Computer Systems Department

BUIC III BASIC PROGRAMMING CONCEPTS AND TECHNIQUES

January 1968

Keesler Technical Training Center
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler 9- 3839

CONTENTS

CHAPTER 1

INTRODUCTION TO PROGRAMMING

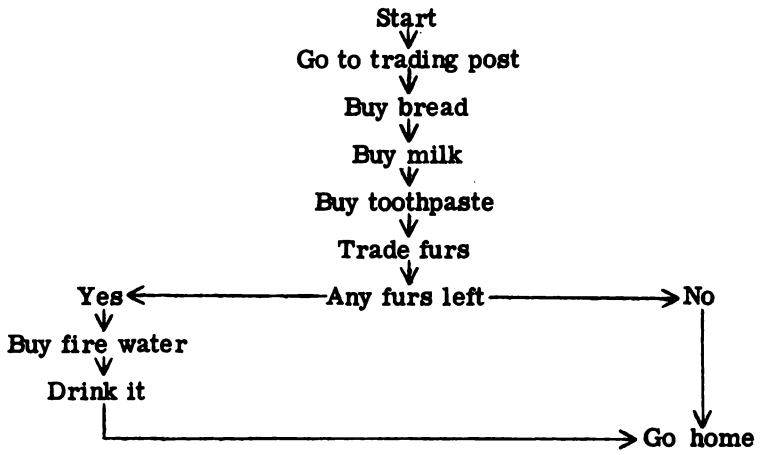
Electronic computers are complex machines of thousands, or even millions of electrical and electronic parts, capable of solving the most complex and intricate problems with extremely high speeds; yet, an electronic computer is totally incapable of doing anything but routing electrons through a predetermined path. As incongruous as the above statements may seem, they are nonetheless true. What is it that gives the electronic computer the capability of performing meaningful logic operations? The answer lies in the ability of the computer to route signals at electronic speeds in a logical manner predetermined by some person whom we shall call a PROGRAMMER.

It is the job of the programmer to provide the computer with the logical sequence of electronic operations necessary to solve a given proglem. However, before a programmer can specify the necessary operations to be performed by the computer, he must first fully understand the problem and be able to analyze it. Then he can determine a method of solution.

Our immediate concern in providing the logical sequence of operations for a computer ("programming") is to develop the techniques of (1) analyzing a given problem, and (2) determining a method of obtaining a solution for the problem. Later, we will learn how to code the individual operation in a manner that will be understood by the computer.

To illustrate a problem and its logical solution imagine this: An Apache brave with a load of furs is sent to the corner trading post by his squaw. His instructions are to buy one loaf of bread, one quart of milk and some toothpaste. If he has enough furs left over, his intentions are to buy a bottle of fire water.

The probable sequence of events would be:



You can see the breakdown of this problem into a logical sequence of steps.

Let's suppose that one day he goes to the trading post with five furs, and it costs four furs for the bread, milk and toothpaste. He will then have one fur left for fire water which he will buy. If, however, the next day the cost of milk goes up one fur and he has only five furs, he will not have enough to buy fire water, and he will have to go directly home.

Notice the same general solution fits both cases. We have arrived at a method of solution without having to know specific values for a given case. We could store the sequence of events (set of instructions) in a computer and then operate on data for any specific case (in this instance, any day) and obtain a solution. The same set of instructions could remain in the computer and could be used repeatedly for any specific situation.

The concept of a stored program (sequence of operations), which operates on different sets of input data is important to understand. The methods of problem-solving rely on this concept.

INFORMATION CODING

Information must be received, as well as transmitted, internally by the computer in a Alpha-numeric code or form recognizable to the computer. Alphanumeric 12-bit Hollerith coding information is used for card input into the computer. Twelve-bit Hollerith information will have to be converted again into a type of coding (6-bit Hollerith) which lends itself more readily to internal transfer and manipulation by the computer.

Some computers operate strictly on binary information, where all data manipulation must be done in pure binary form. For instance, the sum of numbers is obtained by adding them in their binary form. The sum is generated in binary form. Thus, all input data must be converted into binary format. Other computers, however, use various internal coding schemes to handle data. The octal system is an easier method for handling binary information. The computer still operates in binary, but its output can be in octal. Since only three binary digits are necessary to represent an octal number, the conversion from octal to binary can be made by inspection. Most computers are designed so that the octal system can be utilized.

Another example is binary coded decimal (BCD). In this instance, each digit of a decimal number is represented by its binary equivalent. Since the maximum value of any decimal digit will be 9, the number of binary digits necessary to adequately represent any decimal digit will be four.

FIXED AND VARIABLE LENGTH COMPUTER WORDS

In core memory of computers, a group of bits treated as a unit is referred to as a "word." Word lengths are fixed or variable, depending on the particular computer. An example of a fixed word length computer is the SAGE computer (AN/FSQ-7) which uses a word containing 32 bits. The word length of other computers of this type may differ (18 bits, 24 bits, 48 bits, etc.), but in each case the word is given a unique numbered "location" or "address" in core memory which may be specified to gain access to the contents of the word. In Figure 1-1, four computer words containing binary data (addresses Ø through 3) are shown. Note that the contents of the second word, address #1, is the binary equivalent of decimal 31.

ADDRESS	CONTENTS OF ADDRESS
Ø	Ø Ø 1 1 Ø Ø Ø 1 Ø Ø 1 1 1 Ø 1 Ø Ø 1 1 Ø 1 1 Ø Ø 1 Ø Ø 1 1 Ø 1 Ø
1	Ø 1 1 1 1 1
2	Ø Ø Ø 1 1 Ø 1 Ø Ø 1 1 Ø 1 Ø Ø Ø 1 Ø 1 Ø 1 1 Ø 1 1 1 Ø Ø 1 Ø Ø 1
3	Ø Ø Ø Ø Ø Ø Ø 1 Ø 1 1 Ø 1 Ø Ø Ø 1 Ø 1 1 1 Ø Ø 1 Ø 1 Ø Ø 1 1 Ø 1

Figure 1-1.

An example of a variable word length computer is the IBM 1410, one of a number of "character oriented" machines. These computers are character oriented in the sense that a coded character is considered to be the basic addressable unit. A character is represented by a specific number of bits (6 in the case of the IBM 1410) and is normally the smallest unit of data to be manipulated. Each unique address refers to the location of a character rather than a computer word. Figure 1-2 illustrates characters stored in memory (addresses Ø through 7), but keep in mind that each character shown actually represents a specific number of bits in the computer. Note that the contents of address #4 is a "C."

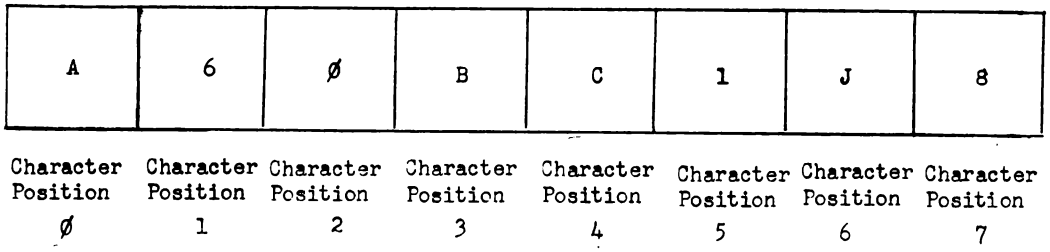


Figure 1-2.

In the variable word length machines the number of characters per word is not fixed, but is determined by the programmer. Each character in the memory of the IBM 1410 has an extra bit associated with it which can be set to signal the beginning of a word. This extra bit will indicate whether the character has a "word mark" associated with it. A word mark is represented in printed form by an inverted circumflex (v) above a character, for example, the letter A with an associated work mark is printed Ǎ. Consider the string of characters in Figure 1-3 and note that here each word boundary is defined by a word mark over the highest order (leftmost character.)

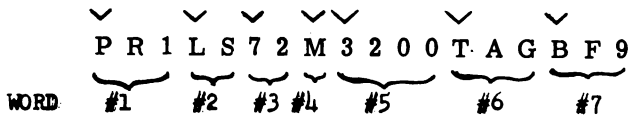


Figure 1-3.

A word is usually referred to by specifying the address of its lowest order character and "looking left" until a word mark is encountered.

Speaking generally, the computer word is accepted to be synonymous to register length. Some computers operate on a register at a time (AN/FSQ-7) and are register oriented machines. A few machines, using a secondary classification, can be considered syllable oriented. These operate on part, all, or more than one register at a time depending on syllable length and the number of syllables in any one operation.

The BUIC Computer (AN/GSA-51A) is register oriented in the sense that it has a fixed length register and is syllable oriented in the sense that it can operate on up to seven syllables in one operation. These syllables may encompass parts of three contiguous registers. The BUIC computer can be classed as a register oriented machine as opposed to character orientation, although it does not meet the strict definition of a register oriented machine.

Most of the IBM computers are character oriented as described above.

INSTRUCTION SETS AND DATA SETS

Information stored within a computer can be classified into two categories--INSTRUCTION WORDS and DATA WORDS. An instruction word is a coded command that tells the computer what to do for a single operation in a program. Each instruction takes a computer word; the length of which is dependent upon the type of computer, fixed length or variable length. A group of instructions that make up a part of the program or a complte program is referred to as an INSTRUCTION SET.

A data word contains a value, represented by either numbers or alphabetic information, which is to be manipulated by the computer in a manner set forth by an instruction word.

Instruction sets are stored in core in a specified sequence determined by the programmer. DATA SETS are also stored in core, normally in a different area than instruction sets since the computer has no way of distinguishing between an instruction word and a data word. If, through a fault in program logic, a data word is accessed during the time that the computer normally expects an instruction word, the computer would attempt to operate the value as an instruction. This is the main reason instructions and data are normally stored in different areas of data sets at the same time. (See Figure 1-4.)

CORE MEMORY

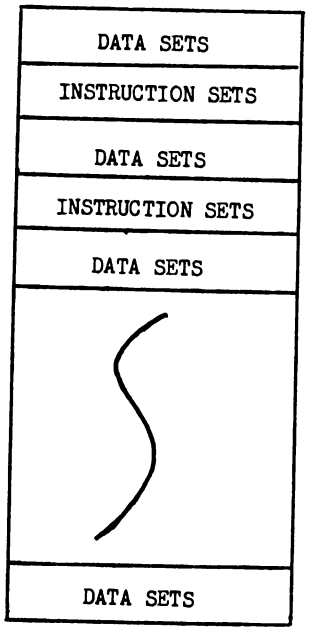


Figure 1-4.

INSTRUCTION WORD FORMAT

In the instruction word of a fixed word length computer one group of bits is termed the COMMAND PORTION and another group of bits is termed the ADDRESS or DIRECTOR. The number of bits in the command and director portions is again dependent upon the system being used. It is important to realize the terms "command" and "director" may not be used in every system, but the concepts are the same in every system. For instance, in some systems they may be called the "operate" and "operand" portions, respectively. The command portion of the instruction word contains a binary-coded command specifying the operation that the

computer is to perform. The address portion or director of the instruction word contains a binary-coded address which identifies a specific location in core memort that contains a value to be manipulated by the commad. The bits which make up the command portion of the word may be subdivided into groups, each having its own operational significance. Figure 1-5 shows one possible configuration for an instruction word.

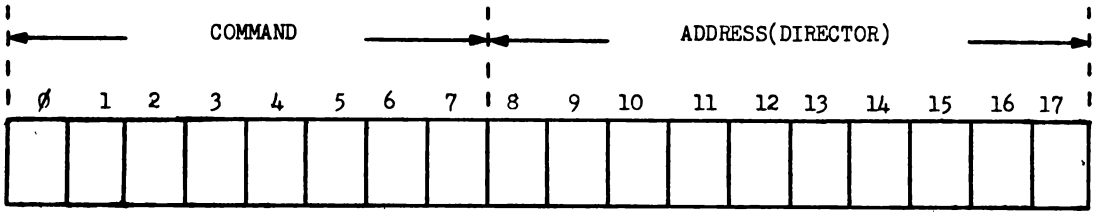


Figure 1-5.

It should be reiterated that in core storage there is no physical difference between an instruction word and a data word. Each is a sequence of binary bits, each bit having a value of "1" or "Ø." The distinction between instruction words and data words is determined by the timing cycle of the computer that extracts the register contents from core. For our purposes in this text, we will classify all timing cycles in two general categories: an "instruction" cycle and a "data" cycle. If a register is extracted from memory during an instruction cycle, the contents of that register are decoded and treated as an instruction. The computer treats the coded configuration of 1's and Ø's as an ADD, SUBTRACT, DIVIDE, or MULTIPLY, etc., instruction. If through some error in the logic of the program, the computer extracts from core a data word during an instruction cycle, it will go through the normal decoding process. At this time there are two possible actions for the computer to take, both of which will result in errors. First the data word might have the configuration of an instruction word and the computer will execute an instruction which was not intentionally programmed. Or, the computer will not recognize the word as a command in its repertoire of instructions and will hang-up.*

During a data cycle the computer extracts the contents of a word from memory, brings it directly into the CPU (Central Processing Unit), and operates on it as data (i.e., adds, subtracts, multiplies, divides, etc.). At this point it is stressed that the words brought from memory to be manipulated may be either a data word or an instruction word. This gives the computer the capability to modify individual program instructions, a very powerful programming tool.

To illustrate how an instruction may be modified, let us assume a hypothetical system in which the command codes are as follows:

23_8 = ADD 25_8 = DIVIDE

24_8 = SUBTRACT 26_8 = MULTIPLY

During a data cycle, location X containing the instruction code 23_8 (ADD), is brought into the CPU and another register containing the value 1 is added to it: 23 + 1 = 24. The sum 24 is stored into location X. The contents of X is now a SUBTRACT instruction, and, if the program ever decoded location X again, it would perform a subtraction.

*Hang-up. A non-programmed stop in a routine. It is usually an unforeseen or unwanted halt in a machine pass. It is most often caused by improper coding of a machine instruction or by the attempted use of a non-existent or improper operation code.

DIRECT ADDRESSING

It was mentioned that the address portion of an instruction word contains the location of the word to be manipulated. This is called direct addressing. There are also many ways in which addressing can be modified.

INDIRECT ADDRESSING

Some computers have the added capability of using INDIRECT ADDRESSING. In indirect addressing, the address portion of the instruction word contains not the address of the data word, but rather the address of a word (data or instruction) which contains the address of the data word. Some indicator must be used in the instruction word to indicate that indirect addressing rather than direct addressing is being used.

The following discussion is based on a hypothetical register of 18 bits. An instruction was being executed by a computer and bit #8 is designated as the indirect address indicator. A value of "Ø" in bit #8 would indicate direct addressing and a value of "1" would indicate indirect addressing.

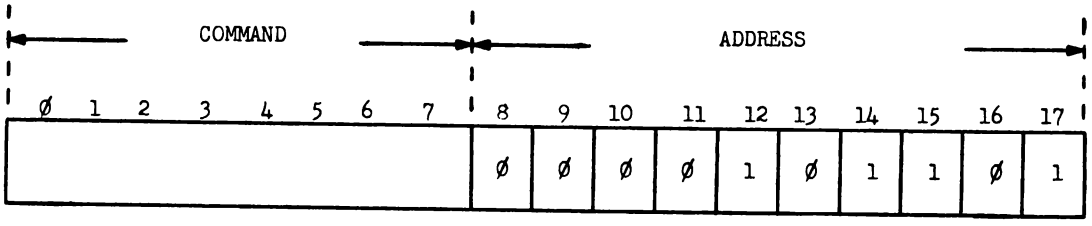


Figure 1-6. Bit #8 Indicates Direct Addressing

In Figure 1-6, the instruction word is using the direct addressing mode and the computer will perform the specified operation on the contents of the register at location $55_{(8)}$.

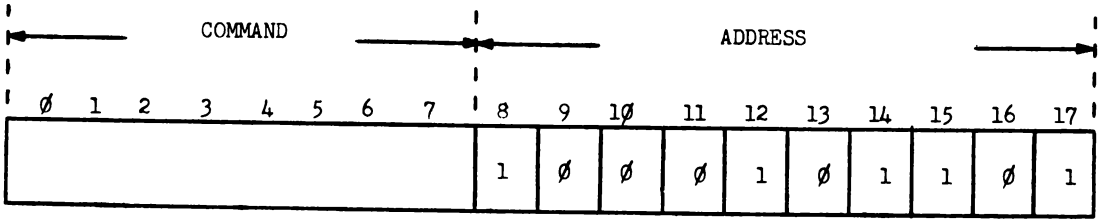
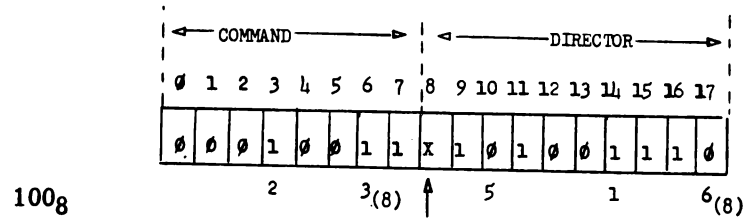


Figure 1-7. Bit #8 Indicates Indirect Addressing

The indirect addressing mode is illustrated in Figure 1-7 by changing bit #8 to a "1." The program now recognizes that location $55_{(8)}$ will contain the address of the operand rather than the operand itself.

Let's take a look at the difference between direct and indirect addressing. Suppose the computer is about to perform the instruction (Figure 1-8) located at core location $1ØØ_{(8)}$. It has decoded the instruction ($23_{(8)}$) which says to add the specified data (or operand) to a register in the CPU.

