ASCII because the BAUDOT routine searches the table from highest address to lowest, and will find the NULL to '@' entry first. Naturally, the table could be re-organized so that some other NULL conversion entry was located first. Or, a different type of lookup routine than the one to be presented can be developed. These factors are simply being pointed out to increase the reader's awareness as to the types of factors that must be considered when performing such operations.

A routine that will use the lookup table to convert ASCII characters to BAUDOT is illustrated next. This program, and the BAUDOT routine discussed earlier could be used to output characters from a program that was actually doing internal processing with ASCII codes.

```
 ASBAUD,   LHI 010       Set page address pointer to location of table
           LLI 000       Set low address pointer to top of table
  FASCII,  CPM           Compare (ASCII) code in accumulator to contents
           JTZ FNDBDO    Of table, if match, do conversion
           INL           Otherwise advance low address pointer
           INL           To next ASCII code location in table
           JMP FASCII    And keep looking for a match
 FNDBDO,   INL           When have ASCII match, advance pointer 1 location
           LAM           And fetch BAUDOT equivalent into accumulator
           RET           Exit lookup routine
```

The above routine assumes that the code (in ASCII) for a character that exists in the table is in the accumulator when the routine is entered. Note that the routine does not test for the end of the table because of that assumption. If for any reason it might be possible for a code to be in the accumulator that was not in the table, then it would be necessary to add an end of table test each time the table pointer was advanced, and to take appropriate action if no match was found in the table.

The next routine does essentially the reverse process, using the same table, to convert BAUDOT codes to ASCII codes. It could be used along with the previously described BAUDIN routine to accept characters from a BAUDOT machine and convert them for use in a program that utilized ASCII codes. As in the above routine, the program assumes that a valid BAUDOT code is in the accumulator when the routine is called. Note that the routine starts searching the table in the opposite direction than the routine presented above.

```
 BAUDAS,   LHI 010       Set page address pointer to location of table
           LLI 177       Set low address pointer to bottom of table
 FBAUDO,   CPM           Compare (BAUDOT) code in accumulator to contents
           JTZ FNDASC    Of table, if match, do conversion
           DCL           Otherwise decrement low address pointer
           DCL           To next BAUDOT code location in table
           JMP FBAUDO    And keep looking for a match
 FNDASC,   DCL           When have BAUDOT match, decr pointer 1 location
           LAM           And fetch ASCII equivalent into accumulator
           RET           Exit lookup routine
```

Naturally, the techniques illustrated to convert between ASCII and BAUDOT codes may be applied to many other types of codes. Indeed, the small computer makes an ideal device for coupling between a variety of I/O devices, particularly in communication applications, thus enabling machines with different characteristics and using different codes to communicate with one another.

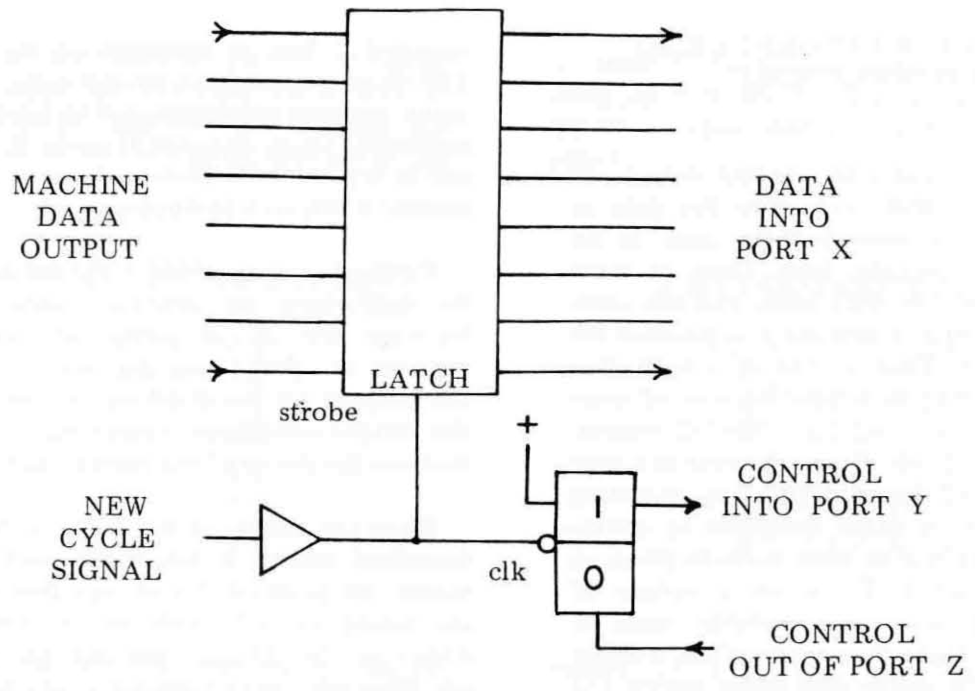
A concept that will be discussed more fully in the next chapter will be briefly mentioned at this time to point out an important concept when dealing with I/O devices connected to the computer. As the reader undoubtably knows, many machines that might be connected to a computer are much slower in operation, in fact often times orders of magnitude slower, than the basic operating cycle of a computer. For instance, an 8008 system requires but a mere 32 millionths of a second in a typical system to execute an input instruction. That is, in that short amount of time it can access an input port and bring in 8 parallel bits of information into the accumulator of the CPU.

The extreme speed of the computer can in fact cause problems when performing I/O operations if steps are not taken to control the situation. Assume, for example, that a person desired to connect an electronic keyboard unit, similar to a typewriter, that would present the ASCII code for the key being depressed in parallel on the lines of an input port. If the person just connected the keyboard output lines to the input lines of an input port, and wanted to develop a program that would accept information from the keyboard, there would be a number of rather tough problems, and they would be related to the speed at which the computer can operate relative to the speed at which a human can depress the keys on a keyboard.

Suppose that the keyboard was directly connected to an input port, and a programmer tried to develop a routine that would simply read the code being sent by the keyboard, store the character in memory, and go on to read the next character. In the first place, how would the program be able to even tell if a key had been depressed? True, one could assume that if no keys were depressed the code being received would be all zeros, and a program could check for that condition. But, even if that was done, the programmer would soon have another problem. When a key was actually depressed, and a non-zero condition received, a short program to place the character in memory and advance the memory pointer would be accomplished in the order of a hundred-millionths of a second. The poor human depressing the key wouldn't have a chance of getting a finger off the depressed key in that amount of time, and in fact it would take on the order of several tenths of a second for a person to remove a finger from a key. In that amount of time, the simple input routine could have read that same character and packed it into memory locations a few hundred times! Not exactly the desired result. What now? Well, one could develop the input algorithm so that, once a non-zero code was received, one would not accept another character until a zero code was observed. That might improve things somewhat, but it would preclude actually being able to receive a zero code (that might represent a valid condition) and, because of technical considerations (such as contact bounce on the mechanical switches of the keyboard) it would not be a very reliable method to utilize.

Instead, it would be far better to place an interface between the keyboard and the computer input port that would accomplish the following objectives. Whenever a key on the keyboard was depressed, the interface would latch (hold) the code represented by the key in an electronic buffer that was connected to the lines of an input port. The buffer would thus hold data from the keyboard. Next, when the key that had been depressed was released, the interface would present a signal to an input line of another input port, termed a control port. Finally, the interface would have a line coming from an output port of the computer that would allow the computer to signal to the interface that it had taken appropriate action. A diagram of an electronic interface with the characteristics described is shown in the next illustration.

With such an interface, one could develop a much more reliable system using an input program that would perform in the manner illustrated after the diagram.



```
MACHIN,  INP Y        Check status of control from machine
         JFS MACHIN   If data not ready then wait by looping
         INP X        Data ready now so fetch DATA
         LBA          Save DATA in register B
         LAI 001      Prepare to pulse line on PORT Z
         OUT Z        Send a logic one on PORT Z control line
         XRA          Clear accumulator
         OUT Z        Send logic zero on PORT Z control line
         LAB          Restore data to accumulator
         RET          Exit routine with DATA in accumulator
```

The above routine assumed that the control line from the interface came into the most significant bit of the accumulator and that the control line going to the interface originated from the least significant bit in the accumulator. Furthermore, while the above routine waited for new data to arrive from the external device by monitoring the input control port continuously, the JFS MACHIN instruction could have been replaced by a directive to have the computer perform some other function(s) before testing INPUT PORT Y again instead of wasting time doing nothing!

A similar type of interface, and similar programming techniques can be applied to a wide variety of devices that might be connected to the computer. While the example showed but one line being used on each control port, one should note that with eight lines available on one port, one can use just a few control ports in a system to monitor and control a large group of external instruments by using the available bit positions.

TESTING FOR ERRORS DURING I/O OPERATIONS

It is often desirable to transmit data to an external device that will store the data in some sort of permanent form, such as on paper tape or magnetic tape. Then, at some later time, read the data back into the computer. During such a process it is possible for errors to occur. That is, bits of information within a word may be altered because of noise or random errors occurring in the I/O system. While such errors are likely to occur at a very low rate in a well designed, properly operating I/O system, it is often desirable to utilize techniques that will at least indicate when an error has occurred. There are a variety of error checking techniques available, some so sophisticated that they can often correct certain types of errors that occur during I/O operations. Two techniques will be discussed here. While neither one of them has error correcting capability, they are capable of detecting the most common type of I/O error which is for a bit in a word changing state.

The first method to be discussed concerns the use of using parity techniques to detect transmission errors. The technique consists of examining a group of bits for the number of bits that are in the '1' condition when it is being readied for transmission, and then setting a bit aside for the purpose to the state that will make the total number of bits that are in the '1' condition either an odd or even count (for the entire group). For instance, it was mentioned earlier that the ASCII code required 7 bits to represent all the possible 128 characters defined by the code, but that many systems employed an 8'th bit for parity purposes. Thus, the ASCII code is ideal for use in typical 8008 systems because there are exactly 8 bits to a computer word.

