Chapter 3

BYTE Reprint

MACHINE LANGUAGE PROGRAMMING FOR THE "8008" and similar microcomputers

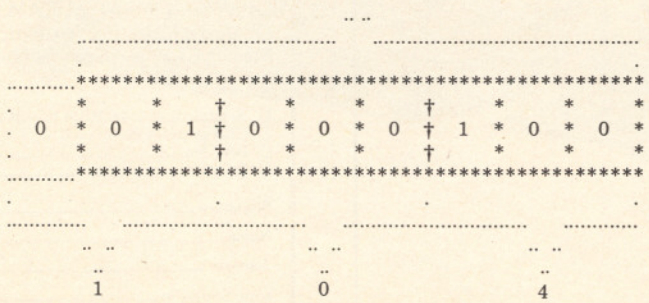
FUNDAMENTAL PROGRAMMING SKILLS

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



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:

1 1 1 1 1 1 1 1

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

0 0 0 0 0 0 0 0

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:

```
  1 1 1 1 1 1 1 1
+ 0 0 0 0 0 0 1 0
-----------------
  0 0 0 0 0 0 0 1
```

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 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.

MEMORY USAGE MAP

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						

PROGRAM DEVELOPMENT WORK SHEET

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

The same type of form may also be used as a program development sheet as shown here . 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 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!

PAGE	LOC	MEMORY 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

PAGE	LOC	MEMORY CONTENTS	MNEMONICS		COMMENTS
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.

PAGE	LOC	MEMORY 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			
* 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 technologists 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 consists 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
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!

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! ■