EXAMPLE 1



X = Ø, Direct Addressing
X = 1, Indirect Addressing

Figure 1-8.

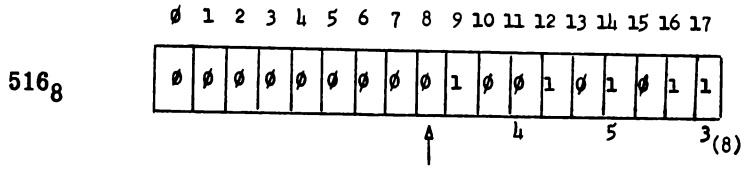


Figure 1-9.

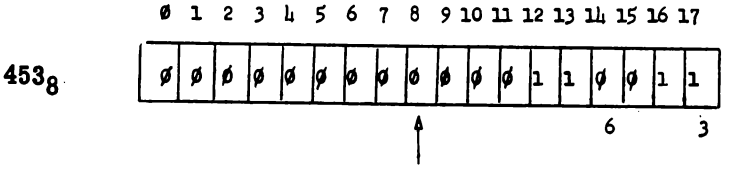


Figure 1-10.

If X = Ø in Figure 1-8, indicating direct addressing, the DATA to be added (the operand) will be contained in core location $516_{(8)}$, which would be the value $453_{(8)}$. If X = 1 in Figure 1-8, indicating indirect addressing, the address of the operand is located in location $516_{(8)}$. At location $516_{(8)}$, Bit #8 is a Ø. Therefore, the program is now in the direct addressing mode. The operand is located at address $453_{(8)}$. The operand is then the contents of location $453_{(8)}$, which is $63_{(8)}$.

RELATIVE ADDRESSING

In the previous mention of direct addressing, the address in the director of an instruction word specified the absolute core location of the operand. In some modes of operation, however, the address in the director is not the absolute core location, but must be added to another reference address to give the actual core location of the operand. This reference address is usually contained within the Base Address Register (BAR), Figure 1-11, and is added to the director automatically in all direct and indirect addressing modes to obtain the absolute core location of the operand. The actual core address is then relative to the base address, which is located in the CPU in the Base Address Register (BAR).

Figure 1-12 shows an instruction word in the CPU that has just been decoded as the command ($23_{(8)}$) which is to add the operand to the contents of a register in the CPU.

EXAMPLE 2

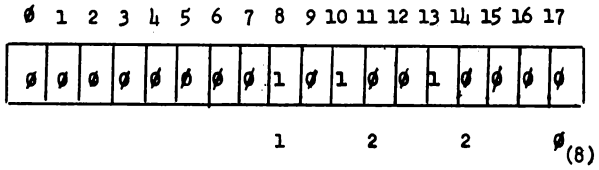
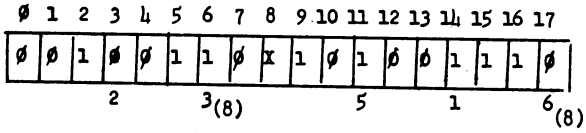


Figure 1-11. Base Address Register



$122Ø_{(8)}$ BAR
$+516_{(8)}$ in CPU
$1736_{(8)}$

X = Ø for Direct Addressing
X = 1 for Indirect Addressing

Figure 1-12. Instruction to be Executed

In Relative Addressing the location of the operand is dependent upon two things:

1. The type of addressing used (X = Ø, direct addressing and X = 1, indirect addressing, Figure 1-12).

2. The base address contained in the BAR (Figure 1-11).

If the indicator bit in Figure 1-12 were a "Ø," the actual operand would be located at core location $1736_{(8)}$ (BAR + DIRECTOR) and have a value of $567_{(8)}$, which would then be added to the register in the CPU. If the indicator were a "1" the address of the next register to investigate is $1736_{(8)}$. Since its indicator bit is "Ø", its contents, which contain a value of $341_{(8)}$, will be added to the register in the central processing unit.

Relative addressing can also be used in accessing program instructions. In this case a Base Program Register (BPR) is utilized. The absolute core location is relative to the value stored in the Base Program Register (BPR). The technique of using a BPR allows locating the same program in different core locations each time it is used.

Let's assume a small program is to be loaded into core from tape. Usually it will be located at the first available core area large enough to hold it. This location may not necessarily be the same each time the program is loaded into core. Where it will be loaded is dependent upon the core allocation at the particular instant the program is to be read in.

The usefulness of the BPR can be shown by the following example. Assume the program was loaded into core at location $2ØØØ_{(8)}$ and following. Suppose it has an instruction that tells the program to jump to instruction $1ØØ_{(8)}$ within the program. With no BPR, the program would go to absolute core location $1ØØ_{(8)}$ which is not in the program area and obviously incorrect. With the BPR loaded to the core location of the first instruction of the program ($2ØØØ_{(8)}$), the computer will branch to the correct location. The addition of the BPR ($2ØØØ_{(8)}$) and $1ØØ_{(8)}$ will cause the computer to jump to the proper core location ($21ØØ_{(8)}$). The computer will always get to the correct location in core regardless of where the program is located if the BPR is properly set.

IMMEDIATE ADDRESSING

Another type of addressing often used is called immediate addressing. In this case, the director portion is the actual operand, rather than the address of the operand. In Example #2, for instance, the datum added would be the actual value of the director, $516_{(8)}$. To indicate immediate addressing, either another indicator bit must be used or perhaps the command portion may use a separate command for immediate addressing.

In the examples of direct, indirect, and immediate addressing, a hypothetical method was used to indicate the proper method of addressing, which may or may not be similar to that used by a given computer. The main point is that when more than one option exists in an addressing system, there must be some method of indicating the mode of addressing that is desired.

It was previously mentioned that in variable word length computers instructions are also variable in length. In some variable word length computers (e.g., the 1410) the first character of an instruction has a word mark over it. Each instruction has one or more "legal" lengths, the length dependent on the number of options selected with that specific command. It can be seem that this provides for great flexibility and compactness in a computer program.

An example of a 1410 instruction would be: A$\overset{\vee}{\text{Ø}}$Ø5ØØØ1ØØØ which means "add the contents of location ØØ5ØØ to the contents of location Ø1ØØØ and store the results in location Ø1ØØØ." Another version using the same instruction would be A$\overset{\vee}{\text{Ø}}$Ø5ØØwhich means "add the contents of location ØØ5ØØ to itself and store the results at location ØØ5ØØ."

This type of computer has the disadvantage, however, of slow speed since data transfer is serial, transferring only one character at a time. Fixed word length computers can transfer data in parallel (i.e., a whole word at a time), since the words are all the same length and this method is physically possible. In a variable word length computer, although each character is transferred in parallel (i.e., all bits of one character at once), the transfer is serial by character, one at a time since the computer does not know the length of the word until it sees a word mark over a character.

Some computers, although primarily having a fixed word length, have a variable length instruction capability. An example of this is the Burroughs D825 Computer used in the BUIC System. The D825 has a fixed length 48-bit memory register. A single instruction can be a minimum of 12 bits long or a maximum of 84 bits long. Instruction flexibility like this allows for more power instruction repertoire. In operation, certain allocated bits in the operation code tell the computer what length or variations the complete instructions will have. Using this method of defining instruction length eliminates the need for word marks. In the D825, although it has a variable length instruction capability, data words are allowed in 48-bit single-register configuration only. Another example of a computer which operates with the same philosophy is the RCA, Central Data Processor, used in the AUTODIN System. Its instructions may range from 28 bits in length to 140 bits in length. That is, a command with no directors to a command with four directors. However, each operand is fixed at a half word (28 bits for this machine) in length. Data may be manipulated in form of 3, 4, 6, or 8 bit addressable characters, or half words. Note that the instruction words are composed of one or more half word units, while data may be composed of units varying in size from one 3 bit character to 28 bits (a half word).

ADDRESSING SYSTEMS

As has been mentioned previously, some computers are designed to use two or more addresses or director fields per instruction word. An ADD instruction in a computer utilizing

two directors may contain the location of data to be added to (augend) in one director wnd the location of the datum to be added (addend) in the other. Either one may be the address of the register in memory where the sum is to be stored. If the computer uses a three director instruction word, the third director would be the storage location for the sum. Systems using more than one director are called MULTIPLE ADDRESSING SYSTEMS. Systems using one director are single addressing systems. Figure 1-13 is a single addressing system. Figure 1-14 and 1-15 are multiple addressing systems.

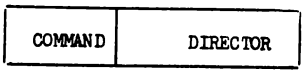


Figure 1-13. Single Address

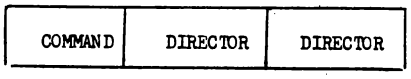


Figure 1-14. Two Directors

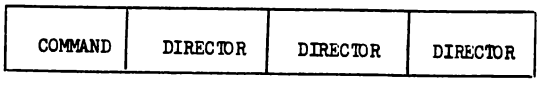


Figure 1-15. Three Directors

CHAPTER 2

FLOW CHARTING TECHNIQUES

The original specifications for a problem usually comes to the programmer in the form of an English prose statement. It is the responsibility of the programmer to translate this statement into a solution. After the method of solution is determined, it can then be coded into some language which is acceptable to a computer.

Problem solution may be accomplished in several different stages, possibly by several different individuals. It begins with the original statement of the problem to complte, concise statements of the methods of solution. The purpose of this block is to find methods of solving problems.

The statement of the problem solution completed by the formulator could very well be a simple English prose statement of the methods used. In order for it to be clear and concise, it could take on the form of an outline, or what is more commonly referred to as OPERATIONAL SPECIFICATIONS, where each line is numbered or lettered and describes a single operation, and the flow is indicated by references to other lines in the operational specifications. That is, the statement of the solution not only specifies the tasks which must be performed, but also the sequence in which they must be performed. Outlines of this type, while they do serve a function, can be difficult to follow, and would certainly increase the task of the coder if he were required to code from them. The solution to the problem can be stated far more clearly in a graphic form, so that inspection alone serves to point out the sequence of operations, and prose is used to describe them. Hence, the FLOW CHART, which is nothing more than a graphic representation of the method used to solve a problem.

The flow chart is not only a design tool, but is also a form of documentation for the final product, flow program. The flow chart serves to facilitate maintenance of the program and to clarify any interfacing with other programs in the same system.

FLOW CHART SYMBOLS

Many symbols are available from which a "language" for flow charting can be selected. The selections made are often programmer-specific, for no standardized set of flow chart symbols exist. In order for a flow chart to provide maximum communication it would see advisable to establish a standardized set of meaning at least within a specific organization or programming group. This is often done, and the conventions to be set down (here) are simply one such set of meanings . . . perhaps the most widely used.

Basically a flow chart consists of seven elements:

1. FUNCTION BOXES used to enclose the prose description of the operations performed in the program. A function box has one entrance and one exit.

2. I/O FUNCTION BOXES used to indicate operations pertinent to peripheral equipment (card reader, mag. drum, line printer, disc, CRT displays, cart punch etc.). An I/O symbol is an inverted trapezoid and has one entrance and one exit.

3. A DECISION SYMBOL, which represents a decision, has one entrance and two or more exits.

4. FLOW LINES used to indicate the sequence of operations performed by the program. Flow lines connect function boxes, decision symbols and connectors.

5. TEXT is used within symbols to describe data manipulations, data computations and to ask questions. Outside the function boxes test is used for comments and to describe the charateristics of a flow line.

6. CONNECTORS are used to connect remote portions of a flow chart. They do not indicate any operations.

7. FLAGS are programmer aids.

FUNCTION BOXES AND I/O SYMBOLS

Function boxes contain descriptions of the operations to be performed by the program. These operations include data manipulation and computations. Data manipulation includes the movement of information from one location to another location within core memory and the CPU. Computational operations include all the arithmetic operations - addition, subtraction, multiplication, division, and exponentiation.

The function box is represented in a flow chart by a rectangle. The text within this rectangle is kept quite brief, while still allowing it to completely specify the operation being performed. The following list includes the common symbols used in function boxes.

FUNCTION	SYMBOL	DESCRIPTION
exchange function:	==	exchange the contents of the left term with the contents of the right term
setting function:	=	set the left term equal to the contents of the right term
addition:	+	"plus"
subtraction:	-	"minus"
multiplication:	*	"times"
division:	/	"divided by"
exponentation:	**	"raised to the power"
absolute value:	|X|	"the absolute value of X"

Some examples of these are as follows:

Figure 2-1.

Figure 2-1 shows the transfer of information from location BB in core memory to location AA in core memory. BB retains its original value. However, the original value in AA is destroyed.

Notice that the equal sign (=) does not have its conventional meaning but is a setting or storing function, setting AA equal to the value of BB, so that they are equal after the operation is completed. If AA had a value of 25 and BB a value of 3, after the setting function, AA would contain a value of 3 and BB would have a value of 3.

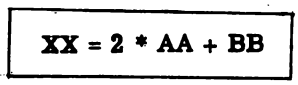


Figure 2-2.

Figure 2-2, a function box, indicates both an arithmetic computation and the storage of the computed value in the location XX. It reads, "Multiply AA by 2, add BB, and store this sum in location XX".

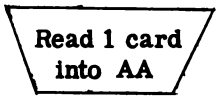


Figure 2-3.

Figure 2-3 is an I/O symbol which represents the transfer of information from the card reader into the internal storage location AA.

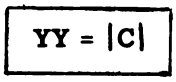


Figure 2-4.

Figure 2-4 reads "take the absolute value of CC and store it into YY". Suppose CC had a value of (-5). After this operation was completed, YY would have a value of (5).

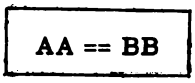


Figure 2-5.

Figure 2-5 reads "exchange the contents of location AA with the contents of location BB". If AA had a value of 1Ø and BB a value of 5, after the exchange AA would contain 5 and BB a value of 1Ø.

DECISION SYMBOLS

A decision symbol is represented in a flow chart by a diamond. It is the ONLY flow chart symbol from which more than one line of flow may exit, and it always contains an operation which can be stated as a question. The following is a list of common symbols which are used in decision blocks.

EQ	"equal to"
NQ	"not equal to"
LS	"is less than"
GR	"is greater than"
LQ	"is less than or equal to"
GQ	"is greater than or equal to"
:	"compared to"

Examples of decision symbols are as follows:

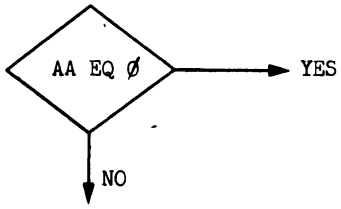


Figure 2-6.

Figure 2-6 asks if AA is equal to zero. The two flow lines leaving the box indicate the direction of flow for the two possible responses to the question.

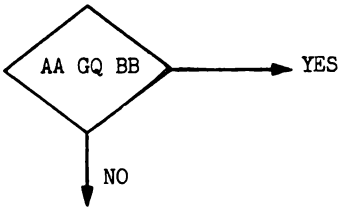


Figure 2-7.

Figure 2-7 asks the question, "Is AA greater than OR equal to BB?" If Yes, AA is either greater than BB OR equal to BB. If no, it is less than BB.

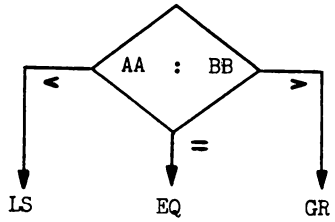


Figure 2-8.

In Figure 2-8 the colon is used to indicate comparison of two values, AA and BB. The three flow lines leaving the symbol indicate the three directions of flow, depending on the relationship which AA has to BB.

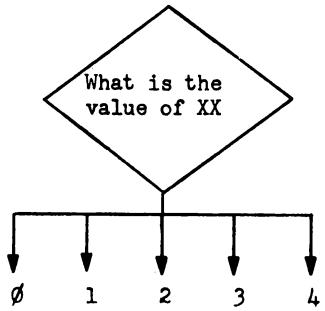


Figure 2-9.

Figure 2-9 represents a sequence of code which will inspect the item XX to determine its value. When such an operation is undertaken, the decision symbol must provide exits for all possible values of the item. This type of question is called a SWITCH.

FLOW LINES

The lines of flow in a flow chart terminate in an arrowhead and are used to indicate the sequence of operations in the program. Function boxes have only one line of flow entering them, and only one leaving them. Decision symbols have only one entry, but two or more flow lines leaving them.

EXAMPLE 1

Given: Core location AA, BB, and CC.

Required: If the value of AA is equal to the value of BB, set CC equal to a value of 1. Otherwise, set CC equal to a value of Ø.

Solution:

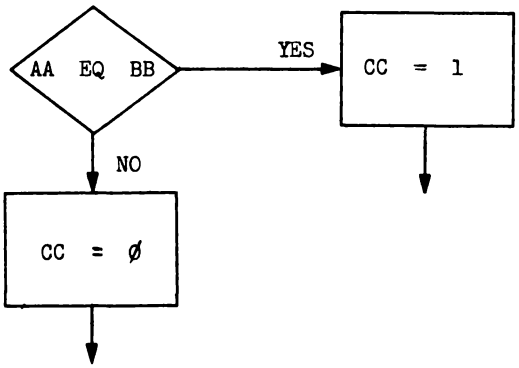


Figure 2-10.

FLAG

A flag is a symbol that appears along a line of flow.