Furthermore, the 8008 CPU has as part of its instruction set specific instructions to facilitate the use of parity techniques. Remember the parity flag that was discussed in the chapter on the 8008 instruction set, and the various conditional branching instructions that use the status of the parity flag?

When the codes for the ASCII subset were described earlier, it was mentioned that the eighth bit position (most significant bit) in the listing was arbitrarily set to the '1' condition as the ASCII code did not use that bit. However, that bit position may be used to specify the desired parity in a system where parity checking is to be employed. For instance, if one wanted to establish an even parity system, one would proceed in the following manner.

Examine the seven bits making up the code for the character to be transmitted (assuming ASCII code for this example). If the number of bits in the character that are a logic '1' are even, that is there are 0, 2, 4, or 6 bits in the '1' state, set the 8'th bit to a '0.' If the number of bits are odd, that is there are 1, 3, 5, or 7 bits in the '1' state, set the 8'th bit to a '1' condition so that the total number of bits in the entire group becomes an even number! Some examples are illustrated below.

	ORIGINAL 7 BIT ASCII CODE	8 BIT EVEN PARITY CODE
(A)	1 000 001	01 000 001
(B)	1 000 010	01 000 010
(C)	1 000 011	11 000 011
(D)	1 000 100	01 000 100
(E)	1 000 101	11 000 101
(0)	0 110 000	00 110 000
(1)	0 110 001	10 110 001

One could also elect to use an odd parity system by essentially reversing the scheme so that the 8'th bit is always set to make the total number of bits in a group that are in the '1' state to be an odd number. ASCII code using an 8'th bit to produce an odd parity system is illustrated below for several characters.

	ORIGINAL 7 BIT ASCII CODE	8 BIT ODD PARITY CODE
(A)	1 000 001	11 000 001
(B)	1 000 010	11 000 010
(C)	1 000 011	01 000 011
(D)	1 000 100	11 000 100
(E)	1 000 101	01 000 101
(0)	0 110 000	10 110 000
(1)	0 110 001	00 110 001

Once one has selected which parity (odd or even) to use with a system, one simply sends the data in the desired mode to the I/O device. Then, when the data is later read into the computer, a check is made on each word of data received to determine if the parity is correct. If it is not, then an error has occurred. Sample routines to generate even parity words for an output routine, and for checking for even parity in an input routine are shown next.

```
SEVENP,  NDA          Assume 7 bit ASCII code in accumulator, 8'th bit
         JTP GOUT     Init 0, if parity even as is, send data
         XRI 200      Otherwise set MSB = 1 to get even parity
  GOUT,  CAL OUTPUT   User routine to transmit data to I/O
         RET          Exit even parity generator routine
REVENP,  NDA          Assume data from I/O device in accumulator
         RTP          Set flags, if even parity, all O.K.
         JMP PERROR   If not even parity do user error routine
```

Similar routines are easily developed for utilizing odd parity. The programmer should note that parity techniques can be used with virtually any coding technique as long as one bit is set aside for the parity indicator. For instance, one could easily adapt parity techniques for the BAUDOT code discussed earlier provided that the I/O device could handle the extra bit. That might not be possible with a BAUDOT coded machine but it might be applicable, say, if BAUDOT code was being written on a magnetic tape unit where extra bits could be added to the code and processed by the I/O unit.

The reader should also be aware of the fact that the use of parity checking techniques is not infallible. It does detect errors that result in an odd number of bits changing state within a group, but not if an even number of state changes occur. It it thus most useful in a system where the expected probability of more than one error occurring in a group of eight bits is extremely low. The programmer might also want to consider when using a parity checking scheme the possibility of transmitting each group of bits twice. Then, when data is read back from the I/O device, an algorithm that will skip the second group if the group is received correctly the first time, or read the second group if an error was detected in the first group, can be utilized. Such a format, while requiring a longer transmit and receive time, can result in highly reliable I/O data handling operations.

Another error checking method that is often used when passing data to and from I/O devices is termed the check-sum technique. The method is quite simple in application, yet remarkably powerful in detecting errors. The technique consists of simply maintaining a one register sum of all the data transmitted within a block. That is, as each word is sent out, it is summed with a register that contains the sum of all previous data words transmitted in the block. (Over-flows in the summing register are ignored.) At the end of a block of data, the two's complement of the sum that has been compiled is sent as the final piece of data in the block.

When the block of data is read back into the computer a similar sum is formed as each data word is received. Then, when the last piece of data is received, which is the two's complement of the check-sum, that value is added to the sum obtained from all the previous data words in the block. The result, if no transmission errors have occurred, will be zero, the result of adding any number to its two's complement. If it is not zero, then a transmission error has occurred. The method is simple and quite reliable. The reader can readily determine that if errors have occurred it will affect the value of the sum as it is formed, and thus likely result in a non-zero value as a final result when the check-sum and its two's complement are added. (Note: It is theoretically possible for just the right number of errors to occur when reading a block of data to result in a zero condition, but it is quite small, hardly enough to lose sleep over!)

A routine for generating a check-sum and placing the two's complement of that value as the last word sent in a block of data followed by a routine that will read back a block of data using a check-sum technique and test to see if any errors occurred is shown below.

```
SCKSUM,   LHI XXX       Set page address where block of data stored
          LLI YYY       Set location on page for start of data block
          LEI ZZZ       Set number words in block counter
          LDI 000       Set check-sum register to 0 at start
NXCKSM,   LAM           Fetch data word from memory
          ADD           Add present data to check-sum value
          LDA           Save new check-sum value
          LAM           Restore original data word from memory
          CAL OUTPUT    Output the data word to I/O device
          INL           Advance memory pointer
          DCE           Decrement word counter
          JFZ NXCKSM    If counter not 0, fetch next data word
          LAD           Put check-sum value in accumulator
          XRI 377       Form two's complement value
          ADI 001       In standard manner
          CAL OUTPUT    Send two's complement of check-sum as last
          RET           Word in block and exit routine
RCKSUM,   LHI XXX       Set page address where block of data goes
          LLI YYY       Set starting location on page for data
          LEI ZZZ       Set number words in block counter
          LDI 000       Set check-sum register to 0 at start
INCKSM,   CAL INPUT     Fetch data from I/O device
```

```
LMA           Store data word in memory
ADD           Add new data to currect check-sum value
LDA           Save new check-sum value
INL           Advance memory pointer
DCE           Decrement word counter
JFZ INCKSM    Get next data word if counter not 0
CAL INPUT     Next word from I/O is two's complement of check-sum
ADD           Add it to check-sum formed by data
RTZ           If result is 0, O.K., exit subroutine
JMP CKSMER    Otherwise go to user error routine
```

The above routines, as the reader will note, assume that data blocks are one page or less in length, and do not cross page boundaries. However, by this time the reader should have little difficulty writing a check-sum routine that could handle larger blocks.

The next chapter will contain more information of interest to those developing programs for I/O operations that require consideration of real-time parameters.

Real-time programming as discussed in this manual applies to the development of programs whose proper execution are dependent on the length of time it takes for the computer to perform an operation or series of instructions. The need for real-time programming is invariably related to the receipt of information from devices at specific times or the control of devices external to the computer whose proper operation depend upon receiving commands from the computer at specific times.

The discussion of the subject of real-time programming has been deferred to the latter portion of this manual as real-time programming is generally more difficult than the development of programs that are not restricted by execution times. The reason is simply that in addition to the logic and technique factors that the programmer must consider when developing any program, the programmer must now add in the factor of how much time it will take for the computer to execute various instructions and instructional sequences. The problem is really one of increased complexity in the program development task.

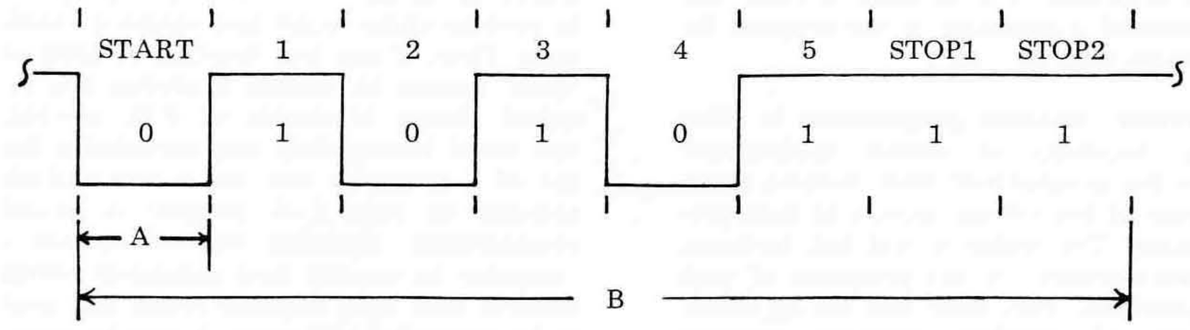
However, real-time programming is often vitally necessary in certain applications. Hence the programmer must become aware of some of the critical aspects of such programming. The reader should not, however, be over-whelmed by the prospects of such complications. For, once one has an understanding of standard machine language programming procedures and has gained a little experience, which one should have obtained by the time one is delving into this section, one should find the aspects of real-time programming simply one step up in difficulty and an enjoyable challenge.

As with many other aspects of programming, proper preparation, such as clearly defining the problem to be handled, and proceeding in an orderly fashion, using methods already discussed, can greatly ease the overall task of developing real-time programs.