EXAMPLE 2

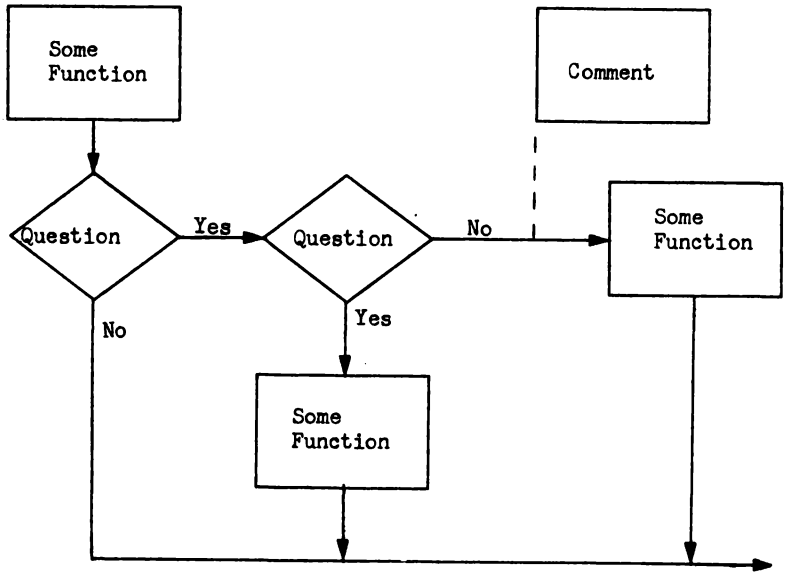


Figure 2-11.

Comments that appear within a flag are usually "nice to know" information. The comment never affects the program; it simply informs, or reminds, the programmer of some condition that exists. It is a programmer-aid to make reading the flow chart easier.

TEXT

Text is used within flow chart symbols to describe functions and ask questions. Text may also appear outside the symbols along the lines of flow or within a flag to indicate the meaning of the line of flow or for comments.

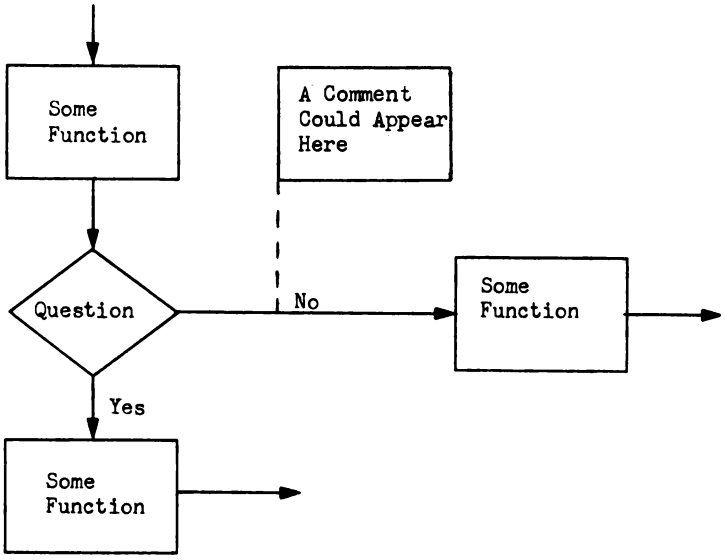


Figure 2-12.

CONNECTIONS AND TERMINAL SYMBOLS

Circles are used in flow charts to indicate entry points or exit points from programs. A special non-circle symbol, an off-page connector, is used when an exit is being made to another page. Since function boxes are entered by only one line of flow, and program logic frequently requires that the same function box be entered from several different exit points, there is a need to provide for the joining of flow lines using connectors.

The connector contains a label and is inserted into the flow at a minimum of 2 points--the exit point and the entry point. There can be only 1 entry point but 1 or more exit points. An indication of an entry point is simply a flow line from a connector. An indication of an exit point is a flow line to a connector. The entry connector and the exit connector(s) contain identical labels.

The oval is used to indicate the beginning or end of a program and will contain the word START or END. This symbol is also used to indicate the beginning or end of a subroutine and will contain the word START or RETURN. Subroutines will be discussed further in Chapter 7.

ON-PAGE CONNECTORS

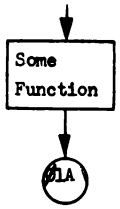


Figure 2-13. Exit Connector

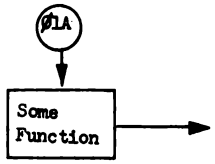


Figure 2-14. Entrance Connector

OFF-PAGE CONNECTORS -

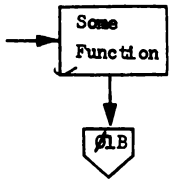


Figure 2-15. Exit Connector
(to another page)

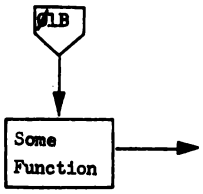


Figure 2-16. Entrance Connector

Given: The core locations AA, BB, CC, and DD containing values.

Required: If AA is greater than BB, set R1 equal to a value of 1; otherwise, set R1 equal to Ø. If AA is greater than CC, set R2 equal to 1; otherwise set R2 equal to Ø. If AA is greater than DD, set R3 equal to 1; otherwise set R3 equal to Ø.

In the following flow chart, (Figure 2-17), notice the use of the oval for the entrance and final exit from the program. Also, notice the use of circles for connectors AA, BB, and CC.

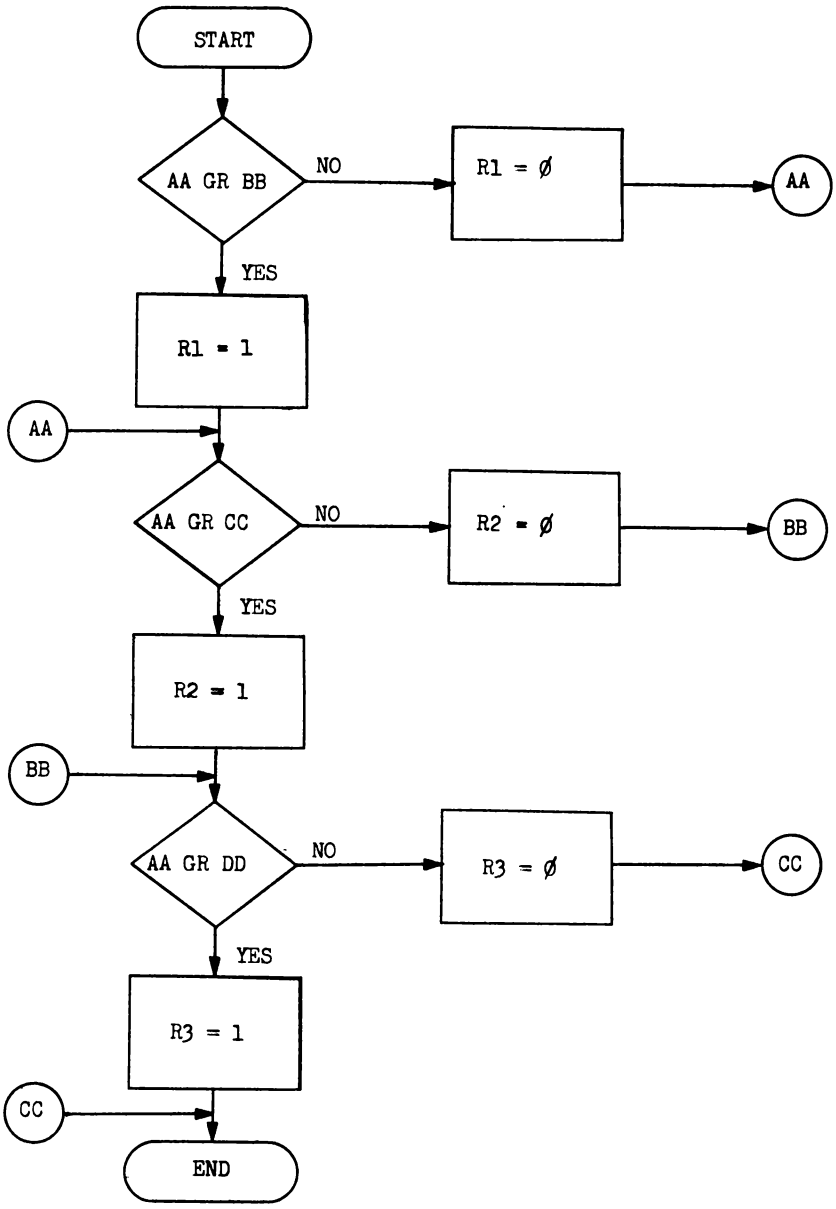


Figure 2-17.

LEVELS OF FLOW CHARTING

There are two levels of flow charting used in solving a problem.

1. A MACRO flow
2. A MICRO flow

A MACRO chart is very general and includes all major areas of the problem. It is used in obtaining a broad overview of the problem and its general solution.

A MICRO flow chart is much more detailed and indicates the precise sequence of events, step-by-step, that will take place in the solution of the problem.

Usually, a macro flow is written first and then is used as a guide; the micro flow is prepared in the detail that is required.

SUMMARY

The flow chart is simply a graphic representation of the method used to solve a problem. It is used in conjunction with data descriptions to provide the input to the coding phase of a program, as well as to provide program documentation for the benefit of those persons whose responsibility it is to maintain the program.

Certain conventions have been established for the form of a flow chart. Rectangles are used to represent any internal transfer of information from location to location as well as mathematical computations. The diamond is used to represent decision-making steps within the program's logic. Circles are used for entries and exits, and for connections between lines of flow. An I/O symbol is used to represent data transferred between internal and external storage. Only one line of flow will enter and leave a function box; the decision symbol has one flow line entering it, and may have several lines leaving it.

The language within the flow chart symbols will be brief, but will completely describe the operations being performed. Arithmetic operations are indicated by their normal symbols except for multiplication and exponentiation which are represented by one or two asterisks, respectively. Text may also appear outside the symbols along the lines of flow, or within a flag to indicate the meaning of the line of flow.

CHAPTER 3

INFORMATION ORGANIZATION

During the operation of a computer program, the main memory of the computer will contain two types of information. The first of these is the program instructions stored in a sequence determined by the programmer. The second type of information within the computer memory is the data with which the program works. Data occupies space in memory just as instructions do, and it cannot be distinguished from instructions by inspection by the computer.

Data in memory is structured to be meaningful and easily handled by the computer program. Thus structures of information inside the computer serve the same purpose as structures outside the computer, that is, to organize information for ease of access and use.

ITEMS*

Countless forms of one type or another, contain various items to be filled in; typical of these is an employment questionnaire. These forms provide space for the items to be filled in, each item being a single piece of information relating to the person filling in the questionnaire.

The items of information used by a computer are similar to these items. Computer items are single pieces of information relating to a specific object.

The items on a form are normally named, that is they have a title to identify what is to be filled in. The items used by computer programs are also named to identify them. Certain conventions for item names exist and are dependent upon the program language and/or computer used. One convention that is universally used is that an item name must be unique to the computer program using it.

Some items on a questionnaire are allocated more space for data than others. The guiding principle seems to be to allow sufficient space for the largest amount of information which the item might be called upon to contain. Items in computer programming are also allocated space on the basis of the size of the values they must contain. Items in theory could range in size from the smallest space available to all of the computer memory.

When referring to information on a questionnaire we distinguish between the name of the item and the information (value) contained there. In a similar manner a distinction must be made in computer programming between the name of an item and its contents. The name of an item in a computer program identifies the location of the information in the computer memory. The contents (value) of an item is the information contained at the location specified by the item name.

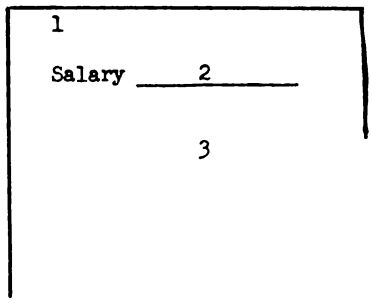
Definition: The NAME of an item refers to its location in memory while the VALUE of an item refers to the contents of that location.

For example, suppose we have an item name ABCD and its contents, 25. In this instance the item ABCD has the value 25. ABCD identifies the location in storage which contains the value 25.

It is important to understand that the name of item is not stored in the item, but only identifies its location in computer memory.

---

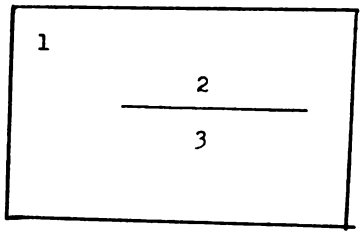
*A basic unit of information in data processing which is sometimes called an element of data.



1. ITEM NAME: Identifies and references LOCATION of item on form.
2. ITEM VALUE: Information contained in the item.
3. ITEM LENGTH: Space on the form occupied by an item.

Figure 3-1.

A single piece of information relating to a specific person.



1. ITEM NAME: Identifies and references LOCATION of item in the computer memory.
2. ITEM VALUE: Information contained in the item.
3. ITEM LENGTH: Space in computer memory occupied by the item.

Figure 3-2.

A single piece of information in computer memory relating to a specific object.

ITEM TYPES

Since the item is generally the BASIC unit of information in the computer, it must contain various types of information. The various item types describe the way in which information is represented in the item. The most common types of information in computer programs fall into three general categories: arithmetic, Hollerith, and logical. For each of these categories there are several ways of representing the information in the computer memory. The fact that reference is being made to a binary computer where representation of any kind of information must be made in binary digit configurations, must be kept in mind. Items are described or declared so that all users of the items know exactly what a particular item looks like and contains. Documents which describe items in a program are called compools or data dictionaries. When preparing to write a program to solve a problem, it must be determined what information is necessary to the solution of the problem. It must then be decided in what form this information will be stored in the computer memory, in other words the structure must be decided upon. Part of the statement of the solution to the problem, indeed a vary important part, is the complete description of the items to be used and their meanings.

ARITHMETIC ITEMS

Information which is strictly arithmetic, or numeric in character is represented as an ARITHMETIC item. The value (contents of an arithmetic item) is represented by its binary equivalent, which has mathematical meaning. There are two kinds of arithmetic items, fixed and floating point. A fixed point arithmetic item is dynamically accounted for by the programmer. On the other hand, a floating point arithmetic item represents a coding scheme where the binary digits have a varying precision that is given as part of the value and which is dynamically accounted for by the computer.

Arithmetic items are described or declared in terms of their name, type, location and size of their field, whether signed or unsigned, any fractional bits, and their units of measurement.

ARITHMETIC EXAMPLE:

Describe an item salary which will hold numbers from 0 to 1500 with a precision of 1/4. Before describing the item we must determine how many bits are needed. Since the only values are positive, the item does not need to contain a sign bit. The greatest magnitude (1500) needs eleven bits and two bits are needed for the fractional part of the value. Therefore the field for this item requires thirteen bits.

DESCRIPTION:

Item Name	SALARY
Item Type	Arithmetic Fixed Point
Beginning Bit Location[1]	1
Item Size[2]	13
Signed/Unsigned	Unsigned
Fractional Bits	2
Units of measurement	Dollar

HOLLERITH ITEMS

Information which represents specific alphabetic, numeric, or special characters is stored as a HOLLERITH item. Each different character is represented by a unique code. Hollerith items are used primarily for input/output and involve little or no manipulation.

HOLLERITH EXAMPLE:

Describe an item which will contain an employee name, where the maximum length of any employee name is eight characters. Assume the code for each letter occupies six bits. (6-bit Hollerith)

DESCRIPTION:

Item Name	NAME
Item Type	Hollerith
Beginning Bit Location[1]	1
Item size[3]	48
Coding Scheme	6-bit Hollerith

LOGICAL ITEMS

Certain information is neither arithmetic nor Hollerith. Instead its value denotes some logical meaning. This type of information is stored as a logical item. There are two subtypes of logical items: Boolean items and status items.

---

[1]Most significant bit (MSB) which is the left most bit of an item
[2]Item length or number of bits
[3]Item length in bits

BOOLEAN ITEM:

A logical item which is one bit in length is called a BOOLEAN item. It has two logical states (on or off, yes or no, etc.) These two states are denoted by the value of the item (Ø or 1).

BOOLEAN EXAMPLE:

Describe an item which will denote whether or not the line printer is available.

DESCRIPTION:

Item Name	PRNT
Item Type	Boolean
Beginning Bit Location	1
Item Size	1
Coding Scheme	Ø = no; 1 = yes

STATUS ITEM (VALUE):

A logical item which has more than two logical states (and thus occupies more than one bit) is called a STATUS item. Each value assumed by the item is associated with a particular logical meaning.

STATUS EXAMPLE:

Describe an item which will denote a certain employee's job. The various jobs are--apprentice, journeyman, craftsman, clerk, secretary, and supervisor. Since there are six possible jobs, the item must have six values. Thus a status item with three bits can be used as it can contain six values.

DESCRIPTION:

Item Name	JOB
Item Type	Status
Beginning Bit Location	2Ø
Item Size	3
Coding Scheme	Ø = Apprentice
	1 = Journeyman
	2 = Craftsman
	3 = Clerk
	4 = Secretary
	5 = Supervisor

BUIC ITEMS

The BUIC III system for item declarations follows the basic concepts presented. However, there are specific procedures to follow.

The procedures that follow are for COMPOOL sensitive items.* The format presented is unique to the BUIC III assembler. The examples that follow in this chapter will adhere to

*Compool sensitive items are items that:
1. Have fixed location in a register.
2. Can be accessed and manipulated by any program in core.
3. Can be accessed by item name reference as contrasted with a specific core location reference.

COMPOOL sensitivity. The other examples throughout this text will follow in format but will not follow with reference to the number of letters that compose an item name. The same will be true for table names.

The standard COMPOOL sensitive item format for the BUIC III assembler is as follows:

ITEM	NAME	PARENT TABLE	BLOCK NUMBER	STARTING BIT	NUMBER OF BITS	ITEM TYPE	SCALING

ITEM - Specified an item declaration.

NAME - Name of the item.

PARENT TABLE - Name of table of which item is part.

BLOCK NUMBER - Specifies the block number of the table in which the item is located.

STARTING BIT - Specifies the starting bit position of the item within the computer word.

NUMBER OF BITS - Specifies the number of bits assigned to the item.

ITEM TYPE - Specifies the item type.

SCALING - Allocates bits for the integral and fractional portion of signed and unsigned items.

The permissible code for the ITEM TYPE is as follows:

B = BOOLEAN

A one-bit indicator treated as one item.

U = UNSIGNED

Any unsigned number. If the number is an integer, it need not have scaling associated with it. If it contains fractional bits, then scaling must be associated with it.

S = SIGNED

Any signed number. Again, it may or may have scaling associated with it.

V = VALUE (STATUS)

The value of each configuration of bits in the item has a specific meaning, i.e.,

1 = north
2 = east
3 = south
4 = west

M = MIXED

Items defined as M type are assumed to include more than one distinct piece of information grouped as one item for programming convenience.

T = TRACK

Items defined as T type are special items that contain track numbers for the BUIC III tracking system. The track numbers are coded in 4-bit switch code and the meaningful bits are 1-16. The item representing the track number is defined as 18 bits to take advantage of BUIC III coding conventions.

D = BINARY CODED DECIMAL

The item is divided into a series of four bits, each set of four bits representing a decimal digit.

H = HOLLERITH

Alphanumeric code that is assumed to be in Burrough's format.

C = CHARACTRON

That code which generates symbols on situation or tabular display scope.

EXAMPLE 1

Problem: Describe an item which will accommodate numbers ranging from -1ØØ to +12Ø with a precision of 1/32.

Solution: The item will, first of all, be signed, since its possible value include both positive and negative numbers. The maximum integral value which the item might have is 12Ø. Since 12Ø is greater than $2^6 - 1$ (63: the largest value which can be contained in 6 bits) but less than $2^7 - 1$ (127: the largest value which can be contained in 7 bits), seven integral bits must be allocated. The precision called for is 1/32, which will require 5 fractional bits. The item then, will have a total of 13 bits allocated. It will be signed, with 7 integral bits and 5 fractional bits. The name of the item is arbitrary, let us call is NUMB.

```
ITEM   NUMB   FOPØ1   1   13   S   Ø7.Ø5
```

EXAMPLE 2

Problem: Describe an item to contain salaries for employees, where the maximum salary is $1000, and no salaries are specified more precisely than 25¢.

Solution: The required item, MONY, will be unsigned (assuming all salaries are positive); it will have 1Ø integral bits, since 1ØØØ is less than $2^{10} - 1$ (1Ø23), and 2 fractional bits to represent values no more precise than 1/4. The total number of bits in SALARY is 12.

```
ITEM   MONY   FOPØ2   1   14   U   12.Ø2
```

EXAMPLE 3

Problem: Describe the item(s) necessary to contain today's date.

Solution:

First, one item is used which has a maximum value of 123199, where the first two digits represent month, the second two digits day, and the last two digits represent the last two digits of the year, where the century is implied; second, three items can be set up MO, DA, and YR, where each has its respective maximum. The first method will require 17 bits, since the value 123199 is less 2^{17}, 131Ø72. The second method will require 4 bits for MO (since 12 is less than 2^4), 5 bits for DA, and 7 bits for YR, for a total of 16 bits to contain the required information. The second method seems to have two advantages; the first is space, if one is concerned about saving a bit, and the second is the logical format of the information, which is much more appealing split into the three items than assembled into the one large number.

```
ITEM   DATE   TIMØ1   1   16   U
```

EXAMPLE 4

Problem:

Describe an item which will be capable of containing distances over a range of Ø.1 miles to 1,ØØØ,ØØØ,ØØØ,ØØØ miles, where order of magnitude is of more importance than precision.

Solution:

The integral portion of this value, if it were represented in fixed point format, would require 4Ø bits. While this is within the range of possibility for some machines, it is far too large for others. The addition of the fractional portion of the item would make it still worse. The obvious solution to the problem is to describe the item as floating point, occupying one word of computer memory. The item's name might be DIST.

```
ITEM   DIST   MILØ1   1   48   F
```

EXAMPLE 5

Problem:

Describe an item to contain card suit for a program which will play bridge.

Solution:

The suit progression for bridge is from clubs at the lower extreme, through diamonds and hearts, in that sequence, to spaces at the upper extreme. This item could be described as Hollerith, but would require eight characters to accommodate the word diamonds, and therefore 48 bits. If the item is described as a status item, the value zero can correspond to clubs, 1 to diamonds, 2 to hearts, and 3 to spaces. The item will occupy 2 bits, as one advantage, and, from the programming standpoint it will be possible to inquire of the item whether it is greater than zero, for example, which is saying "is it greater than clubs?" The item SUIT, then, is a 2-bit status item where zero corresponds to clubs, 1 to diamons, 2 to hearts, and 3 to spades.

```
ITEM SUIT PLYØ1  1 2 V(Clubs) V(Diamonds) V(Hearts) V(Spades)
```

V( ) - is every possible status. Each status must be enclosed in parenthesis and preceded by the letter "V".

Problem: Describe an item which will indicate whether the weather is or is not rainy.

Solution: The item that will fit the problem perfectly is the Boolean type item. Using the Boolean item RAIN, we can let zero equal false; it is not raining, or we can let one equal true; it is raining.

```
ITEM    RAIN    WETØ1    12    1    B
```

The programmer, preparing to write a program to solve a problem, must first determine what information is necessary to the solution of the problem. He must also decide the form in which this information will be stored in computer memory . . . in other words he must design the required items. Part of the statement of the solution to the problem, indeed a very important part of it, is the complete description of the items to be used and their meanings.

TABLES

The information required for the computer solution of a problem frequently falls naturally into a tabular organization, as does information for manual processing. Since the BASIC UNIT OF INFORMATION is the item, it will also appear in this context. Frequently, several items will be used to describe an object, and this set of items will be repeated to describe a series of objects. The structure describes it as a TABLE. A telephone directory is as good an example of a table as one could hope to find. The items appearing in it are name, address, and telephone number. They are repeated for as many times as there are objects, people, to describe. The set of items which describe a specific object is known, in computer terminology, as an ENTRY. Thus, one line in a telephone directory, containing one name, one address, and one telephone number is an entry. The table, then, is a collection of entries describing objects which have been classified together. In tables constructed for computer usage, entries are generally numbered, beginning with ZERO by convention. The last entry in a table will, therefore, have a number one less than the total number of entries in the table (n-1). The space occupied by an entry is known as a slot. Its size depends on a number of factors which include the number and size of the items in the entry. Within any one table, the size of the slot will be constant. When the size of the slot is constant, the entry is referred to as a FIXED LENGTH ENTRY. When the entries contain identical items, the entries are referred to as FIXED STRUCTURE ENTRIES.

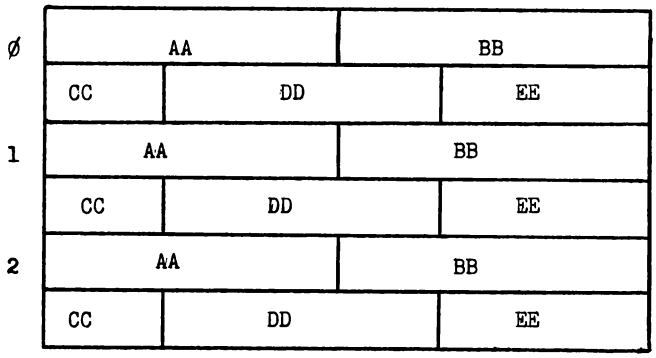


Figure 3-3. Fixed Length and Fixed Structure Entries

Figure 3-3 is an example of fixed length and fixed structure entries. The size of a slot is traditionally an integral number of words, one or greater. Tradition is mentioned here because

it is certainly not impossible to construct and work with tables which have slot sizes which are not integral multiples of a word. It is, however, much easier to deal with tables which conform to this tradition, and many programming languages impose this constraint.

Since tabular information is so greatly used in programming, it is necessary to distinguish between items which appear in tables and items, such as those described above, which appear outside the environment of a table. The distinction is made by calling the items in tables "TABULAR ITEMS," and those outside tables either "NONTABULAR" or "SIMPLE".

TABLE DECLARATION STATEMENT

In the preceding section the format for coding item declaration statements was given. These formats are used whether an item is non-tabular or tabular. When one or more items are to be organized as part of a table which is the more typical use, all items within a single ENTRY of the table are defined consecutively. The entire entry is preceded by a table declaration statement to completely define the format of a table. The format for definition of a table is as follows:

TABLE NAME NUMBER WORDS NUMBER BLOCKS WORDS/BLOCK TABLE TYPE

TABLE - Specifies a table declaration.

NAME - Name of the table.

NUMBER WORDS - Specifies the number of words in the table. It does not include the control word for variable length tables.

NUMBER BLOCKS - Specifies the number of blocks comprising the table.

WORDS/BLOCK - Specifies the number of words in each block of the table. Equal to the number of entries in the table.

TABLE TYPE - Specifies the table type.

The permissible code for the TABLE TYPE is as follows:

STRUCTURE	LENGTH	ORGANIZATION
R = rigid	R = rigid	P = parallel
V = variable	V = variable	S = serial

BUIC III COMPOOL limits table names to three characters. Item names are limited to four characters. The rest of this text does not conform to these limitations. With this exception, all item and table declarations are correct, relative to COMPOOL format.

TABLE DECLARATION EXAMPLES:

Define a table, ADI, that is 48 entries long, serial organization and rigid structure.

```
TABLE ADI     48     Ø1     48     R R S
```

Define a table, GZC, that is 45 entries long, with two blocks, parallel organization, and rigid structure.

```
TABLE GZC     9Ø     Ø2     45     R R P
```

Define a table, MSG, that is variable length, with two blocks, serial organization and rigid structure.

```
TABLE MSG     1Ø     Ø2     Ø5     R V S
```

The above examples are COMPOOL sensitive.

TABLE PACKING

The size of an item is determined by the MAXIMUM VALUE of which the item is capable. When items are grouped together into entries, certain problems arise with respect to their allocation into words and parts of words. The problem of this arrangement is called packing, and is one of the factors involved in the DETERMINATION OF SLOT SIZE. It is obvious that, if a table is to contain three items, AA, BB, and CC, and each of them occupies no more than a word of computer storage, these items could be arranged so that each is RIGHT-JUSTIFIED WITHIN A WORD of storage and the leftover bits are not used. In this case, the size of the slot would be three words. However, it might also be possible, depending on the exact size of the items and on the size of the computer register, to PACK two, or perhaps all three, of the items into the same word, thus conserving one or two registers in each slot. The first method is not entirely without its advantages--true it may waste considerable SPACE, but, on the other hand, TIME will no doubt be gained in the processing of the tables since full words can be referenced and no special instructions are needed in order to access an individual item. The conflict here is one which continually confronts the programmer . . . the necessity for a trade-off of TIME versus SPACE. The decision, of course, depends largely on the exact requirements of the problem . . . in some cases, time is far more valuable than is space. In many cases, the reverse is true. In most cases, however, both time and space are at a premium, and the programmer must try to optimize the use of both.

When items of no more than one word in length are stored separately in registers, they are called UNPACKED. When some or all of the items in a table are stored together in one word, the table is called PACKED, as are the items in it. In arranging to pack a table, the programmer need keep in mind only one rule, and that is that NO ITEM IS SPLIT between two words of computer memory. This rule is simply a convention, established because of the serious problems in processing caused by having a single item stored partly in one word and partly in another.

EXAMPLE 7

Given: The items XX, 7 bits long, YY, 2Ø bits long, and ZZ, 12 bits long are to be stored in a table.

Required: Show the arrangement of words of a typical entry containing these items if the computer to be used has a 48-bit word and space is to be conserved.

Solution: Since the total number of bits occupied by the three items is under 48, the entire group of items can be packed into one word. The arrangement of these items within the word is arbitrary, in this case, and could be as follows:

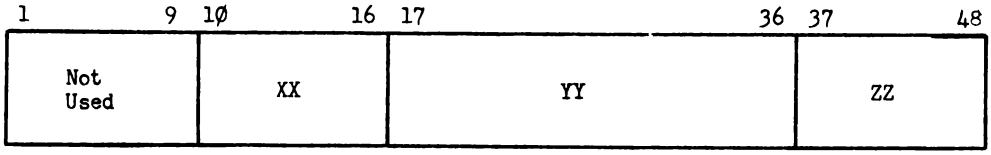


Figure 3-4. A Packed Entry of 3 Items.

The numbers above the diagram in Figure 3-4 are the bit positions allocated to the items, where the bits are numbered from one at the left.

EXAMPLE 8

Given: The items PT'NO, 45 bits long, MAKE, 3 bits long, QTY'OH, 10 bits long, YEAR, 14 bits long, and PRICE, 24 bits long, in Figure 3-5 are to be stored in a table.

Required: Show the arrangement of the words of a typical entry containing these items if the computer word being used is 48 bits long, for both the cases where storage space need not be conserved.

Solution: In order to conserve storage space, it is necessary to pack the table containing these items. Allocating space from left to right within words, and taking the items in the order in which they are listed, the following arrangement is produced.

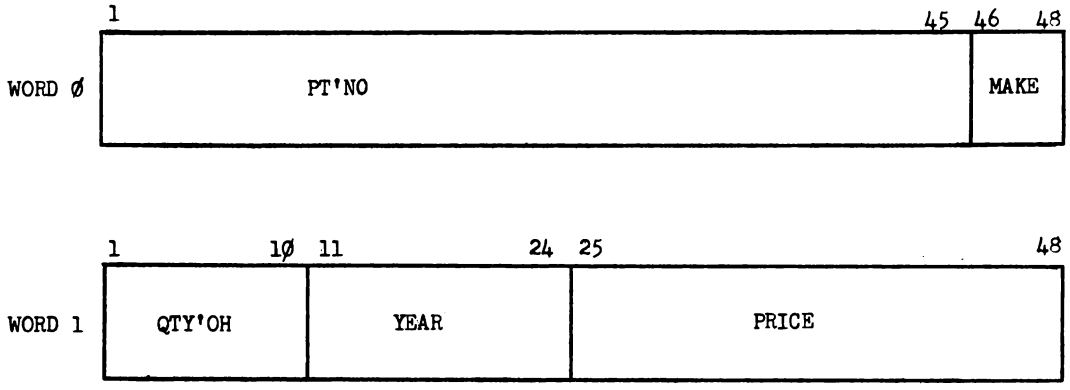


Figure 3-5. A Two-Word Entry.

Notice in Figure 3-5, that the listing of the items was quite fortunate. Had they been listed in a different order, would the same results have been produced? If not, would the difference have lengthened the slot from two words to three or more?

If storage space need not be conserved, the following arrangement of the entry will be produced:

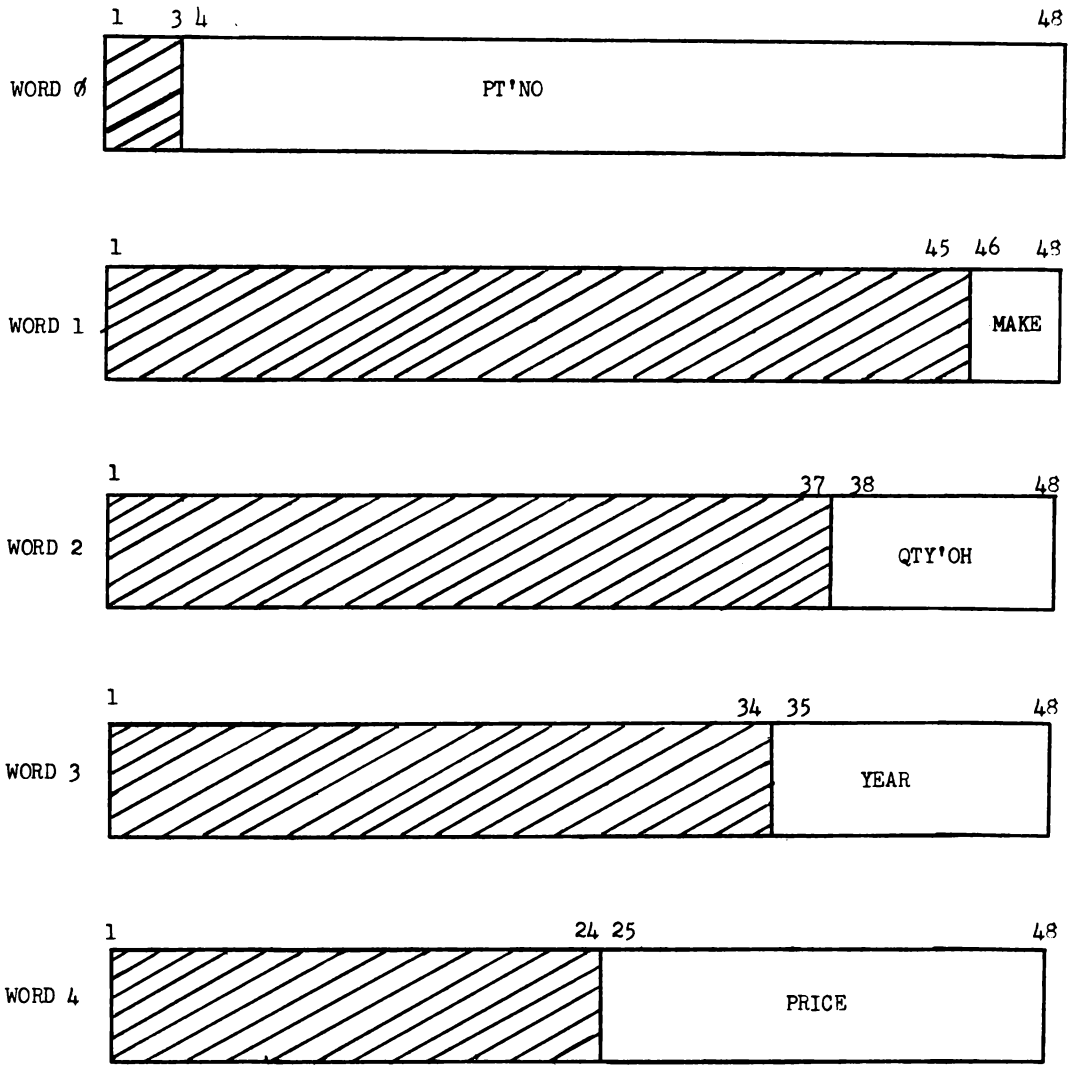


Figure 3-6. Unpacked Entry of 5 Words.