The last several pages of Chapter One presented the typical execution times for the various classes of instructions available in an '8008' based microcomputer. The times shown are those for an '8008' unit whose master clock has been adjusted to a nominal frequency of 500 kilohertz. When getting down to practical applications, one must realize that any system will have some finite deviation from the nominal frequency. For instance, if an '8008' system has a crystal controlled master clock, the possible variation from the nominal frequency might be in the order of 0.05 to 0.1 percent. Some '8008' systems might have resistor-capacitor controlled master clocks and the possible deviation from the nominal frequency could be considerably more, up to four or five percent. In any event, when contemplating the development of real-time programs, one must always take into account the possible variation from nominal of the master clock frequency. In fact, one should plan programs to operate under worst case variation conditions. Thus, if one was thinking of using an '8008' system to control a process that required timing accuracies of 0.01 percent, one could immediately stop considering the use of a computer that had a master clock accurate to only 0.05 percent! A second consideration regarding whether to use a computer to control time dependent events involves how close together events that need to be controlled will occur. It may be observed by examining the information at the end of Chapter One, that almost all instructions require a minimum of 20 microseconds to execute in an '8008' system. Thus, one cannot plan on using such a computer to control events that are less that that far apart in time. In fact, because I/O operations themselves take 24 to 32 microseconds, and because those instructions would invariably be required to be in use when dealing with external devices, along with the fact that one will almost certainly want to use some other instructions between I/O commands, it is a pretty good rule of thumb to disqualify the use of an '8008' based system as a real-time controller if any two events dependent upon timing from the computer will occur within 100 microseconds. A second rule of thumb is to immediately reject the use of such a system as a real-time controller if the application will require much more than one thousand I/O operations per second. Unless such operations are strictly repetitive and the previous rule (events are at least 100 microseconds apart) can be met. This second rule of thumb is derived from practical experience with programming overhead that results when a variety of time-dependent events must be juggled in a real-time program.

The prospective real-time programmer should become familiar with the lengths of time required to execute the various classes of instructions. One of the first new habits to learn when preparing real-time programs is to write down the execution time required for each instruction along side of the mnemonic as the program is written. It then becomes an easy matter to figure out totals for various portions of the routine. Additionally, it is often helpful to write down the total execution times along paths and loops on a flow chart of the program. Real-time programming often requires a fair amount of juggling between choices of instructions used and alternate sequences of commands in order to obtain desired program execution times. Having critical timing information on hand in the forms suggested can provide the programmer with a quick view of how the program development effort is proceeding.

In any programming application, flow charting is an extremely valuable aid to enabling one to obtain an overall view of a program's operation. In real-time programming another tool of equal importance should be brought into use. That tool is a TIMING DIAGRAM. A timing diagram illustrates the relationships in time between the occurrences of specific events of interest to the programmer.



TIMING DIAGRAM FOR SENDING BAUDOT CHARACTER 'Y' OR '6' TO PRINTER

A timing diagram is illustrated above. The diagram indicates the desired status of a signal line as a function of time for an electronic signal that is to provide information to a BAUDOT coded typing machine. The diagram shows the signal conditions required to direct the machine to print the letter Y or the figure '6' depending on which mode the machine is operating in (LETTERS or FIGURES) at the time the code is sent. This diagram will be used to develop a sample program for operating a BAUDOT printer mechanism as an introduction to the considerations required when dealing with real-time programming.

In order to clarify the diagram a brief

explanation of the operation of a BAUDOT coded printing mechanism will be presented. The printing mechanism is assumed to be an ASYNCHRONOUS device in that it requires START and STOP information. Once the printing mechanism has begun a cycle of operation as the result of receiving a START signal level, the machine will examine the status of a signal line during specific time periods in order to receive a CODE that will enable it to print a specific character. At the end of a machine cycle the machine expects to see a STOP signal. The STOP signal must last for a certain amount of time so that the machine may complete various mechanical operations and reset itself in order to be ready to receive more signals and commence a new cycle. A CYCLE in this context shall mean a certain number of units of time. The TIMING DIAGRAM just illustrated shows a cycle that is divided into eight equal units of time. The first unit of time is reserved for a START pulse. By definition in this example, the start pulse is a logic zero level as shown in the diagram. The next five units of time in the cycle are used to transmit the BAUDOT code for whatever character is to be printed by the machine. The last two units of time are defined to be a logic one level to serve as STOP pulses. This information is summarized in the timing diagram. To put the diagram to practical use, one must define the length of a unit of time in the diagram! For instance, suppose one had a printing mechanism that was designed to operate correctly when each unit of time (the length of time denoted by the distance labeled A in the above diagram) was 20 milliseconds (nominally). An entire cycle would thus require 160 milliseconds (the time span marked off by the distance labeled B in the diagram).

If it was desired to have the computer send a signal on an output line that closely approximated the desired signal pattern, one would have to develop a program that would change the state of the line on an output port that was supplying the signal to the machine at the times indicated by the marker lines in the diagram (where the signal changes state). Such a program would be a real-time program!

Real-time programming for this type of application is relatively straightforward. First of all, there is only one signal line to be concerned with (in many real-time applications there may be a multitude of lines to control)! Secondly, the amount of time between events is quite large so that there will not be any requirement for fancy programming that has to be streamlined for maximum speed of operation. In fact, all one really has to do is make some simple mathematical calculations and develop some TIMING LOOPS that will make the program wait for the desired length of time between sending bits of information to the output port that will carry the signal to the typing unit. The program becomes simply a little fancier version of the PARALLEL TO SERIAL output program discussed in the previous chapter.

A suitable program is presented below. A discussion will be presented after the program. Note now that the execution times have been provided alongside the time-dependent portions of the program.

```
                     BDOUT,   LCI 006      Set bit counter = number of bits + 1
                              NDA          Set carry bit equal to '0'
                              RAL          Bring '0' from carry into LSB of ACC
24                   MORBDO,  OUT X        Send START or CODE bits to machine
20                            RAR          Position next bit of information
44 + 19,848                   CAL BDELAY   Give machine one unit of time
20                            DCC          See if finished START and CODE bits
44 / 36                       JFZ MORBDO   If not, send next bit
32                            LAI 001      Prepare to send,STOP bits
24                            OUT X        Send STOP bit number one
```

```
44 + 19,848                   CAL BDELAY   Give machine one more unit of time
44 + 20                       CAL DUMMY    Provide a little more time
44 + 20                       CAL DUMMY    Provide a little more time
24                            OUT X        Send stop bit number two
44 + 19,848                   CAL BDELAY   Give machine one more unit of time
44 + 20                       CAL DUMMY    Provide a little more time
44 + 20                       CAL DUMMY    Provide a little more time
                              RET          Exit outputting a character routine

20                  DUMMY,    RET          Short routine to eat up time

32                  BDELAY,   LDI 215      Set timer loop counter
24                            OUT Z        Output to unused port to trim time
24                            OUT Z        Output to unused port to trim time
44 + 20                       CAL DUMMY    Use a little more time B4 starting loop
44 + 20             MDELAY,   CAL DUMMY    Form a time consuming loop
20                            DCD          See if time expired (counter = zero?)
12 / 20                       RTZ          Exit back to calling routine when done
44                            JMP MDELAY   Otherwise continue using up time
```

The above routine assumed that the data to the printing machine originated from the least significant bit in the accumulator.

The reader should note that for cases where there are two possible execution times for an instruction, such as a conditional instruction, that the time required for the condition most often occurring in the program was shown first, followed by the time required when the other condition occurred.

The program was initially developed by writing the main portion with the time required for the BDELAY subroutine considered as an unknown factor. When the basic format of the program had been determined the execution time of the loop starting at the label MORBDO which included the five instructions:

```
MORBDO, OUT X
        RAR
        CAL BDELAY
        DCC
        JFZ MORBDO
```

was calculated, leaving out the as yet undetermined time of BDELAY. The time required by the five instructions when looping was found to be 152 microseconds. Since it was known that a total of 20,000 microseconds (20 milliseconds) was desired between outputting each bit in the code it was then easy to calculate that:

$$20{,}000 - 152 = 19{,}848$$

microseconds delay was required in the subroutine BDELAY.

The subroutine BDELAY is a typical example of a timing delay loop. The main portion of the delay loop starts at MDELAY and includes the four instructions:

```
MDELAY, CAL DUMMY
        DCD
        RTZ
        JMP MDELAY
```

The theory behind the BDELAY subroutine was to execute the MDELAY loop the required number of times to get close to a delay of 19,848 microseconds and then close any gap by the setup instruction(s) for the loop.

The time required to complete the four instructions in the MDELAY loop when the RTZ condition is not met is 140 microseconds. Finding out how many times it is necessary to execute the loop to get close to a delay of 19,848 microseconds is a simple matter of dividing. Doing so yielded a figure of almost 142 (decimal). Taking into account the fact that it was not desirable to go over the alloted time, and the fact that setting up the loop would take some time, the figure of 141 decimal was chosen, which is 215 octal. One other factor had to be considered. When the counter in the loop reached zero, the RTZ instruction would be executed and the JMP MDELAY command would not. Thus, the full loop would only be executed 140 (decimal) times. The last time through the MDELAY routine would only take 104 microseconds. Thus, at this point it was possible to calculate the total delay caused by executing the MDELAY loop the selected number of times: 140 X 140 = 19,600 plus 104 for a total of 19,704 microseconds. Then it was an easy matter to determine how much time to use to setup the MDELAY subroutine. The desired total delay of 19,848 minus the 19,704 microseconds consumed by executing the MDELAY routine 141 (decimal) times left 144 microseconds to be consumed. The LDI 215 at the start of BDELAY only required 32 microseconds so 112 more microseconds were consumed by adding the filler instructions CAL DUMMY and two OUT X commands. The total BDELAY subroutine then was exactly equal to the desired delay time of 19,848 microseconds!

After sending the START and five CODE bits it was necessary to send a two unit STOP pulse. Since the STOP pulse by definition was to be a logic one, it was necessary to setup the stop bit as a one in the accumulator. The reader can calculate that the actual delay between the sending of the last CODE bit and the first STOP unit in the routine comes out to be 20,024 microseconds. Remember in making the calculation that the JFZ MORBDO instruction will only require 36 microseconds on the final execution of the loop thereby reducing the loop execution time to 19,992 microseconds and the LAI 001 will add 32 microseconds to that value before the next OUT X instruction can be executed. However, for the application, the value of 20,024 is plenty close enough to 20,000 (off by about 0.1%) to operate a mechanical machine which can typically operate reliably with the timing off as much as 10 to 20 percent!