Concern with the question of packing items into tables actually requires, as is obvious from Figure 3-6, a certain amount of knowledge of the computer to be used. Because this matter falls into this realm, the topic will not be discussed beyond the level just presented. The topic which follows will also rely to some extent on reference to the computer register, but it has, perhaps, more general applicability than does item packing.

TABLE STRUCTURE

Depending on the SEQUENCE OF THE WORDS of the entry and the sequence of the ENTRIES themselves, tables may be categorized in two ways. Although this area is one apart from table packing, the two are connected in that table packing determines the size of the slot.

The most obvious method for organizing a table will be discussed first, and is called serial structure. This name is used because tables organized in this way have the words of an entry in series and the entries in series. The structure will be explained by means of an example.

EXAMPLE 9

Problem: Show the table described in example 8, given that it is 1ØØ entries long, beginning at register 2ØØ of computer memory, and that it has a serial structure.

Solution: (1) When the table is packed

```
TABLE    PRO    2ØØ    2    1ØØ    R R S
```

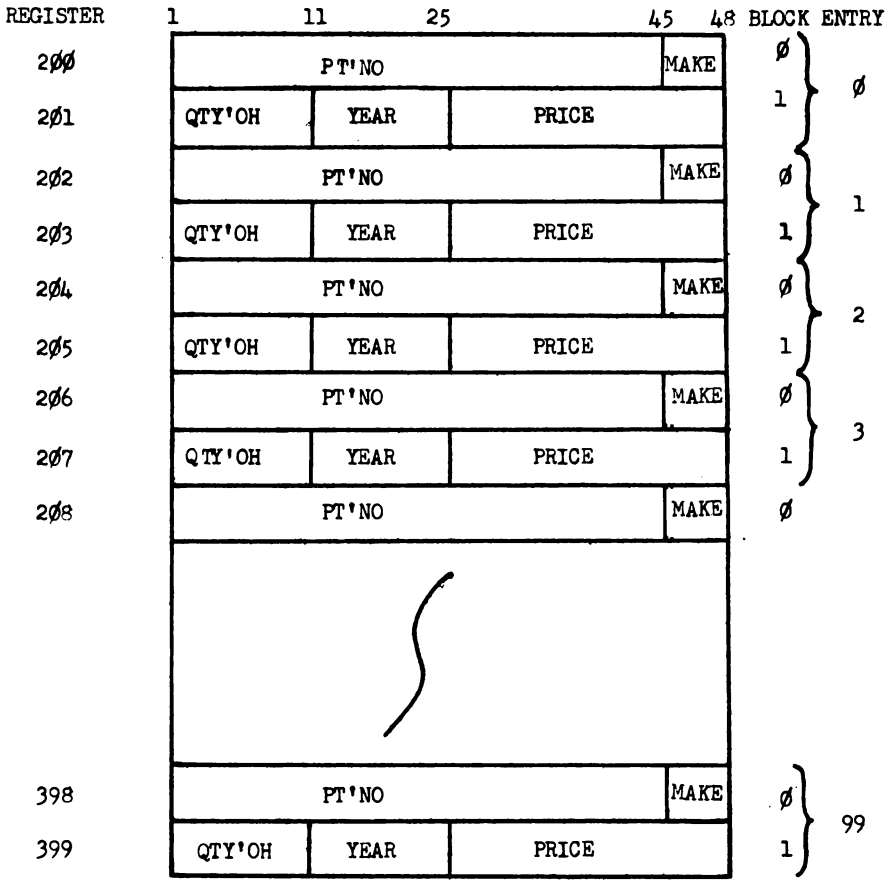


Figure 3-7. Packed Table.

Notice that all slots are two words in length, fixed length, and that each entry is identical to all other entries, fixed structure. Notice also that the table occupies 2ØØ registers of memory . . . the number of entries times the number of words per entry.

(2) When the table is not packed

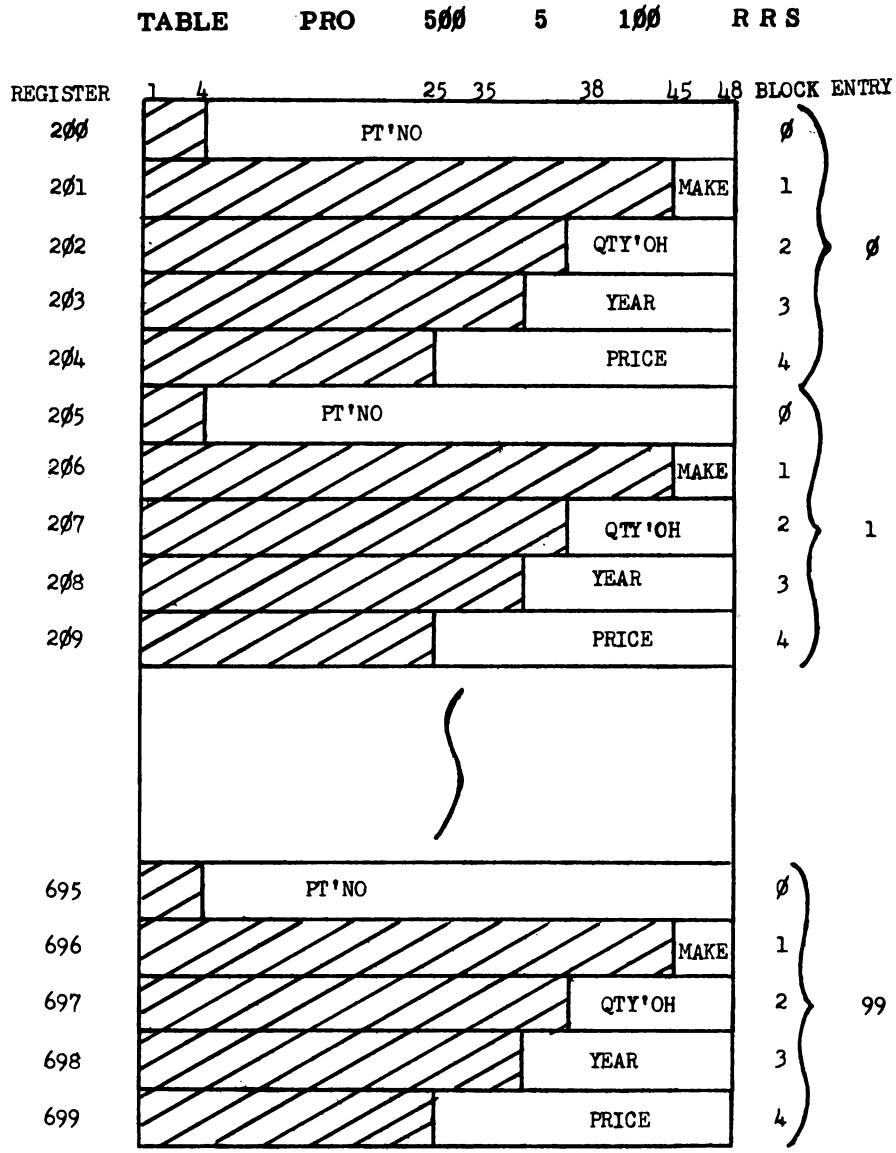


Figure 3-8. Unpacked Serial Table.

Notice in Figure 3-8 that, although the table length has increased considerably due to the increase in slot size, the properties of the serial table remain the same. The words of an entry are stored in series in a serial table, and the entries are stored in series. The second method for organizing tables, called PARALLEL STRUCTURE, divides the entry into SUB-ENTRIES of one word in length. These sub-entries are grouped together into BLOCKS, where a block is simply defined to be a set of sequential registers, and the blocks are grouped together

to form the table. Once again, the principles can be described most effectively by means of an example (Figure 3-9).

EXAMPLE 10

Problem: Show the table described in example 8, given that it is 1ØØ entries long, beginning at register 2ØØ of computer memory, and that it has a parallel structure.

Solution: (1) When the table is packed

TABLE PRO 26Ø 2 1ØØ R R P

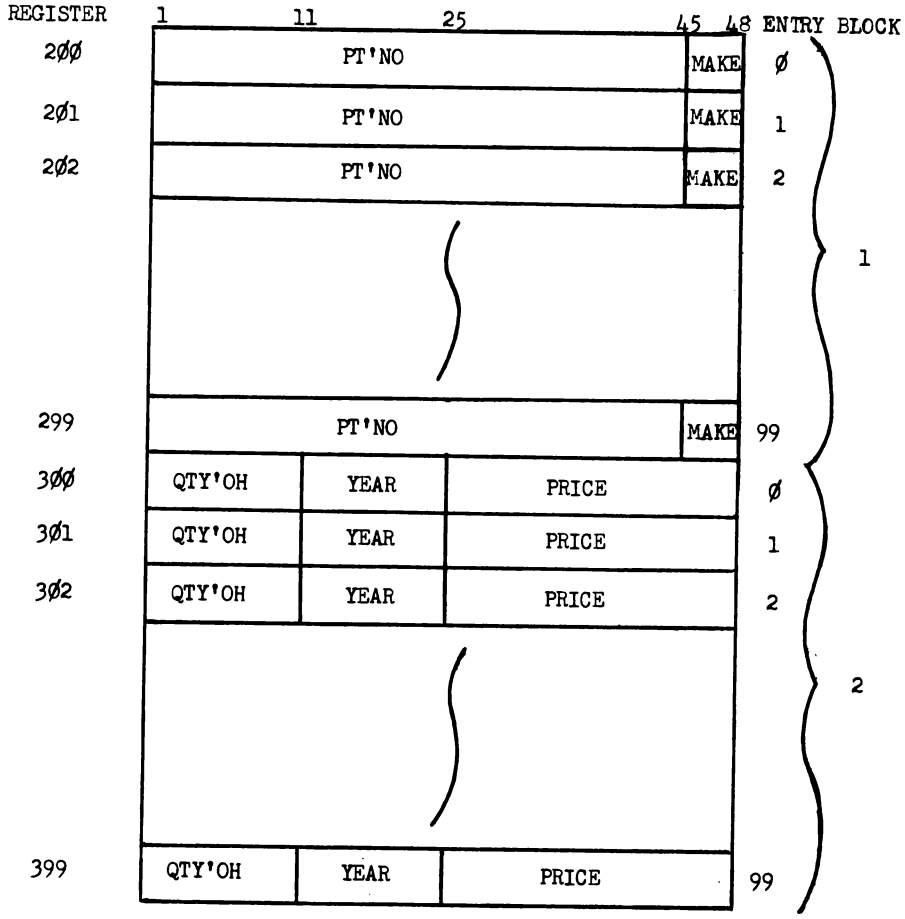


Figure 3-9. Parallel Packed Table.

Notice in Figure 3-9 that all of the registers in block one contain exactly the same items, arranged in the same format; the same is true of block two. The table is called parallel because the RELATIVE LOCATIONS of all of the words of a given entry are equal, where relative means relative to the beginning of the block. The beginning of block one is register 2ØØ; relative to that, the first word of entry 4 can be computed by adding four to the beginning address of block one . . . namely, 2ØØ. Thus, in a parallel table all the words of an entry have the same relative locations with respect to the beginning of their blocks. This characteristic of the parallel table provides ease of manipulation in machine code for some computers. The difference between processing the two types of tables will not be detected, however, in flow charting.

(2) When the table is not packed.

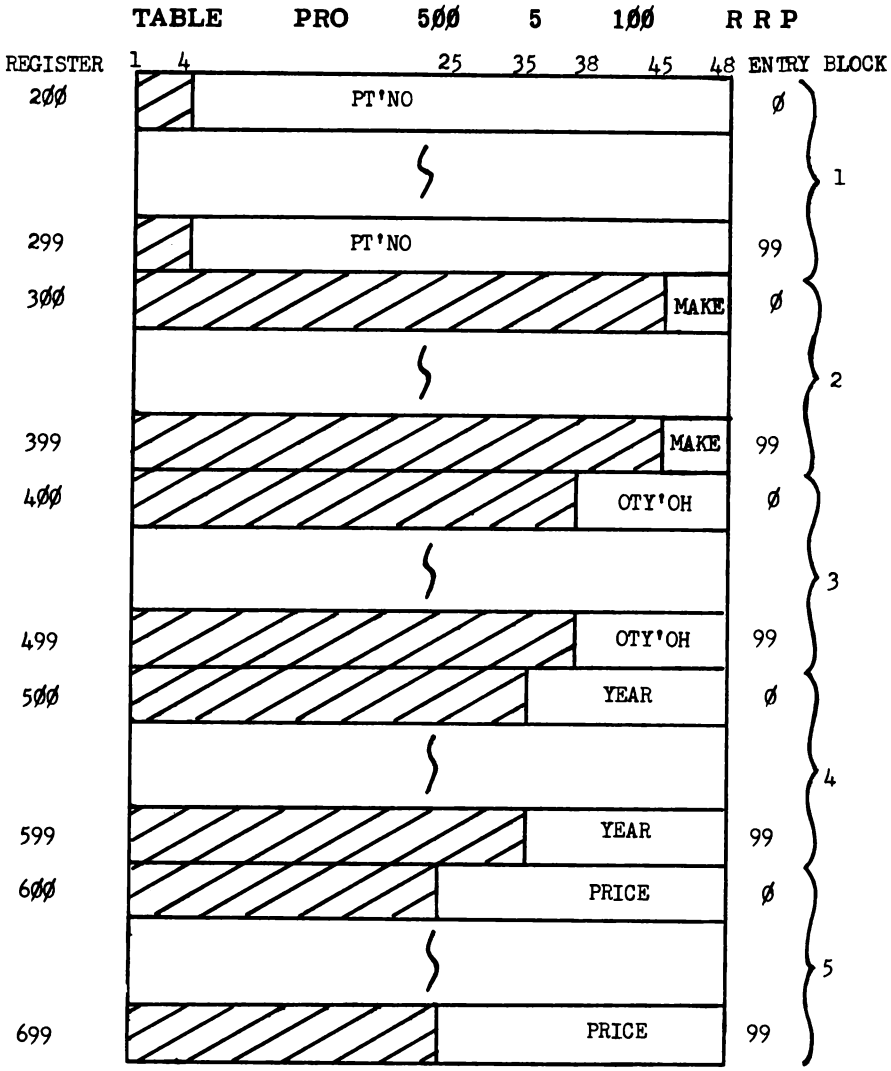


Figure 3-10. A Parallel Unpacked Table.

Notice that, in Figure 3-10, there are five blocks of 1ØØ registers each, and that, once again, the relative address of each word of a given entry is the same as the relative address of each word of the same entry.

SUMMARY

Tables, are composed of ITEMS grouped together into entries containing information about the same object. The entries, which occupy SLOTS, contain information about related objects. The size of the slot is determined by the size and number of the items in the entry, and the type of PACKING which is used. The size of the table is determined by the number of entries and the size of the entry. The table may be organized in either SERIAL STRUCTURE, where the words of an entry are consecutive and the entries are consecutive, or in PARALLEL STRUCTURE, where identical one-word sub-entries are grouped together into BLOCKS, and the blocks are grouped together to form the table; noting that where the entry is one word in length, no distinction can be made between serial and parallel structure.

CHAPTER 4

ORDERING SCHEMES

INTRODUCTION

In addition to table design and item types, the problem of information organization for computer handling also includes considerations with respect to the order, or sequence, of the information stored in the entries of the table.

Most tables will contain more than one item per entry, and one or more of these items may be chosen as "key" item(s). A KEY ITEM is that item in a table which governs the order of the entries, and by which entries are normally accessed.

Although the selection of the key item is not the topic to which this chapter is addressing itself, it is appropriate to mention that this selection is amost entirely based upon the need of the particular processing problem. A personnel table, for example, which contains such items as MAN'NO, NAME, AGE, SALARY, DEPENDENTS, DEPT, JOB'CODE, SUPERVISOR, etc., could be ordered on the basis of almost any one of these items. However, for payroll purposes MAN'NO or NAME would probably be chosen. For the purpose of keeping track of transfers from department to department, a table ordered on the basis of DEPT might be the most efficient.

Once a key item has been chosen, what ordering schemes are available? Four such schemes are to be considered here. They are:

1. Ascending order
2. Descending order
3. Unknown order
4. Consecutive order

ASCENDING ORDER

Let us first consider the ascending order. The reader is familiar with this sequence scheme, for it is found in many ordinary collections of information. The telephone directory, for example, is organized in ascending sequency by name. The dictionary is organized in ascending alphabetic sequence. The table of contents of any book is organized in ascending sequence by chapter number or by page number, while the index is in ascending alphabetic sequence.

The essential characteristic of ascending sequence is simply that the key item increases by some POSITIVE VALUE from one entry to the next. This characteristic does not prohibit an increase by zero; or, in other words, the key item from successive entries MAY be identical.

The following examples illustrate ascending order:

EXAMPLE 1

Entry Number	MAN'NO	SALARY
Ø	ØØØ4	1ØØ.ØØ
1	ØØ1Ø	219.5Ø
2	ØØ2Ø	89.75
3	ØØ21	15Ø.ØØ
.	.	.
.	.	.
.	.	.
999	7463	5ØØ.ØØ

MAN'NO is the key item in this table, which is ordered in ascending sequence. Since man number is ordinarily unique, no duplicate keys will be found in this table.

EXAMPLE 2

Jones Margaret 426ØNeoshoAvMN . . . . . EX 7-9424

Jones Margaret 1943 2Øth SM. . . . . EX 4-3693

Jones Margaret B 1742 MalclmAvWLA . . . . . GR 8-2227

This example is taken from the telephone directory. Notice the increase of zero in the key item, NAME from the first entry to the second. Neither address nor telephone number appears to be an auxiliary key in this case, for no increase is apparent in them from the first entry to the second.

EXAMPLE 3

Entry Number	DEPT	JOB'CODE	YEARS
Ø	Ø1	Ø2	1
1	Ø1	1Ø	5
2	Ø1	23	2
3	2Ø	Ø5	1Ø
4	2Ø	75	1
.	.	.	.
.	.	.	.
.	.	.	.
675	98	12	4

Example #3 is taken from a personnel table which contains two key items; the item DEPT is the primary key, while JOB'CODE is a secondary key. Within one department, the entries of the table are organized in ascending sequence on the item JOB'CODE. The departments

themselves are in ascending sequence. In this case, it is entirely possible that the combined key DEPT-JOB'CODE increase by zero, producing consecutive entries with the same identification.

Ascending sequence provides ease of access for tables whose key items do not increase in a rigid pattern, but for which some ordering scheme is desireable due to the processing for which they are intended. For example, looking up information in a table is a great deal easier when the table is organized. Can you imagine looking up phone numbers if the telephone directory has no organization? The same sort of problem exists in computer memory. Furthermore, additions to a table and deletions from it become a great deal easier if the table is organized in a logical sequence.

DESCENDING ORDER

Descending order is simply the reverse of ascending sequence. It is characterized by the fact that the key item(s) decrease (though the amount of decrease may be zero) from one entry to the next. The processing of a table organized in descending sequence will not differ a great deal from the processing of one organized in ascending sequence. It has the same advantages as does the first type. If this is the case, why will descending sequence ever be chosen in preference to ascending? The answer to this question lies solely in specific cases. (Index registers, which are usually used for these purposes partially govern the choice.) If, for instance, the higher order values of the key item must be accessed most frequently, AND the equipment being used functions best when memory locations are accessed from lower to higher, then descending sequence will indeed be preferable. In short, then, the choice between these two methods depends almost entirely on the method of accessing and the computer to be used.

UNKNOWN ORDER

The third scheme listed is unknown (random) sequence, or the complete lack of a known sequence. It seems somewhat paradoxical to state that no sequence is a type of sequence; however, let us persue random sequence to determine its advantages, if indeed it has any.

Let us first consider the means by which "organized" tables, that is, those in ascending or descending sequence, get to be that way. In order for a table to be organized into ascending or descending sequence, one of two things must occur. Either the information being entered into the table is sorted (either manually or by EAM equipment) before it is stored in the table . . . a long and boring process if the table is at all large, but a cheap one . . . or, once in the table in random sequence, the information must be sorted under program control. The latter of the two methods constitutes a large area in the scope of programming skills. There are many ways to sort a table under program control, most of which will be covered later. Suffice it to state that, at best, when they are concerned with large amounts of data, sorting programs consume a great deal of time. Thus, program time can be cut down appreciably if sorting is not required.

This leads us to consider when "random" sequence can be effectively employed. There are many cases in which all the information in a table must be processed and the order in which it is processed is of no consequence. Let us borrow an example from the BUIC system, in which radar returns must be processed and displayed on the consoles of the military personnel who will make decisions based on these data. The returns come into the BUIC system from remote radar locations, arriving and being stored on the drums in random sequence. When the computer system is ready to process these radar returns and to display them, they must all be displayed (and processed) and there is no advantage to picking a certain sequence of processing. In this case, therefore, it is certainly not advisable to spend the computer time necessary to sort these data into any sequence.

We might consider that a payroll operation might just as well be completed in random sequence, since at each processing interval all entries must be processed, and there is no advantage to the processing system in selecting a sequence. However, in this case, there is, clearly, a point in time at which the results of the processing are much more useful if they are sorted. Distribution of the resulting paychecks can be much enhanced if the paychecks are already in reasonable sequence. Further, a payroll table, once sorted can remain sorted unless additions or deletions are necessary. Thus, the sorting operation need not be repeated often. And, last, the same table which is used for the payroll operation might also be used for other operations which require an organization scheme. In general, the use of unknown sequence is advantageous when it occurs naturally and a particular known sequence is not beneficial in the processing.

CONSECUTIVE ORDER

Finally, we come to the consideration of the consecutive sequencing scheme. This scheme is bounded by very rigid constrains and is highly advantageous when it can be used. The consecutive sequence requires that the key item(s) increase BY THE VALUE ONE from one entry to the next. It is, therefore, a modification of the ascending sequence. Its advantage lies in the ease with which information can be accessed when the key increases in this fashion. In fact, no "looking" is necessary in order to find a particular entry . . . if the key is known, the location is also known. If the values of the key do not increase by one, naturally, this scheme has a disadvantage in the space wasted by meaningless entries.

Examples will best illustrate the possible variations of this scheme:

EXAMPLE 4

Entry Number	ANGLE	SINE	COSINE
Ø	Ø	.ØØØØØ	1.ØØØØØ
1	1	.Ø1745	.99985
2	2	.Ø349Ø	.99939
.	.	.	.
.	.	.	.
.	.	.	.
89	89	.99985	.Ø1745

In this excerpt from a table of the values of trigonometric functions for angles given in degrees, it is clear that the key item ANGLE is consecutive in the entries of the table. Furthermore, since the values began at zero, they correspond exactly to the entry numbers for the entries in which they are stored. In such as case as this, the key item is actually eliminated as unnecessary material in the entry. Given an item IN'ANGLE for which we would like to find the sine, we simply refer to the item SINE in the entry whose number is equal to the value in the item IN'ANGLE. Such tables as those of trigonometric functions are ideally suited to organization in consecutive format, for none of the values of the key are useless or blank. When there are large gaps in the sequence of values of the key, such an organization is surely space-wasting.

EXAMPLE 5

Entry Number	PART'NO	QUAN
Ø	1ØØ	25
1	1Ø1	1ØØ
2	1Ø2	2
3	1Ø3	Ø
.	.	.
.	.	.
.	.	.
99	199	57

In example #5 from a parts inventory, the key item PART'NO is consecutive, though it does not begin at the value zero. Its value is, nonetheless, related directly to the value of the entry number of its position . . . clearly, the information for part number 125 is stored in entry 25. As in the previous case, the key item does not necessarily have to appear in the table, when the table is consecutive.

The words "CONSECUTIVE" and "SUCCESSIVE" are very closely associated in meaning. While the former implies a difference of one, as described above, the latter implies a FIXED DIFFERENCE, not necessarily one.

Because of this similarity, tables which are, in fact, successive are frequently categorized with consecutive tables. Successive tables are tables whose key items vary from entry to entry by a fixed value.

EXAMPLE 6

Entry Number	XX	XY
Ø	Ø.Ø	Ø.Ø
1	Ø.5	Ø.321
2	1.Ø	1.285
3	1.5	2.89
.	.	.
.	.	.
.	.	.
1Ø	5.Ø	32.13

This table of coordinates on a parabola of focal length 28 inches, where x is given in feet and y in inches, is successive. The key item is XX, which ascends with a fixed value of .5; it can easily be seen that if the "unit" were considered to be 1/2, then the difference between two adjacent values of the key is one "unit".

ALGORITHM

The table lends itself to the same sort of immediate referencing as does a consecutive table, except that an "ALGORITHM" or rule for computation, must be used to determine the entry number, given the value for x. Clearly, in this case, entry number = 2 * XX or K = 2 * XX where K is the entry number. It would also be possible to leave out the item XX in this table, because of this relationship.

EXAMPLE 7

Entry Number	MAX'CAP	FIXD	VAR
Ø	5	6ØØØ	3ØØØ
1	1Ø	12ØØØ	22ØØ
2	15	18ØØØ	15ØØ
.	.	.	.
.	.	.	.
.	.	.	.
4	25	3ØØØØ	6ØØ

In this table, taken from those of a management game, the key item MAX'CAP, increases by a fixed value of 5. The table is therefore successive, and the item MAX'CAP may be removed from the table if an algorithm can be devised whereby the entry number can be computer given a value for MAX'CAP. In this example, the computation is as follows: I = (MAX'CAP - 5) /5.

SUMMARY

Four major plans for the organization of information in tables have been discussed, ascending sequence, descending sequence, unknown sequence, and consecutive (including successive) sequence. Each has its particular advantages for specific processing needs, and no hard and fast rule applies to the choice of method.

CHAPTER 5

BASIC METHODS OF ACCESSING INFORMATION

INTRODUCTION

As the programmer plans the organization of the information with which his program will deal, he does so with the thought in mind, that the items in the table he is creating must eventually be accessed and the data within the table manipulated. With a given set of data, then, he must attempt to determine the method of structuring the table so that each item can be easily accessed.

The considerations influencing the programmer's decision include: entry structure; number of items per register, and number of registers per entry; table structure: parallel or serial; table length: fixed or variable; the order of the information in the table: ordered or unordered; and the possible order of new information input to the table during processing.

It can be said, then, that the organization of a table is based primarily on the nature of the information in it, and the methods used to access this information.

ITEM IDENTIFICATION

The process of locating or retrieving a specific item of information from a previously organized table is called information retrieval. In order to determine the location of a specific item, there must be a means of identifying the item.

Undoubtedly, the most obvious means for identifying a piece of information is by its name. In fact, when a non-tabular item is accessed, its name designates the complete identification. The name ALPHA then, given to a non-tabular item, represents the location of ALPHA. This relationship, of course, can only be true when the names given to non-tabular items are unique.

On the other hand, when the item to be used describes a series of objects, and thus occurs in a table, its name is not sufficient to identify it. The name of this item must be modified in some way to distinguish one item from the others like it. For example, if the name BETA were contained in a 5Ø entry table, reference to the name BETA could mean any one of 5Ø possible occurrences of BETA.

SUBSCRIPTS

It can be seen, upon examination of tables, that while the name of an item remains constant throughout the table, the entry numbers vary from entry to entry. It seems reasonable, then, to use this variable factor, the entry number, as a means for distinguishing one item from another in the same table. The combination of item name AND entry number allows the programmer to determine the specific item of many possible items with which he is concerned, and constitutes the complete identification of that item.

Since entries in a table containing "n" entries are numbered from Ø to n-1, the first entry in a list containing the item BETA could be referenced to as BETA (Ø), the fifth as BETA (4), and the last as BETA (n-1). The parentheses are simply a convenient means of representing the relative position of the entry in the table. The numbers within the parentheses are commonly called indices or subscripts.

In order to process all of the entries of a specific table, the programmer must designate both the item name and the number of the entry from which that item must be taken for each of the entries in the table.

EXAMPLE 1

Given:

TABLE	ABC	5 1 5	R	R	P
ITEM	BETA	ABCØ1	1	8	U
ITEM	SUM	1 48 U			
ITEM	AA	1 48 U			

Required: Find the sum of all the values of BETA and store this sum in the non-tabular item SUM.

Solution:

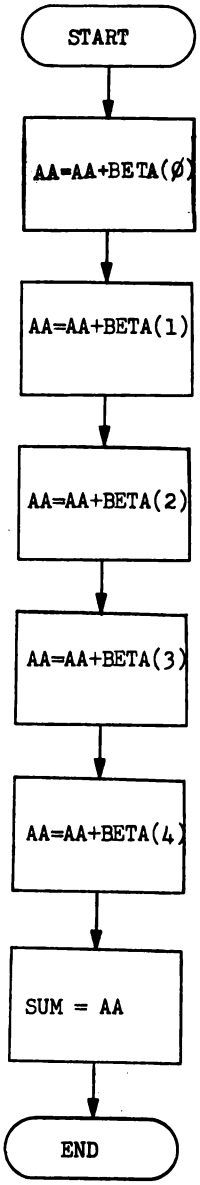


Figure 5-1.

The solution in Example 1 requires independent reference to each occurrence of the item BETA. AA is a CPU register used to contain the intermediate results as the operation is carried out. The solution to the problem can be represented graphically in the following way as well.

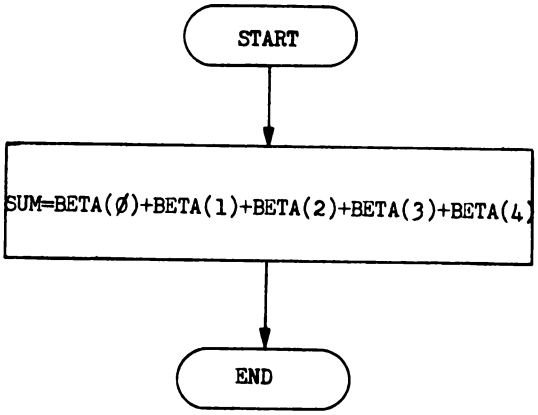


Figure 5-2.

Although this flow chart is far easier for the writer, the first solution actually represents the machine operations in the solution of the given problem.

EXAMPLE 2

Given:	TABLE	BBB	5 1 5	R	R	P
	ITEM	BETA	BBBØ1	1	1Ø	U
	ITEM	TALLY	1 3 U			

Required: Count in the non-tabular item TALLY the number of values of BETA which are equal to zero.

Solution: See Figure 5-3, next page.

The solution to this problem is not as easily simplified as is the solution to example 1. Notice, however, that beyond the first question, the flow chart simply repeats itself, using different subscript values for the item BETA.

In both of the preceding examples, it was necessary to access successive values of the item BETA, for all the occurrences of BETA in the table. In order to represent this, or indeed to accomplish it on the computer, it was necessary to repeat the same or similar operations several time, changing only the subscript for an item. This is a tedious method indeed, and seems an unreasonable one as well.

VARIABLE SUBSCRIPTS

The problem of repetitive accesses to items in a table can be easily solved by subscripting the item name with a non-tabular item (index word) which will be initialized, updated and tested under program control. Using this principle, the programmer can make use of a "loop" in which the named item need be referred to only once for any one operation. The general form for identifying a tabular item using this scheme will be NAME(ITEM), where NAME represents

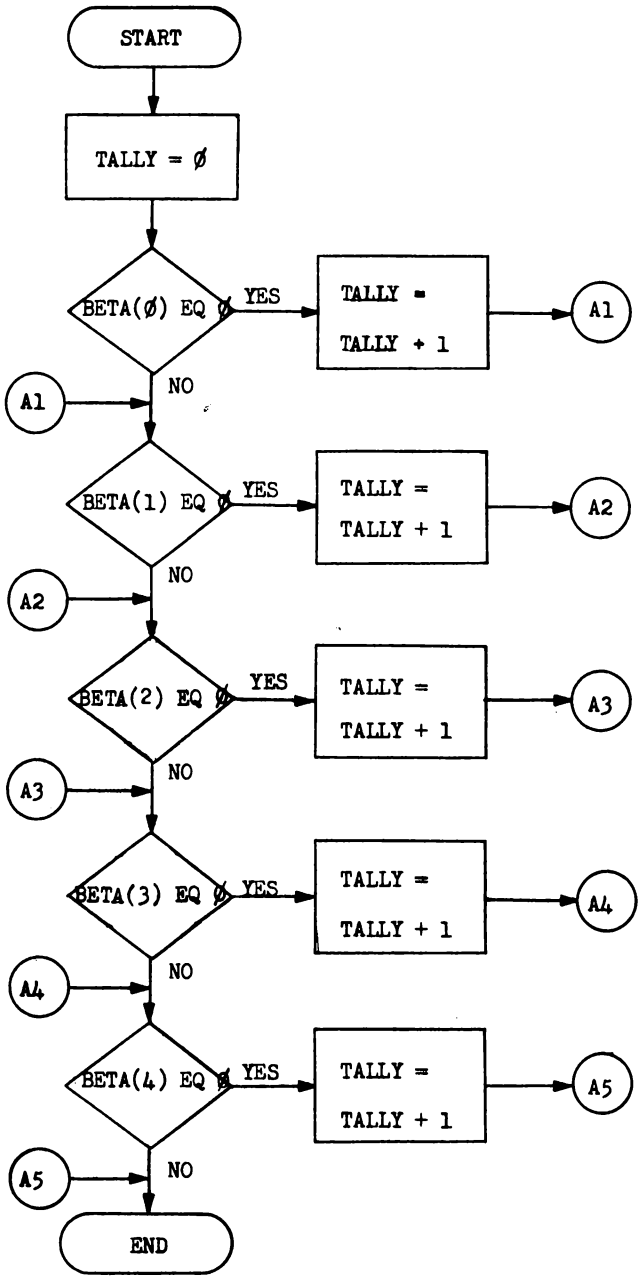


Figure 5-3.

the name of the item being accessed, and ITEM represents a non-tabular ITEM, the index word, whose value is being used as a subscript.