The delay between the first stop unit and the second, as well as the final delay to complete the second stop unit, was made to come out nicely to 20,000 microseconds by the insertion of the CAL DUMMY commands following the CAL BDELAY instructions.

The routine just presented, as the reader can undoubtably see, could be modified to serve a variety of electro-mechanical printing machines operating at different speeds by changing the timing loops. The program could also be modified to serve ASCII coded machines, or other types of codes by changing the bit counter and possibly altering the length of the STOP pulse(s) depending on the type of machine being driven. Furthermore, the techniques demonstrated can be applied to many other types of problems.

A similar routine could be developed to receive data from the same kind of BAUDOT machine. However, when receiving data from such a unit there are a few new concepts to consider.

When the computer was sending information to the printing mechanism it had an advantage it will not have when it is used to receive information from the machine. Namely, when transmitting, the computer had control of when the external machine would be operated. In the reverse mode, the computer will have no knowledge of when the external device will begin to operate and transmit data to the computer!

Additionally, once a character starts arriving on a line of an input port, the tolerance situation reverses. What is meant by this is that when the computer sent data to the printer mechanism, it was possible for the computer to be much more accurate in providing proper timing to the machine, than the machine required to operate successfully. Thus, if the time period for a unit of time was off a few tenths of a percent when generated by the computer, it would not affect the operation of the machine. However, when the computer is receiving data from the machine, the start of each unit of time may be off by as much as 10 percent because of the loose tolerance of the electro-mechanical machinery. If the computer program does not make proper allowances for such possible variations, then incorrect data may be received.

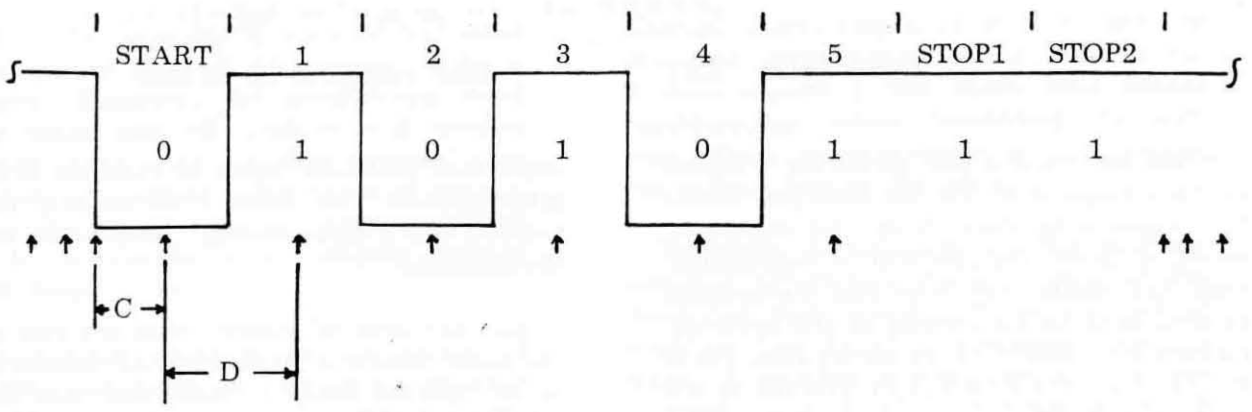
Fortunately, the problems related to these concepts are not too difficult to overcome. The first problem, determining when the external machine is starting to send, can be solved by periodically checking the input line for the presence of a zero logic condition indicating a START bit. (Note: while there is another manner in which one could detect the beginning of an external operation in a properly equipped microcomputer system, through the use of a hardware generated INTERRUPT scheme, such a method is more properly concerned with hardware considerations which are not within the intended subject matter of this manual. If such a detection scheme were used, the remainder of this discussion on handling the receipt of the incoming data would still apply.) Naturally, how often one checked for the presence of a START bit would have an affect on the overall ability of a real-time program to receive the data. For instance, assuming a START bit is present for 20 milliseconds as in the case for the hypothetical machine being discussed, it would be foolish to test for the presence of such a start bit at periods that were 21 milliseconds apart! In fact, because of other considerations, it would not be wise to check for a START bit much less often than every few milliseconds.

The second problem of dealing with the loose tolerance of the machinery can be effectively dealt with by adjusting the receive routine so that it SAMPLES the incoming signal at the theoretical middle of a unit of time rather than at the beginning or end of a time period. Of course the ability to do this also depends on how closely one is able to detect the actual start of a character as it is sent by the machine.

A timing diagram showing a BAUDOT character being sent by a machine is illustrated at the top of the next page. Short upward point arrows along the bottom of the diagram illustrate the times at which a real-time program would need to sample the incoming line in order to correctly receive the data. Note that prior to the time a START signal is detected, the computer should sample the line often in order to minimize the period of time in which a START signal might be present but undetected. Next, it is desirable to adjust the sample period so that it coincides with the theoretical middle of a unit of time, rather than sample at integers of units of time after the start signal was detected. This method compensates for the tolerance problem mentioned previously.

Finally, after the fifth code bit has been received, one may observe that it will not be necessary to start testing for a new start pulse for about two and a half time units as it is known that the machine will be using that time to complete its operation cycle. Thus, the computer would be able to perform some other functions for about 50 milliseconds before going back to the SAMPLE mode to look for a new START bit. That is enough time to perform a few thousand or more instructions in a typical microcomputer system!

A sample routine for receiving information from a device in accordance with the timing diagram illustrated, assuming that the time span marked C in the timing diagram was 10 milliseconds, and that marked D was 20 milliseconds, is illlustrated following the diagram. The reader may not that it is essentially an expanded version of a SERIAL TO PARALLEL routine .



TIMING DIAGRAM FOR RECEIVING BAUDOT CHARACTER 'Y' OR '6'

```
                         BDIN,    LBI 000        Clear incoming storage register
                                  LCI 005        Set bit counter
32                     STRTIN,    INP X          Look for START bit
32                                NDI 200        Mask off irrelevant data
44 / 36                           JTS STRTIN     If no START bit, form sampling loop
44 + 9796                         CAL HDELAY     If find logic '0' assume start, delay
32                                INP X          To middle of START unit & verify
32                                NDI 200        By making appropriate test
36 / 44                           JTS STRTIN     If not '0' here assume false START pulse
44 + 20                           CAL DUMMY      Stretch the delay a little
44                                JMP MORBD1     Stretch the delay a little more
44 + 19748             MORBDI,    CAL IDELAY     Main delay loop, almost 1 full time unit
32                                INP X          Get next bit
32                                NDI 200        Trim to just desired data
20                                RAL            Save incoming bit in carry flag
20                                LAB            Get any previous bits
20                                RAR            Rotate new bit fm carry to register
20                                LBA            Save in register B
20                                DCC            Decrement bits counter
44 / 36                           JFZ MORBDI     Delay & fetch next incoming bit
20                                RRC            Have all 5 bits, right justify
20                                RRC            In accumulator by rotating
20                                RRC            Before preparing to exit routine
44 + 9796                         CAL HDELAY     Optional delay to make sure into STOP
44 + 20                           CAL DUMMY      Part of optional delay
44 + 20                           CAL DUMMY      Part of optional delay
20                                RET            Units area before exiting subroutine

32                     IDELAY,    LDI 215        Set time loop counter
12                                RTS            Trim time, this condition never met
44 + 20                RDELAY,    CAL DUMMY      Time consuming loop
20                                DCD            Decrement counter
12 / 20                           RTZ            Exit to calling rtn when counter = zero
44                                JMP RDELAY     Otherwise continue using up time
```

```
32         HDELAY,  LDI 106        Set time loop counter
44                  JMP RDELAY     Go use up about half a time unit

20          DUMMY,  RET            Short routine to use up time
```

While the routine just presented is similar in many respects to the one described earlier for transmitting data from the computer, several different features will be high-lighted. First, the reader may note that the program expects data to be arriving at the most significant bit position of the accumulator (as in the SERIAL TO PARALLEL routine in the previous chapter).

Next, the reader should note that the three instructions starting at the label STRTIN form a loop to test for a START bit arriving from the input port. The reader can see that the loop requires 108 microseconds to execute and thus it is possible for a start unit to have been present for almost that length of time before it is detected. For instance, if the start pulse actually started just a microsecond after the INP X instruction at STRTIN was executed, that pulse would not be detected until the INP X instruction was executed on the next round. However, it is also possible for the program to detect the start bit at just about the instant it actually happens. Thus, there can be a variation in detecting the beginning of the START time unit of about 108 microseconds. Now, the actual detection of the start pulse is used as a reference for delaying to the middle of a time unit in order to sample the remaining bits in the desired region. On the average, one could assume that the start pulse was detected in about the middle of the possible range of variation, which would be about 54 microseconds after the pulse actually started. This information is used to establish approximately how long the HDELAY loop should be in order to get close to the theoretical middle of a time unit. Thus, if one assumes that on an average, the start pulse is detected 54 microseconds after it began, and one adds 144 microseconds for the execution of the instructions from STRTIN to the CAL HDELAY instruction, one can determine that HDELAY needs to consume 9802 microseconds. The value 9796 actually developed was a close enough compromise for the situation.

Another area of interest near the end of the main routine is marked by the comments as an optional delay to make sure that the program has consumed enough time so that the sending unit will be sending the STOP units before the routine is exited. As pointed out earlier, after the five data bits have been sampled the computer has quite a bit of time, up to about 50 milliseconds in which to perform some other functions because the sending machine would be unable to send a new START pulse until it had completed its current cycle which includes two units of time for the STOP pulses. However, in some instances, the computer may not require anywhere near that amount of time to process the character it had just received. In such cases the programmer would want to make sure the program did not start looking for a new START bit before the last DATA bit had been completed. The optional half a time unit delay ensures in such a case that the machine would be in its stop units phase, which by definition here would be a logic one state, before the computer began looking for a new logic zero condition that would signify the start of a new character.