As an example of the technique mentioned above, one might look at the expression BETA (INDEX). This expression refers to the item BETA in the entry whose number is contained in the non-tabular item INDEX. If the table containing the item BETA were 5Ø entries long, and all entries had to be accessed, the programmer would set up the item INDEX so that it would contain values from Ø to 49 inclusive. It is a simple matter to initialize the value in INDEX at Ø, update it at the proper time by the desired amount (one, in this case), and test it for a final value greater than 49. Similarly, one could initialize INDEX at 49, decrease it by one at the proper time, and test it for a final value less than zero.

EXAMPLE 3

Given: A 5-entry table containing the item BETA.

Required: Find the sum of all the values of BETA, and store this sum in the non-tabular item SUM.

Solution:

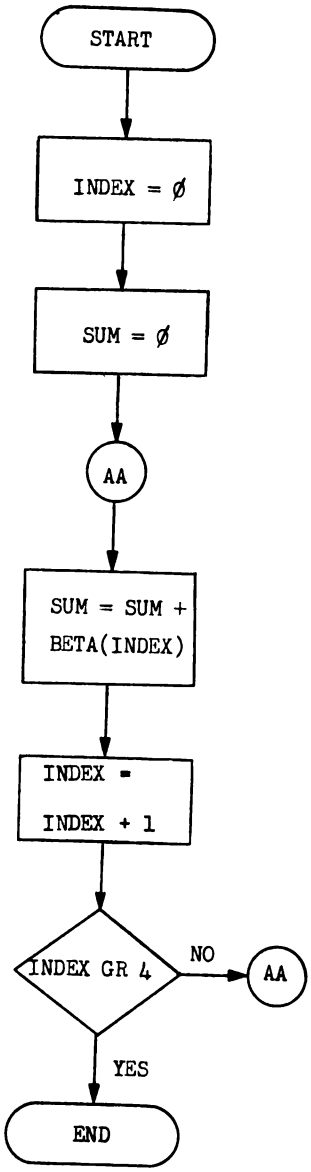


Figure 5-4.

This problem, exactly the same as the problem in Example 1, has been solved by using the technique described above, namely, the variable subscript. Notice that, after the entry to the program, the first two boxes are "initializing" steps. The first of them, set INDEX = ∅, is initializing the variable subscript. The second is initializing the non-tabular item SUM. This step is necessary not because of the demands of the problem, but because of the method used to solve it.

The section of the problem which begins at AA is the "body" of the loop. As such, it is concerned with actually performing the desired operation, in this case, the forming of a sum.

The two boxes preceding the END in this program account for the updating, or "modification" of the loop, and the testing of it, respectively. Notice that in this case the variable subscript was first modified, then tested. It is also possible to test the subscript first and then modify it.

EXAMPLE 4

Given: A 5-entry table containing the item BETA.

Required: Count in the non-tabular item TALLY the number of values of BETA which are equal to zero.

The solution for this problem makes use of a decrementing loop, that is, one initialized at the maximum value and decreasing to the minimum value of the variable subscript. Further, the abbreviation I is used for the subscript. The use of single letters for variables whose primary functions are as subscripts is common practice in programming.

Solution:

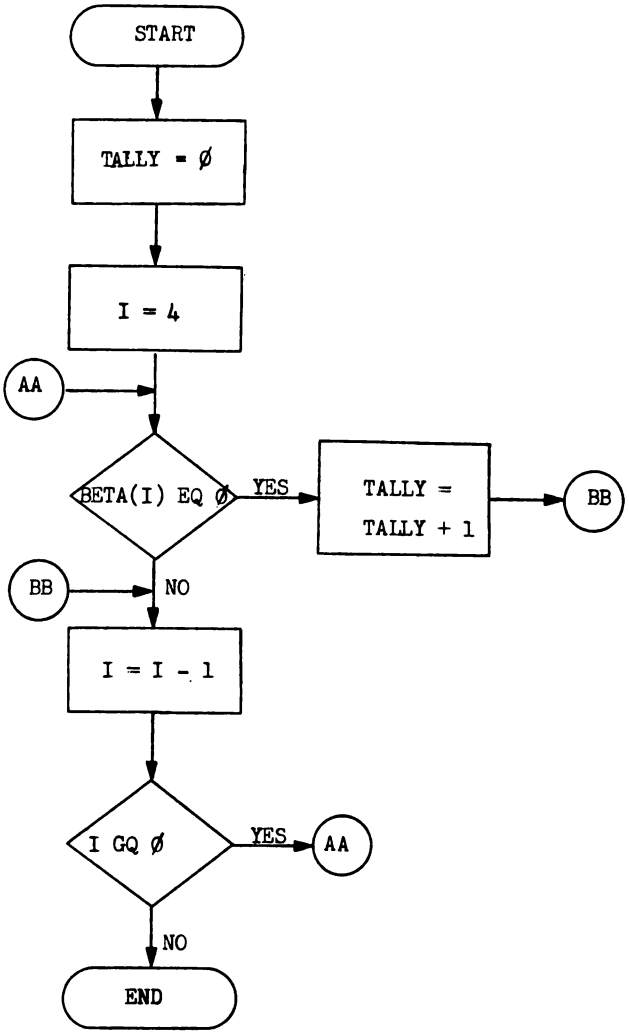


Figure 5-5.

FORMULAS AS SUBSCRIPTS (ALGORITHMS)

It is quite possible that items in a table have been organized such that the subscripts which will complete the identification of an item, and thus establish its location, will be generated as a result of some computation. A rule that is used to compute an entry address is known as an algorithm. In these cases, it is not necessary to assign the computed value to some non-tabular item which then could be used as a subscript. Rather, it is possible to use the algorithm itself as the subscript.

EXAMPLE 5

Given: A 360-entry table containing the items SINE, COSINE, and TANGENT, where the entry number is equal to the angle whose trigonometric functions are represented in it. The non-tabular items ALPHA, BETA, and GAMMA contain values for angles in degrees.

Required: Set item COS, a non-tabular item, to the cosine of an angle computed as follows:

ALPHA + 2 BETA - 3 (GAMMA/4)

Solution:

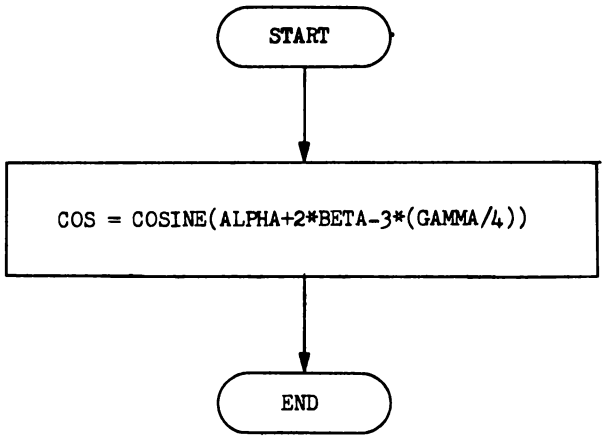


Figure 5-6.

Notice that the solution uses the entire algorithm as a subscript, eliminating the need to make a separate computation of the angle in question, store it in an item, and use that item as the subscript for the item COSINE.

The discussion of subscripts has thus far taken the reader through the use of constants as subscripts, as well as the use of variables and algorithms as subscripts. It becomes apparent that ANY single valued function may be used as a subscript. The only real restriction placed upon a subscript is that it must be an integer. This is reasonable since subscripts are in fact entry numbers, and entry numbers are integers.

TABLE SEARCH

In many programming problems, not all entries of a table need be processed, but rather only selected ones. If the selection is based on the identification of the entry, no difficulty exists which cannot be solved using the methods described above. If, however, the basis for selection of the entries is NOT their locations, but rather one or several of the values stored within them, then finding a specific record becomes a matter of searching for it. The search consists not only of the determination of location, but also of a value comparison. When an item compares favorably with the given criterion, the search is terminated.

SLOT-BY-SLOT-SEARCH

Conceptually, the simplest way to conduct a table search is to proceed through the table entry comparing the key item with the given value and accepting or rejecting it on that basis. Because of the nature of the information contained therein, some tables must be processed in this way; however, this technique is one of the more inefficient ones for this purpose.

The efficience of a searching technique is generally measured in terms of the number of memory accesses and comparisons which are necessary to locate an arbitrary entry. Obviously, this method will, for an "n"-entry table, require an average of n/2 accesses and comparisons to locate a given entry. It is possible to improve this average slightly by ordering the information in the table so that the most popular entries are accessed first, but this still does not provide what would be judged to be a highly efficient search.

EXAMPLE 6

Given: A 49-entry table containing the item XX.

Required: Find the largest value of XX and store it in the non-tabular item LARGE.

Solution:

In the solution to this example, notice the initializing steps which occur between the entry to the program and the AA connector. The body of the loop appears at AA and the following, where each entry is inspected and compared with the previous value of LARGE. The area of the program beginning at BB is the modification and test portion of the program, where subscript I is first tested, then, if it is within the proper range, it is increased and another iteration of the loop is performed. The technique used in this solution is the slot-by-slot search.

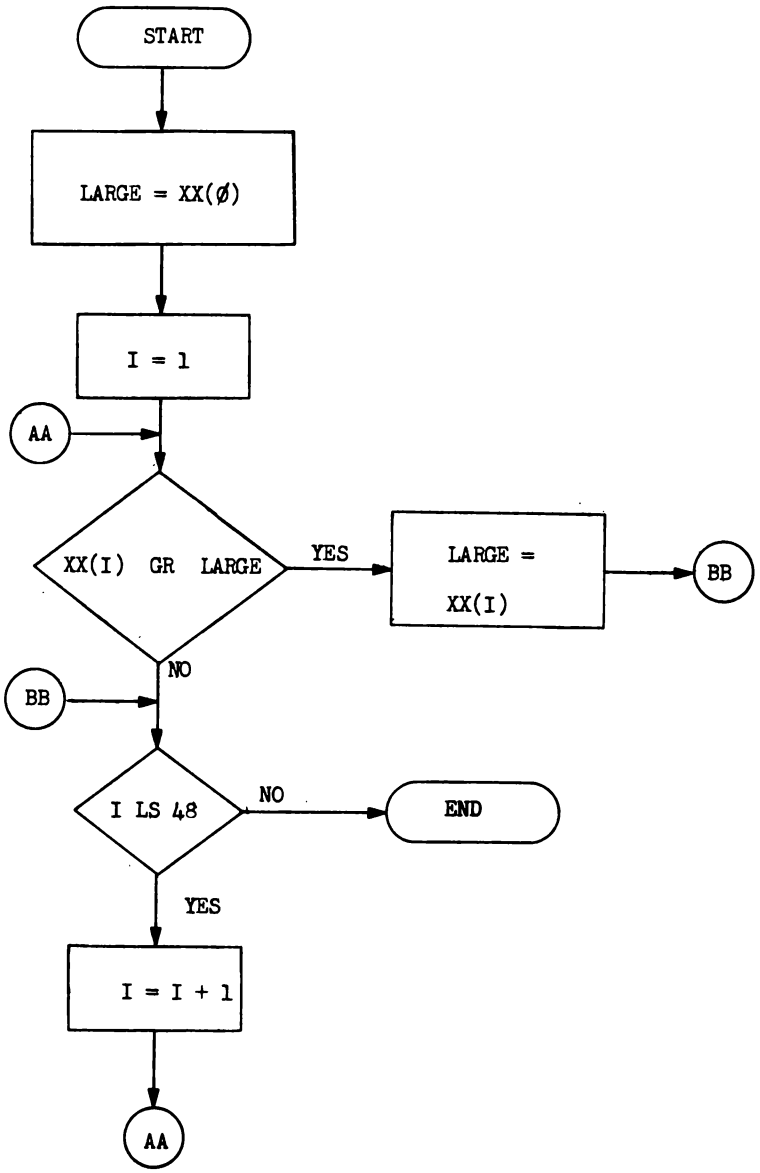


Figure 5-7.

EXAMPLE 7

Given:

TABLE	ABC	1ØØ	1	1ØØ	R	V	P
ITEM	ALPHA	ABCØ1			1	1Ø	U
TABLE	DEF	1ØØ	1	1ØØ	R	V	P
ITEM	BETA	DEFØ1			1	1Ø	U

I CONTROLS ABC
J CONTROLS DEF

Required: In the table DEF following the last meaningful entry of BETA store all the values of ALPHA which are greater than zero.

Solution:

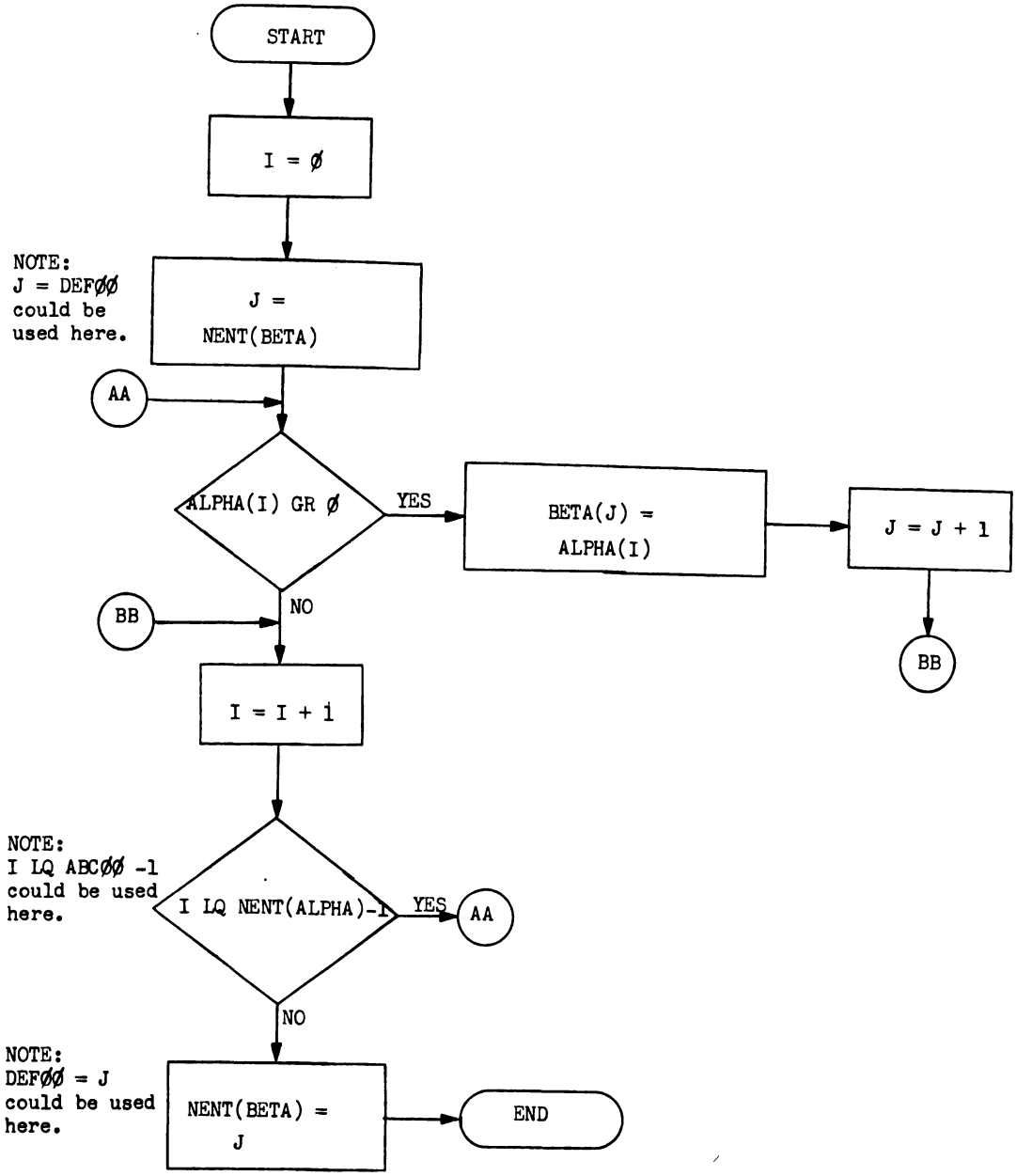


Figure 5-8.

Notice that the solution to the problem, first of all, involves a slot-by-slot search of the table containing ALPHA for those values of ALPHA which are greater than zero. The first two steps are, once again, initializing ones, the first setting up the subscript I for the processing of ALPHA, the second setting up J to store into BETA. Note that J begins at NENT(BETA) since the last significant entry already in BETA is stored at BETA(NENT(BETA)-1).

The area of AA and continuing to CC is the body of the loop, where ALPHA is inspected and stored, if storage is required. Notice that J is increased only when a new BETA has been stored.

The area beginning at BB is the increment and test portion of the program, where I is increased and tested against NENT(ALPHA) since the table containing ALPHA is also variable length. If the subscript is within the required range, NENT(BETA), which should be altered since new values have been added to the table containing BETA, is reset to J, and the program stops. The student should be convinced that at this point in the program execution J does contain the correct value for the setting of NENT(BETA).

BINARY SEARCH

If the information in a table can be ordered into some sequence based on the values of one of the items in the entries, the table then can be handled with more sophisticated methods than the slot-by-slot search.