Finally, the reader might take note of an interesting trick to get a rather short additional delay by the use of the RTS instruction as the second command in the IDELAY subroutine. A conditional return instruction when the condition is not met is the only type of command in an '8008' CPU that will use just 12 microseconds of time. The RTS instruction inserted at that point will never have the TRUE condition met as the reader may verify by close examination of the possible condition of the SIGN FLAG at that point in the program. It is a good technique to remember if a 12 microsecond delay is required. However, the programmer must make certain that the condition will never be satisfied when used for that purpose! (Remember, virtually all other types of instructions take up at least 20 microseconds of time to be executed in a nominally adjusted '8008' based system.)

As another example of the details of real-time programming, the above example will be expanded to demonstrate how the program could be improved to increase the reliability of receiving correct data from the external machine. As many readers may know, the incoming data from an electro-mechanical machine may be noisy. That is, a signal that is supposed to be, for instance, in the logic one state for an entire unit of time may occasionally go to the logic zero condition for small fractions of a unit of time, or vice-versa. In the program just presented the computer sampled the state of the incoming signal just once in each unit of time. If by chance it should sample the signal at the moment that noise was present on the signal line, incorrect data might be received. In a critical application, it might be desirable to reduce the chance of such an error occurring. This could be done by sampling the incoming signal several times during each unit of time and then computing an average of the value received to determine whether the signal was truly in a logic one or logic zero state. For instance, one could elect to sample the signal five times near the middle of each time unit and then make a decision as to whether the signal was a logic one or zero by determining which state was detected three or more times out of the five samples. Such a sampling method would greatly reduce the chances of noise causing an incorrect signal level to be received by the computer.

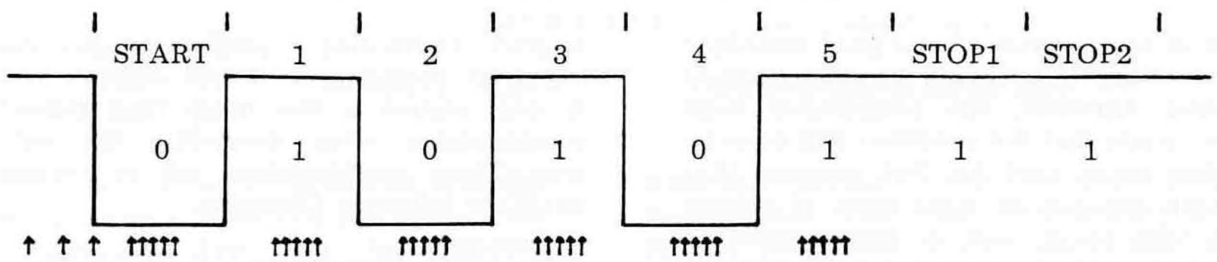
The timing diagram illustrated at the top of the next page shows a signal being sampled at multiple points as indicated by the arrows at the bottom of the signal diagram. Developing a program to give the improved performance is not difficult but it does require a few more time related considerations when developing the software. These considerations will be pointed out in the following discussion.

To begin development of the multiple-sampling program a major subroutine was developed that would perform the task of sampling five times in succession, keeping track of whether a logic one or zero was received, and finally determining which state was received most often. The subroutine with the execution time for each instruction is presented immediately following the timing diagram on the next page. The reader might pay special attention to the manner in which the predominant signal state was determined in the program.

Information regarding the amount of time required to execute portions of the multiple sampling routine were required before the overall routine could be developed for reasons that will soon be apparent.

The reader may confirm that the time between each of the five samples will be 280 microseconds for a typical '8008' system regardless of what signal state is received. It is important to notice how the sampling routine was balanced by the appropriate choice of instructions so that the receipt of either signal state resulted in the same total time to execute the sampling loop. If this requirement were not met the programmer would have quite a difficult time trying to develop an accurate routine based on all the possible combinations of one and zero signal states the could be received!

The reader should also note that the setup time, that is the time to execute the instructions from the label SAMPLE to BITEST plus the time to actually call the subroutine would require 108 microseconds. That is, it will take 108 microseconds from the time the program starts to call the subroutine until the first



TIMING DIAGRAM FOR MULTIPLE SAMPLING OF INCOMING SIGNAL

```
32         SAMPLE,  LDI 005       Set counter for number of samples
32                  LEI 377       Setup register E for storing signal state
32         BITEST,  INP X         Sample current signal on input line
32                  NDI 200       Mask off unused input lines
44 / 36             CTS PLUSE     Increment E if signal a logic one
32                  NDI 200       Restore flags to reflect ACC contents
36 / 44             CFS MINUSE    Decrement E if signal a logic zero
20                  DCD           Decrement sampling counter
44 / 36             JFZ BITEST    Sample again if counter not equal to '0'
20                  LAE           When have 5 samples place E into ACC
32                  NDI 200       Mask off all but most significant bit
20                  RET           Exit with predominant state in MSB

20          PLUSE,  INE           Increment register E
20                  RET           Exit subroutine

20         MINUSE,  DCE           Decrement register E
20                  RET           Exit subroutine
```

INP X instruction is encountered.

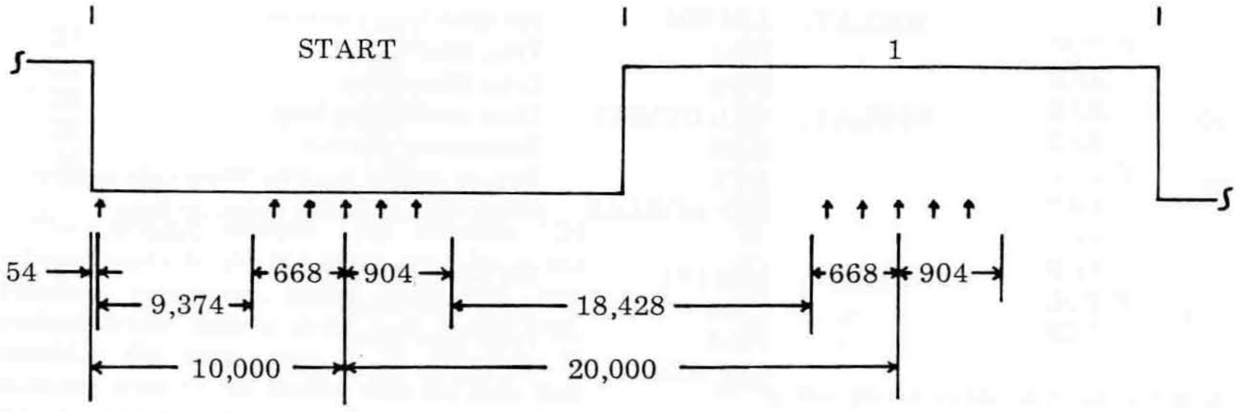
Additionally, the reader should note that it will require 344 microseconds from the time the fifth sample is taken until the subroutine is actually exited.

It is important to know these relationships so that the entire subroutine can be properly located within a time frame. For instance, since it would be desirable to have the third sample take place at the theoretical middle of a unit of time it will be necessary to start calling the sample subroutine when there are about 668 microseconds remaining before the theoretical middle of the unit of time. This is because it will require 108 microseconds between the first and second sample and another 280 microseconds between the second and third sample.

Similarly it is important to know that there will be 904 microseconds from the time the third sample is taken until the routine is exited. As 280 microseconds will be taken between sample number three and four, another 280 microseconds between samples four and five, and an additional 344 microseconds from sample number five to the time the routine is exited.

With this information now available one can calculate how much time should be used from the time a start bit is received until it is time to call the sample subroutine so that the third sample point will be in the middle of a unit of time. And, after that, how much delay to provide from the time the sample subroutine is exited in one unit of time until it is to be called again to sample the signal in the middle range of the next unit of time.

In a situation such as the one being discussed, it is often helpful to produce an expanded timing diagram to illustrate smaller portions of critical time relationships. An expanded diagram showing the information just derived as it applies to the START bit and the first DATA bit of the example incoming signal is shown below. Remember, the diagram only illustrates two units of time out of the eight contained in the character!



EXPANDED TIMING DIAGRAM

With the timing requirements of the SAMPLE subroutine known, the appropriate delays to place the sampling subroutine such that the third sample is at the middle of a unit of time can be ascertained as shown on the above expanded timing diagram. It is then a relatively easy matter to modify the program previously developed for the case when only a single sample was taken per time unit so that it calls the SAMPLE subroutine. An example of such a routine is presented next.

```
                       BDIN,    LBI 000        Clear incoming storage register
                                LCI 005        Set bit counter
32                     STRTIN,  INP X          Look for a START bit
32                              NDI 200        Mask off irrelevant data
44 / 36                         JTS STRTIN     If not START bit, form sampling loop
44 + 9184                       CAL HDELAY     If find logic zero, assume start, delay
44 + 1528                       CAL SAMPLE     and then do multiple sample on start bit
36 / 44                         JTS STRTIN     If result not zero assume false start
44 + 20                         CAL DUMMY      Add compensating delay before entering
20                              NDA            Main DATA sampling routine
20                              NDA            With these three instructions
44 + 18240             MORBDI,  CAL IDELAY     Execute main delay loop
```

```
44 + 1528                     CAL SAMPLE    Multiple sample routine on DATA bits
20                            RAL           Save resulting state in carry flag
20                            LAB           Get any previous bits
20                            RAR           Rotate new bit from carry into ACC
20                            LBA           Save formation in register B
20                            DCC           Decrement bits counter
44 / 36                       JFZ MORBDI    Delay and then fetch next DATA bit
20                            RRC           Have all five DATA bits so right justify
20                            RRC           In accumulator by rotating
20                            RRC           Before preparing to exit
44 + 9184                     CAL HDELAY    Optional delay to reach STOP area
20                            RET           Exit BAUDOT input subroutine

32                  IDELAY,   LDI 202       Set time loop counter
20                            NDA           Trim time delay
20                            NDA           Trim time delay
44 + 20             RDELAY,   CAL DUMMY     Time consuming loop
20                            DCD           Decrement counter
12 / 20                       RTZ           Exit to calling routine when cntr is zero
44                            JMP RDELAY    Otherwise continue using up time

32                  HDELAY,   LDI 101       Set time loop counter
20                            NDA           Trim time delay
20                            NDA           Trim time delay
44                            JMP RDELAY    Go use up more time

20                  DUMMY,    RET           Short routine to use up time
```

The information presented to this point in the chapter has been concerned with illustrating techniques to coordinate the execution of a program with the timing requirement of an external device, through the method of providing time delays to effectively slow down the execution of a program. However, another aspect of real-time programming involves essentially the opposite objective. That is to obtain maximum speed of operation from a computer program so that it may handle events that might be occurring quite rapidly. The balance of this chapter will present several basic guide lines for streamlining the operation of a program to obtain maximum speed of execution.

Perhaps the first point to present is that there is a corollary between obtaining maximum operating speed and the amount of memory required by the program that may at first seem a little strange. That is that as one attempts to program most microcomputers to execute a function in a minimum amount of time, one generally will increase the amount of memory required to hold the program! The reason for this relationship is that streamlining a program generally requires the elimination or reduction in the use of loops and subroutines, which, the reader may recall, were earlier stressed for their ability to save memory storage space!

To illustrate how the elimination of loops can dramatically reduce the time required to execute a specific function, consider the example presented next. In this case, a programmer needs to load three consecutive words in memory with the contents of the accumulator in as little time as possible. A routine using a loop might appear as shown next.

```
32                      LBI 003
28         AGAIN,       LMA
20                      INL
20                      DCB
44 / 36                 JFZ AGAIN
```

The reader may easily calculate that the total time required to execute the above loop would be 360 microseconds. A routine that did not use a loop could be executed in about one third the time in this particular case as illustrated next.

```
28                      LMA
20                      INL
28                      LMA
20                      INL
28                      LMA
```

The straight routine only requires 124 microseconds to do the same job. While the corollary mentioned above might not seem evident when such a short loop is involved, consider the same case if 20 locations in memory were to be loaded with the data that was in the accumulator. One can calculate that the loop method would only require eight (decimal) locations in memory for the operating portion of the program and would execute in 2,264 microseconds. On the other hand, the straight routine method would require some 39 locations in memory for storage of the operating program, but that straight routine would be executed in a mere 940 microseconds.

The elimination of subroutines can also greatly speed up the operation of a critical portion of a program as shown by the following example. The following subroutine method might be used as part of a program that was to rapidly output the contents of the accumulator as a series of octal digits.

```
24                      OUT X
44 + 80                 CAL ROTAND
24                      OUT X
44 + 80                 CAL ROTAND
24                      OUT X
16                      HLT
```

Where the subroutine ROTAND appears as:

```
20        ROTAND,       RAR
20                      RAR
20                      RAR
20                      RET
```

One can calculate that executing the above subroutined program would require 336 microseconds. The straight program method shown below only requires 208 microseconds to do the same function.

```
24                      OUT X
20                      RAR
20                      RAR
20                      RAR
24                      OUT X
20                      RAR
20                      RAR
20                      RAR
24                      OUT X
16                      HLT
```

While the above example does not support the memory usage corallary one can see that if the subroutine were somewhat longer, say it contained eight or nine instructions, then the corallary would be true.

Another rule of thumb to apply towards developing programs to operate in a minimum amount of time is to do as much work as possible with CPU registers instead of with memory. For instance, suppose one had an instrument interfaced to an '8008' system that periodically needed to send a short burst of data to the computer for storage. For technical considerations assume that is was desired to receive the burst as rapidly as possible, after which the computer would have some idle time to process the data. One can readily see by the following example that it will take much less time to store, for instance, four characters in CPU registers, than to store the same amount directly in memory locations. A routine to store the characters directly in memory as illustrated next would require a total of 300 microseconds.

Storing the data in CPU registers would only require 216 microseconds using the following routine.

```
32                      INP X
20                      LBA
32                      INP X
20                      LCA
32                      INP X
20                      LDA
32                      INP X
20                      LEA
```

The factor that might be particularly valuable in a time-tight situation is that each character in the second routine could be accepted at 52 microsecond intervals while the first routine could not accept the characters at a rate faster than every 80 microseconds. Naturally, the above example is strictly limited to the case where very short bursts are being handled as there are a limited number of CPU registers available in which to store data. However, the principle can be valuable.

The concept of utilizing CPU registers as much as possible can be extended to a variety of applications besides the one just illustrated. For instance, it is often advantageous to setup CPU registers in advance of a critical time period in order to streamline a program during selected operating periods. For instance, suppose one needed to input data at a fast rate and also perform some manipulation of the data, such as perform a two's complement operation and then deposit the data in memory. One way to develop the routine would be as follows.

```
32        RECEIV,       INP X
32                      NDI 377
32                      ADI 001
28                      LMA
20                      INL
44 / 36                 JFZ RECEIV
```

The above routine could have the time factor decreased by about 12 percent if, prior to entering the loop (a necessary evil in this example because a large block of data is hypothetically being processed), one first set CPU register B to contain '377' and CPU register C to hold '001' and used the routine shown next.

```
32        RECEIV,       INP X
20                      NDB
20                      ADC
28                      LMA
20                      INL
44 / 36                 JFZ RECEIV
```

A few closing comments on the subject of streamlining real-time programs would include the mention that if subroutines are necessary, to use those valuable RESTART commands which only require 20 microseconds for an effective CALL instead of 44 microseconds. Additionally, the programmer should pay strict attention to overall program organization in order to reduce time consuming overhead operations. Or, at least to defer such operations for execution during non-critical time periods.

Finally, real-time programming is an area where the creative programmer can have a lot of fun. Experiment, look for new methods to solve a particular problem. You may find a better, faster way! Such as:

Have the first instruction of the above routine located at the address of restart location 'X.' Modify the routine as illustrated below and cut another seven percent off the execution time of the routine!

```
32                      INP X
20                      NDB
20                      ADC
28                      LMA
20                      INL
12 / 20                 RTZ
20                      RST X
```

PROM PROGRAMMING CONSIDERATIONS

For readers who may not be familiar with the abbreviation, a PROM is a PROGRAMMABLE READ-ONLY MEMORY element. A programmable read-only memory element is an electronic device that can be programmed with a program using a special instrument so that it contains a permanent program. Some PROM elements can be ERASED and reprogrammed by using special instruments which are generally too expensive for the average user to have readily available. When the programs in such elements need to be changed it is generally necessary to send the device back to the manufacturer or representative for processing.

The key feature that a READ-ONLY MEMORY element has over a RAM (read and write memory) device is that once a program has been placed in a ROM it is non-volatile, or permanent. A semiconductor RAM device will lose its contents if power is removed from the device. A ROM will retain the information placed in it if power is removed. Thus, the ROM is an ideal memory device in which to store programs that are permanent in nature, or that have frequent uses in a system where power may frequently be removed. It eliminates the process of having to load programs back into memory when a computer system is initially powered-up for a period of operation.

The key disadvantage of the ROM is that the computer cannot alter the contents of those memory locations assigned to a ROM device. Thus one must take special precautions when designing programs that are to reside in a ROM device.

For instance, one cannot use memory addresses in a ROM to store temporary pointers and counters for a program that needs to alter such pointers and counters during the program's operation. Similarly, one cannot use any such locations for any kind of temporary storage of data or other temporary information because, as just mentioned, the computer will not be able to write the information into the ROM!

Thus, if a program is to be stored in a ROM, and it is necessary to use pointers and counters in a program (as will certainly be the case in many applications), one should arrange the program to use CPU registers for those purposes. Or, to use addresses in memory that will contain RAM elements.

A ROM element can be considered as a hardware memory element and as such, one of the first matters one should consider when planning on installing ROMs in a computer system is where to assign the ROM elements in memory. A good rule of thumb is to place such elements at the upper extreme addresses available in the system. For instance, if one has an '8008' system capable of addressing up to 4 K of memory, (PAGES 00 through 17) it would be advisable in most cases to develop programs for ROM(s) that are on page 17, or if more pages are required for ROMs, to work downward from that address. (Most ROM and PROM devices can contain 256 eight bit words, or one page in a typical '8008' system.) This allows all addresses below the ROM element(s) to be available as one continuous block of read and write memory. This is generally a more convenient arrangement than, say, sticking a ROM element on page 10 in such a system, thus dividing the available addresses for RAM memory into two separate areas.

Alternatively, one might want to consider placing ROM elements at the lowest available addresses for the system, and leaving the upper addresses available as one continuous block for RAM elements. However, unless a system is being designed to serve as a special function device, it is generally wise to not use a ROM on page 00 in an '8008' system, as it will occupy all the possible RESTART (RST) instruction locations! The exception to this would be if one deliberately wanted to have power-up routines that used the interrupt facility of the '8008' system in conjunction with a ROM to automatically go to a RESTART location. The RST class of instructions, which use the special locations on page 00, are particularly useful commands with general purpose applications, as discussed elsewhere in this manual. One should consider their general purpose capabilities carefully before deciding to restrict them to a ROM application.

The types of programs that are generally most suitable for placement on ROMs include: routines to assist getting a system on-line immediately following power turn-on, such as I/O routines and program loaders, frequently utilized programs that one may not want to have to be bothered loading each time a system is started, or programs for dedicated applications.

For instance, a user with an electronic typewriter might want to put a standard routine to input and output information to the device (which could be called by general routines) and possibly a loader program that would enable the user to quickly load programs into RAM memory via a paper tape reader. In such an application, one might also have space on a PROM to include a simple program that would enable one to examine and modify memory locations using the electronic typewriter device. Thus, whenever power was applied to the computer system, one would instantly be in a position to load larger programs into RAM memory. Or, to immediately use an electronic keyboard to place information into RAM memory. Without a ROM, the user would have to use manual control methods to load a loader program or other routines into memory. The savings in time one can achieve by using a ROM to store start-up programs over having to use purely manual procedures can be well worth the cost of a ROM or PROM device.

However, a user who desired to develop such a package for storage on a ROM device would have to be particularly careful when developing the I/O routine if such a routine required real-time programming considerations, such as a timing loop. For instance, the reader who has read the previous chapter will realize that if the computer program itself will control the actual operation of a device such as an electronic typewriter, and timing loops are established to control the precise time at which events will occur, that the actual timing required to properly operate a device will be a function of the device being controlled as well as the timing in the computer itself. The accuracy at which such timing must be maintained is a function of the accuracy of the timing in the computer system and the device itself. This accuracy may vary between different units. If a fixed timing loop was programmed into a PROM, and at some later data the external device was replaced with a different one, or the timing of the computer was adjusted, the original timing loop might be made invalid. Thus, in such an application it might be wise to place the actual data value that is to control the timing loop in a RAM location, and then have the program in the PROM access that value, which would be manually inserted by the operator, rather than having the value fixed in the PROM. The following two subroutines will help clarify the point.

PROM PROGRAM WITH A FIXED TIMING LOOP VALUE

```
  TIME,   LDI 100      Set timing loop counter
 TIMER,   CAL DUMMY    Delay subroutine
          DCD          Decrement timing loop counter
          RTZ          Exit subroutine when time delay done
          JMP TIMER    Otherwise continue timing loop
```

PROM PROGRAM WITH CAPABILITY TO ALTER TIMING LOOP VALUE

```
     TIME,  LHI XXX     Set pointer to RAM location where
            LLI YYY     Timing loop counter stored
            LDM         Set timing loop counter value
    TIMER,  - - -       Same as previous routine
```

The second routine illustrated above assumes that the CPU memory pointer registers will be setup to point to a location in RAM memory where the actual loop counter value will have been placed by the operator. While the method necessitates the operator having to set the proper value into RAM memory before using the program in the ROM, it avoids the problem of having a useless program in the ROM if a timing value must be altered at some future date. It should be apparent that this kind of scheme can be applied to any similar situation where a value used by a program might conceivably need to be altered.

If, for some reason, one did not want to have to dedicate a location in RAM memory for a variable value in such a routine, there is still another trick that can save the day in such a situation. The operator could manually load the D register in the CPU prior to using the above type of subroutine (or have an external routine in RAM memory perform the same function before using the routine). In that case, one could eliminate the portion of the above routine labeled TIME and simply use that portion labeled TIMER.

A good rule of thumb to apply when considering the use of ROM in a system is to tailor the program for compactness. After all, the more routines or subroutines one can store on a PROM, the more useful the device will be. Make every effort to save memory space by judicious use of subroutining, with multiple entry points if applicable, and by use of program loops. An earlier chapter stressed the concepts and provided guide lines and formulas for calculating when such techniques are applicable. One should figure on spending some extra time when developing programs to be stored on ROMs in order to look for ways to save memory space. Try to use every available location on a PROM. After all, any unused locations will be permanently wasted. If one finds one has some room left in a PROM after one has placed the programs required to be on the device for a particular application, consider the possibility of tucking in a few small routines that would have general usefulness. Such subroutines as SWITCH, ADV, and CNTDWN which were presented and used frequently in examples throughout this publication are typical kinds of generally useful subroutines that one might consider having on a ROM rather than wasting locations. These types of routines would then always be available in the system for use by programs residing in RAM memory.

Above all, however, once one has developed routines for a PROM, one should thoroughly CHECK and TEST the program(s) to make sure they are absolutely operating as intended. It is a bit costly to have to make a program patch on a read-only memory element!

CREATIVE PROGRAMMING CONCEPTS

Once one has become familiar with the fundamental aspects of machine language programming. Once one is familiar with the mnemonics that represent the machine language commands and can mentally think of the functions that those mnemonics represent. Once one has learned how to formalize and plan out a program, understands flow charting, and memory allocation or mapping. Once one has had some practice at developing algorithms and combining smaller algorithms into full sized programs by subroutining. Once one is familiar with setting up pointers, counters, forming program loops, utilizing bit masks. Once one has a feel for organizing data for tables, and understands how data can be sorted. Once one understands how mathematical information may be processed by the computer. And, once one knows how to get data into and out of the CPU from and to some external device. For example, once one has spent a little time studying the aspects of machine language programming a computer, as one will have done by reading (and hopefully learning!) the information presented in the preceeding sections of this manual. Then, one should be in a position to understand and appreciate the true potential of a digital computer when its power is unleashed under the auspices of a creative programmer. That is when one can really start having fun creating and developing completely original programs to perform myriads of personally desired functions. This is the point at which one may take a broad view of the immense capability of the machine by standing back and pondering some scenes, much the way an artist would ponder a blank canvas before starting to paint a concept or image that existed purely in the artist's mind. The discussion that follows merely presents some ways in which to view the capability of a digital computer. Some points of view that may help programmers approach programming tasks with creativity. No great magic is claimed for the ideas presented. No guarantee is made that the points of view will inspire everyone to greater programming creativity or ability. But, it is known that the views presented have helped at least one programmer to create countless programs, some of which others had claimed couldn't be done on a small machine, and solve numerous programming problems, while having a lot of fun, and quite often saving a lot of time! Thus, the ideas will be presented in the hopes that perhaps a few others will benefit a little, or a lot.

It must be admitted that to some readers the concepts discussed in this section might seem trivial at first glance. Perhaps the reason some people initially see the concepts as trivial is because they are profoundly broad, and to some lucky people, perhaps instinctively obvious. However, most readers will probably find the concepts grow as one does more and more programming, until one day the reader discovers a profoundly simple way to handle a programming problem based on a variation of one sort or another of the concepts presented in this section.

For what they are worth, the concepts to be presented will be discussed in three parts.

THE ONE DIMENSIONAL VIEW

The underlying principal in this entire discussion on creative programming is to leave out the details of the operation of the CPU and its associated registers. It is known that the CPU and the associated registers can do a whole host of specific operations, mathematical, Boolean logic, execute conditional branches and whatever. These functions will be taken for granted in the following discussion. What is important in the present situation is to realize that the power of the computer is in its memory. The CPU obtains its instructions from memory, and the CPU is able to manipulate information in memory. The CPU is able to access a particular word in memory, in the case of an '8008' system, by pointing to the address using the H & L registers. For each specific address there is a specific word in memory that contains eight binary bits.

One way to view organization of memory is to think of memory as being one long line of words, stacked one after the other. In fact, this is the way virtually any machine language programmer first starts thinking of memory because of the simple way in which each memory address corresponds to a word in memory, and memory addresses are simply a series of consecutive words.

```
**************************************************
*  ADDR  NO.  N      *  MEM WORD  NO.  N         *
**************************************************
*  ADDR  NO.  N + 1  *  MEM WORD  NO.  N + 1     *
**************************************************
*  ADDR  NO.  N + 2  *  MEM WORD  NO.  N + 2     *
**************************************************
         .          .          .       .        .
         .          .          .       .        .
**************************************************
*  ADDR  NO.  N + X  *  MEM WORD  NO.  N + X     *
**************************************************
```

Thus one can consider memory as simply being one long string of locations that may be filled with whatever information is desired in a serial sequence. If one were to fill each memory word with a code that symbolized a letter or digit, or punctuation symbol, one could proceed to fill a string of memory locations with English (or French, or German, or whatever) words, and go on to form sentences, and by using other codes, to separate sentences into paragraphs.

```
    N          O           W          SPACE         I          S

 ADDR N    ADDR N+1    ADDR N+2    ADDR N+3    ADDR N+4    ADDR N+5
```

Or, one could place mathematical values in memory locations, separate those values by operator symbols, and process columns of mathematical data. (Assuming in this strict case that the values were small enough to be stored in one memory word.)

```
ADDR  N   :  +100
ADDR N+1  :  MINUS
ADDR N+2  :   - 50
ADDR N+3  :  EQUAL
```

Or, the contents of memory words may be used to symbolize just about any abstract item that the programmer might desire. The programmer need simply form a code that the programmer desires to have symbolize something.

```
ADDR  N   :  SYMBOL FOR APPLES
ADDR N+1  :  SYMBOL FOR PEARS
ADDR N+2  :  SYMBOL FOR BANANAS
ADDR N+3  :  SYMBOL FOR CHERRIES
ADDR N+4  :  SYMBOL FOR LEMONS
ADDR N+5  :  SYMBOL FOR BELLS
```

The reader should realize here that the concept being presented is concentrating on how memory is utilized for handling data or information. It is taken for granted that a portion of memory will be used for the actual operating program that controls the manipulation of the memory that is being used for the data. Thus, in the previous examples, one must realize that an operating program will place the codes for letters or digits, punctuation marks, spaces, and so forth, and perform whatever processing is desired. An operating program will take the values given in the mathematical example and interpret the symbols and perform the desired functions. And, an operating program in the third example would recognize a particular code to mean apples, and print or display the entire word (or picture!) when it interpreted that code. The primary point being made is that the data is organized as a long line of information. That line of information can be arbitrarily split up into many parts, and pieces of the line be considered as forming one particular section, as in the case when one English word is formed from a series of letters. The long line is simply formed and locations along the line are marked by a memory address.

However, and this the creative programmer should take particular note of, the fact that locations are marked along the line by memory address can be transformed by the programmer so that memory addresses essentially stand for any arbitrarily assigned marker. In other words, to the programmer, memory address number N can correspond to time T, or distance D, or point Z. Thus, one can store, say, the value of the amplitude of a signal at time T in one location, the value at time T+T in the next location, the value at time T+2T in the next location. Furthermore, it should be apparent that T can be scaled as desired by appropriate programming so that T represents one microsecond, or millisecond, or second, or a year!

Furthermore, one can actually go beyond the point of considering the locations to be a long straight line, by considering the possibility of manipulating the line of locations as a piece of string. One can figuratively cut the piece of string at any desired location and form the string into a ring or circle. This is easily accomplished by simply having the memory address pointer go back to location N when it reaches location N+X. Consider the possibility of doing such an operation with three sections of the line, and using the technique to simulate a one armed bandit machine:

ADDR N	APPLE	ADDR N+X+1	PEAR	ADDR N+2X+1	BANANA
ADDR N+1	PEAR	ADDR N+X+2	BANANA	ADDR N+2X+2	LEMON
ADDR N+2	CHERRY	ADDR N+X+3	LEMON	ADDR N+2X+3	APPLE
ADDR N+3	BANANA	ADDR N+X+4	BELL	ADDR N+2X+4	Bell
ADDR N+4	LEMON	ADDR N+X+5	CHERRY	ADDR N+2X+5	PEAR
ADDR N+5	BELL	ADDR N+X+6	APPLE	ADDR N+2X+6	CHERRY

One could develop algorithms to spin the memory pointer around each ring and randomly come to a stop at a location within each ring. The results of the events in all three rings could then be processed to determine whether one hit a jackpot or missed. The details of such a program will be left to the creative programmer, but the concept of how one could approach such a simulation project is hopefully clear.

Finally, to take the one dimension view a little further, one can go down to the bit level. Since a memory word in an '8008' system actually consists of eight individual bits, one could consider memory to be a long list of '1's' and '0's.' Each memory location contains eight bits, and by using consecutive memory locations one can build up long strings of bits. Again, the string can be broken at any desired point and manipulated as desired. This technique can be used, say, to simulate a huge shift register (using rotate instructions), or to represent an event occuring, or not occurring at points in time, or at distances along a line. In this view, a bit is addressed as being at a specific position within a specific memory address location. While the programming overhead to manipulate such data will generally be more complicated than the case where entire memory words are used to represent a symbol or piece of data, one can see that the basic concept of considering all bits in memory as being formed of one continuous line of ones and zeros is a valid, and often useful, image.

THE TWO DIMENSIONAL VIEW

The concept of viewing memory as a two dimensional plane will be started by considering an image at the bit level.

```
                    ADDR N  *  ADDR N+X+1  *  ADDR N+2X+1

ADDR  N      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
             1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
             1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1
             1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1
             1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
             1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1
             1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
             1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
             1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1
             1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1
             1 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 1
             1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1
             1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1
             1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
             1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
             1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
             1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1
             1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1
             1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1
             1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1
             1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
ADDR N+X     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

                    ADDR  N+X  *  ADDR  N+2X  *  ADDR  N+3X
```

The above diagram illustrates an image created by the status of the bits in a plane of memory. The plane was established by essentially taking lines of memory addresses (as presented in the one dimensional view) and placing them alongside one another to form a surface or plane. This convention would be established by the manner in which the programmer manipulated the memory pointer in the CPU. In the above illustration the plane is established at the most fundamental (and complex) level, and bits within each word are manipulated. As may be observed in the above diagram, one can view and manipulate bits in memory so as to form pictures or diagrams. The above represents a rectangle, a diamond, and a cross as an image made up of appropriate ones and zeros in selected bit positions. One could manipulate portions of memory to represent pictures. (Or charts, graphs, plots!) The degree of detail which one can obtain by such manipulations is a function of how many bits are used to represent a given area of a real (or proposed real) object. The above example presents all kinds of possibities for the creative programmer. One can use such techniques to form models, create patterns, and so forth.

In fact, going the other way so to speak, that is from having the computer generate patterns or objects, one can also take the two dimensional concept and apply it towards having the computer recognize objects by projecting their shape or form as a similar image of ones and zeros in memory.

Much research is currently being conducted towards developing algorithms that can recognize objects. One approach that is being studied is an interesting application of the two dimensional concept. A picture of an object is mapped into memory with '1's' being used to represent the area occupied by the object, and '0's' for areas outside. Then, the computer is trained to identify objects by using algorithms based on a neighboring bits scheme. In this manner, the computer determines how many '0's' surround a '1,' and performs calculations to find the outline and shape of the object. These findings are then coupled with complex algorithms to attempt to identify the object from a class of possibilities.

Such programs are of course quite complex and the details of such manipulations are somewhat esoteric. But, the idea is intrigueing and can provide fertilization for the creative programmer's imagination.

Taking the two dimensional view to the memory word level is perhaps a bit less complicated (it is! it is!) than considering it at the bit level. In this case, one needs only envision a plane of memory words which can contain codes for letters, numbers, symbols, or actual mathematical values. The reader has already seen examples of programs that could be considered as two dimensional in organization. One, for instance, was described in chaoter four in the presentation of the names sorting program. There, lines of names were formed one beneath the other in order to make the sort routine easier to program. One might review the diagram showing the sample names stored in memory as they relate to the memory addresses, which was presented near the end of chapter four.

The programmer is again reminded that as in the one dimensional view, the memory addresses that form the X and Y boundaries of a two dimensional memory plane can actually be thought of as arbitrary units, such as time, frequency, or distance, and the programmer also has the freedom to scale both the X and Y boundaries by appropriate software. The next illustration shows how an altitude map of a geographical area might be stored in a plane of memory.

	N	N+X	N+2X	N+3X	N+4X	N+5X	N+6X	
N	060	065	070	075	074	070	064	500 YDS
N+1	061	076	084	083	080	076	070	400 YDS
N+2	062	078	088	098	096	091	082	300 YDS
N+3	062	078	090	102	101	089	072	200 YDS
N+4	055	070	075	053	047	063	039	100 YDS
N+(X-1)	040	035	020	010	011	009	008	0 YDS
	0 YDS	100 YDS	200 YDS	300 YDS	400 YDS	500 YDS	600 YDS	

In the above illustration each memory location contains a value that represents the elevation of a piece of land. The top and left side of the illustration shows the actual memory addresses in the computer while the bottom and right side illustrate that each address actually stands for 100 yards distance. It should be apparent that the elevation factors could be, instead, inches of rainwater, or a temperature profile for the area, or, as previously mentioned, that the yards can be almost anything else the programmer might desire to define.

As a final example of the two dimensional concept, the reader will be left with the following diagram, which hopefully, will encourage one to consider the possibilities for much more complex board games.

```
                    N          N + X + 1      N + 2X + 1

                         *                *
      N             X    *        O       *       X           N + 2X + 1
                         *                *
               ***********************************
                         *                *
      N + 1         O    *        X       *       O           N + 2X + 2
                         *                *
               ***********************************
                         *                *
      N + X         O    *        X       *       X           N + 3X
                         *                *

                  N + X        N + 2X          N + 3X
```

Finally, the reader will be reminded, that in a manner similar to forming a ring as discussed in the one dimensional view, one can also consider forming a cyclinder out of a plane with interesting ramifications.

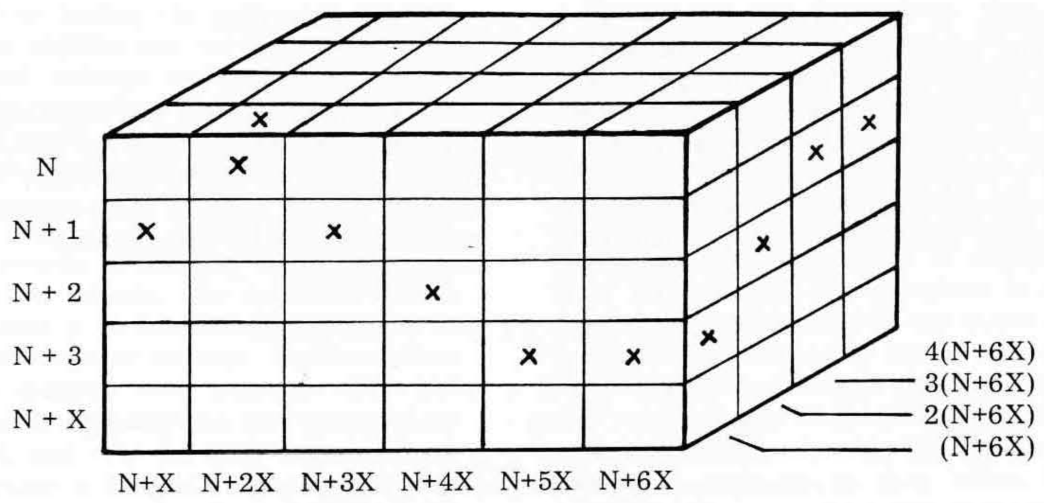
THE THREE DIMENSIONAL VIEW

It should be apparent that if one can set up memory locations by appropriate addressing to represent lines and planes, one can extend the principle out to the third dimension to form cubes of memory. There are many interesting possibilities when memory is viewed in this manner. One can plot three dimensional graphs or vectors. One can approach many types of modeling and manipulate such models so as to obtain different cross-sectional views.

As in the case of the one and two dimensional images, the programmer can substitute (effectively) memory addresses for special scale factors, now along three axis. And, as in the previous examples, one can take such manipulations down to the bit level if desired.

A diagram on the following page presents an image of memory when viewed as a three dimensional working area.

It is hoped that by this time the reader has received sufficient information on the practical aspects of machine language programming from the preceeding chapters, and that this concluding chapter has provided some stimulating concepts, so that the reader may go on to develop programs that will be of particular value to the individual. It is also hoped that those who have been introduced to the subject by this manual will find machine language programming an exciting, enjoyable, and in as many ways as possible, a rewarding endeavor!



THREE DIMENSIONAL VIEW OF MEMORY