Suppose, for example, that a table containing the items AA and BB can be arranged so that the values contained in the item AA are in ascending sequence; so that the smaller values for AA are in the lower (Ø, 1, 2, etc.) entry numbers of the table and the larger values for AA are in the higher (49, 50, 51, etc.) entry numbers. In this case, if the first value to be examined is located in the middle entry of the table, and it is not the desired value, then the desired entry will lie in either the upper half of the table or in the lower half (if the desired entry exists at all). Since the values of AA are ordered, it can always be determined in which half of the table the entry sought is located.

This procedure halves the size of the table which remains to be searched, hence the term "binary search". To continue the process, the new table is split again and the middle term is inspected; this procedure is repeated until either the desired entry is found or until the search if narrowed down to a single entry (or address).

Although the concept of a binary search is not a particularly difficult one, some of the details of its programming can become rather trying. For instance, how does one compute the middle entry number? What if the computation of an entry number produces a fraction? Will the value be rounded or truncated? How many subscript-like items are required? How will they be used?

Perhaps the best method to resolve some of these problems is to consider an example, and determine whether the proposed solution will work under the given circumstances.

EXAMPLE 8

Consider a table containing values for the items XX and YY, where XX is the key item. Let us say that the table is variable length, and that the current

value for NENT(XX) is 8. Let us consider the case where the value contains the following values:

	XX	YY
Ø	1	50
1	7	17
2	13	22
3	18	6
4	22	125
5	25	9
6	27	4
7	28	2

Figure 5-9.

Using the binary search techniques, we will look for the entry containing XX EQ 25. We will perform the following steps in conducting the search:

1. Compute the middle entry number. We can compute the middle entry with this formula:

middle entry = (Top Entry # + Bottom Entry #)/2

or

(Ø + 8)/2 = 4

2. Inspect the middle entry. XX(4) NQ 25; it is, in fact, less than 25, from which we can deduce that the desired entry lies in the lower half of the table. And entry (4) now becomes the top entry of the new table.

3. Compute the new middle entry number. $\frac{(4 + 8)}{2} = 6$

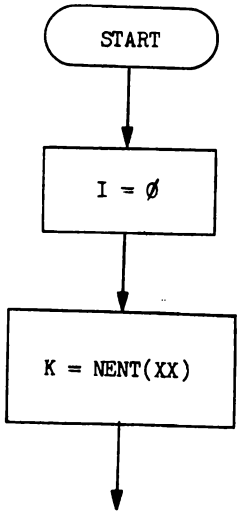
4. Inspect the middle entry. XX(6) NQ 25; it is greater than 25. Thus, the desired entry lies in the upper half of the table. The bottom entry number is now 6.

5. Compute the new middle entry number. Using the principle developed in 1 above, we make this computation by dividing the ending entry number of the new table plus the beginning entry number of the new table by two. Thus, (4 + 6)/2 = 5.

6. Inspect the middle entry. XX(5) EQ25. The correct entry has been found; the search is terminated.

Reviewing this method, let us attempt to construct a flow chart of the process. Step 1, computing the initial middle entry number appears to stand alone, that is, to be an unrepeated one. Looking, however, at the other steps which involve the computation of a middle entry number, can we arrive at a general form which will apply to all of these situations, so that it can be reused for the computations? It is possible to consider, in each case, that a sub-table is being used which has both a beginning and ending entry number. The only respect in which the first case differs from this pattern is that it is not really a sub-table. It does, however, have beginning and ending entry numbers, namely, zero and NENT(XX). Let us, then, set up non-tabular items to contain the beginning and ending entry numbers of the sub-table with which we are presently concerned, and initially set these items to the values zero and NENT(XX).

We begin, then, with:

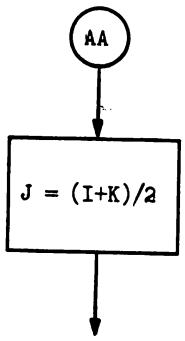


To be used for beginning entry number.

To be used for ending entry number.

Figure 5-10.

Continuing then, to the computational use of these Index registers, we can construct a general computation of middle entry number, through which the program will loop:



To be used for middle entry number.

Figure 5-11.

Steps 2, 4, and 6, above, all involve the inspection of the middle entry. They illustrate the three possible exits from a comparison of two values; they are equal, the first is larger, or the first is smaller. The following illustrates this comparison.

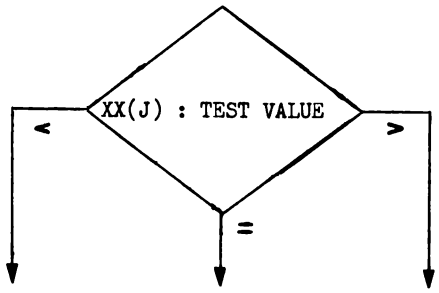


Figure 5-12.

If the result of the comparison is equality, as in step 6, the search is terminated; thus, we can add the following.

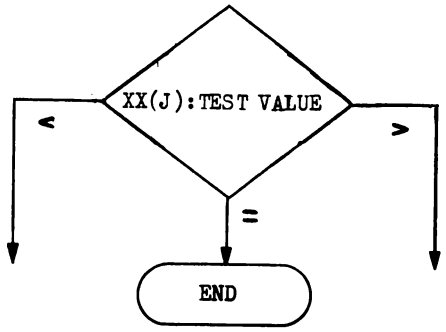


Figure 5-13.

If the result of the comparison is the "less than" case, as in step 3, a new middle entry number must be computed. Since we intended to reuse the computation at AA, we need only revise the value for either I or K at this point. In this case, the new table is the lower half of the present table, and it is therefore the beginning entry number, I, which must be changed. We can add the following flow chart.

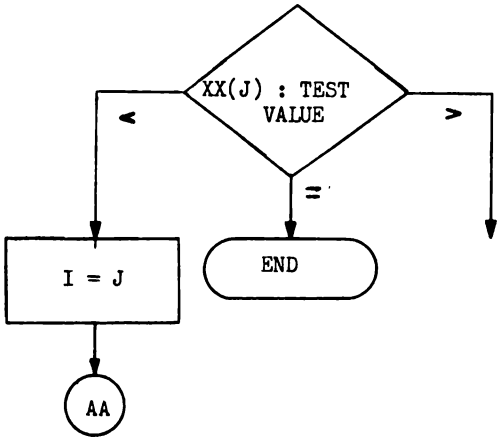


Figure 5-14.

If the result of the comparison is the "greater than" case, we are at step 5 of our solution, and need only to reset the ending entry number in order to proceed. Thus we have:

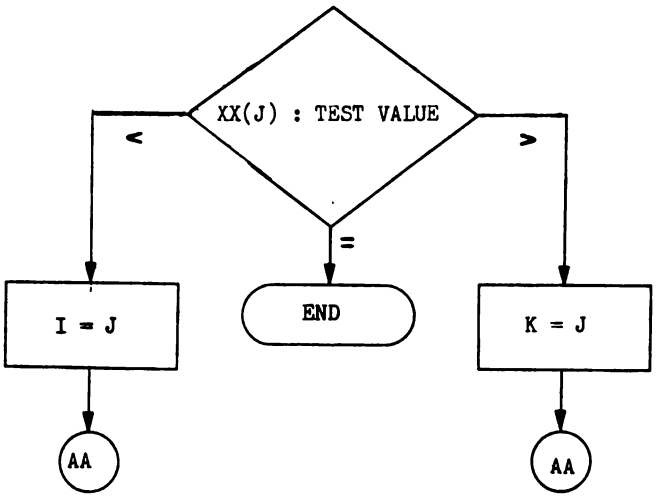


Figure 5-15.

Putting the entire flow chart together, we have:

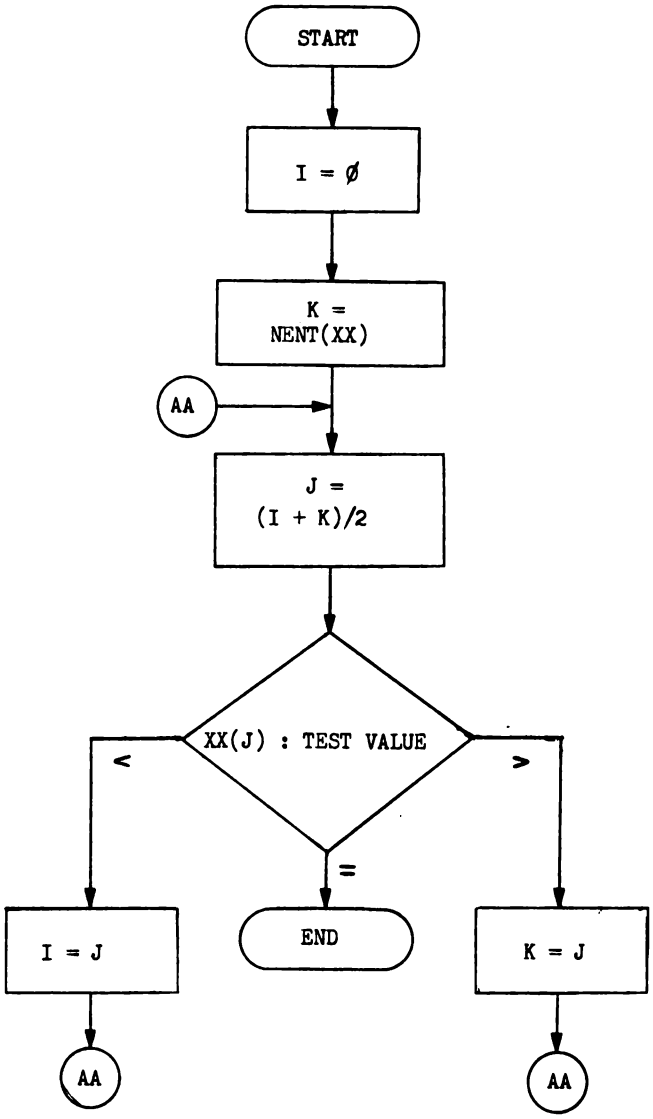


Figure 5-16.

Using this flow chart, let us now go back to the sample table, trying to find XX EQ 1. Going through the flow chart, we perform the following operations:

1. Set I to zero.
2. Set K to 8.
3. Set J to (Ø + 8)/2, or 4.
4. Compare XX(4) to 1. XX(4) is greater than 1.
5. Set K to J, or 4.
6. Set J to (Ø + 4)/2, or 2.
7. Compare XX(2) to 1. XX(2) is greater than 1.
8. Set K to J, or 2.
9. Set J to (Ø + 2)/2, or 1.
10. Compare XX(1) to 1. XX(1) is greater than 1.
11. Set K to J, or 1.
12. Set J to (Ø+1)/2. But at this point, the result is fractional. We must decide, then, whether fractions will be rounded or truncated. It is obvious, because of this particular situation, that rounding is not a good choice. Rounding, it is clear, would never allow access to entry zero. Let us, then, specify at this point that truncation of fractions will occur at step AA of the flow chart.
13. Compare XX(Ø) to 1. They are equal.
14. Stop.

Since we have, by considering this case, introduced a new restriction on the flow chart of the binary search, we ought to consider another of the "end cases" for this flow. Let us use the binary search on the sample table again, this time searching for the value XX EQ 28.

1. Set I to zero.
2. Set K to 8.
3. Set J to (Ø + 8)/2, or 4.
4. Compare XX(4) to 28. XX(4) is less than 28.
5. Set I to J, or 4.
6. Set J to (4 + 8)/2, or 6.
7. Compare XX(6) to 28. XX(6) is less than 28.

8. Set I to J, or 6.

9. Set J to (6 + 8)2, or 7.

10. Compare XX(7) to 28. They are equal.

11. Stop.

The technique of truncation does not seem to prevent access to the last entry in the table. Let us, before we abandon the sample table, consider one last problem. What will happen, using this flow chart, if we search for a value of XX which does not exist in the table, for example, XX EQ 4? Clearly, since, if the value did exist in the table, it would lie between the 1 and the 7, the search will resemble the search for XX EQ 1. In fact, it will be identical to that search until step 13, where equality will not be found, but rather, XX(Ø) will be less than 4. Returning to the flow chart, we would insert, then, as step 14) the setting of I to Ø; this would be followed by step 15) where we would set J to (Ø + 1)/2, truncation indicated, producing zero again. In other words, we will find ourselves in a never-ending inspection of entry zero of this table, which is certainly not a desireable situation. It appears that this problem must be resolved somehow, since we have shown a clear deficiency in the flow chart.

To restate the problem: the flow chart does not provide an "error exit" for the case where the desired value does not exist in the table. In order to resolve the problem, the programmer must determine some means of detecting this situation when it has occured. Some possibilities are: 1. When the same entry has been accessed, or is about to be accessed, more than once in sequence, the desired entry does not exist (the latter can be determined without the addition of a counting function; or 2. when I, the top entry number becomes greater than K, the bottom entry number. This condition indicates that the top of the table and the bottom of the table have by-passed each other.

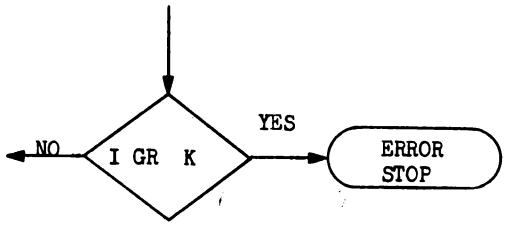


Figure 5-17.

The difficulty arises at the time of the setting of I and/or K. Moreover, the problem arises precisely when the value to which either I or K is being set is the same as the value it presently contains. It should be, and is, rather simple to eliminate this problem by asking a question before each of the settings in question. The question to be asked is posed above, actually, and is "IS K EQ J? or IS I EQ J?" the yes response to either question indicates an error condition.

The modified flow chart, then, has the following form:

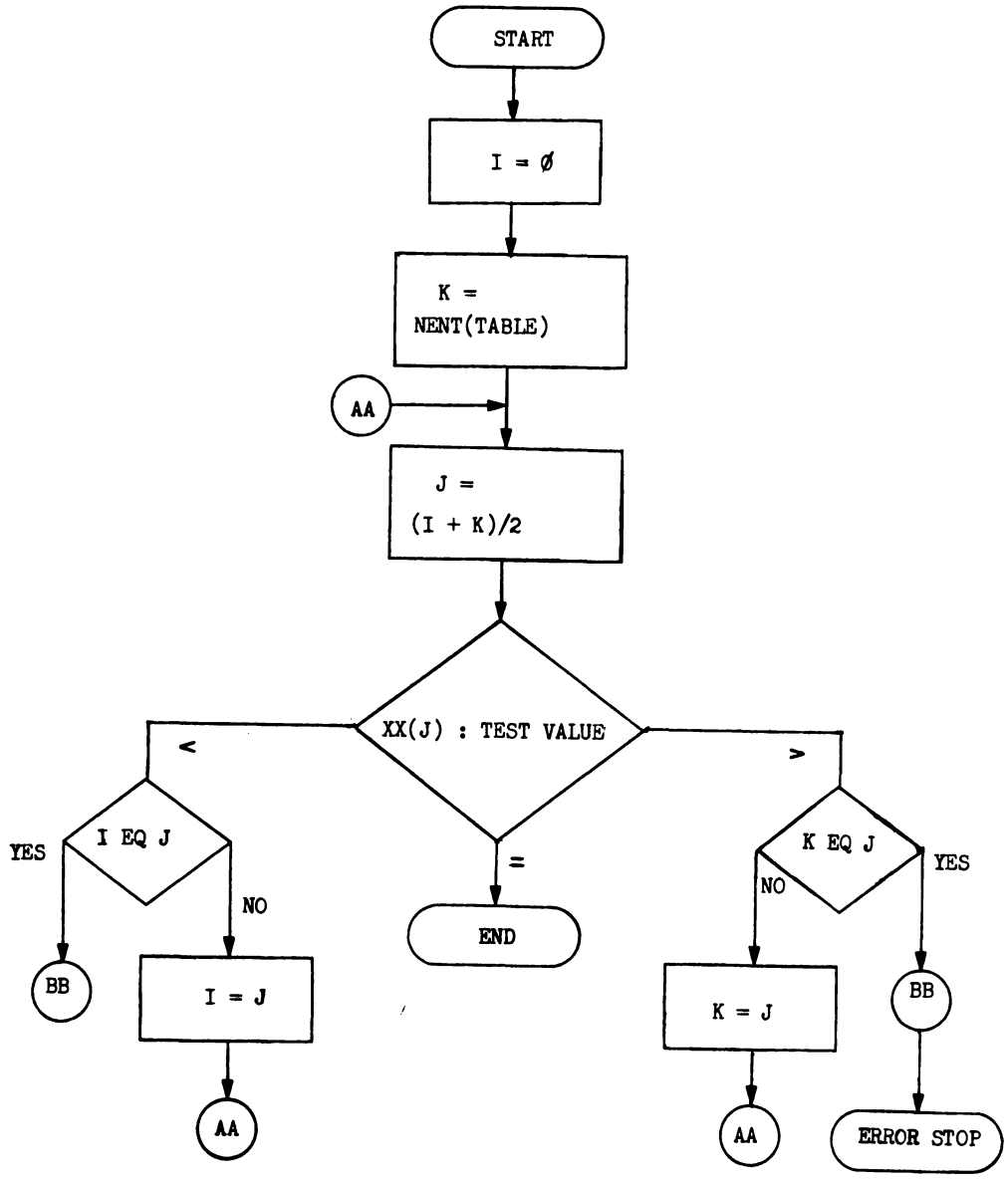


Figure 5-18.

It only remains to insure that the flow chart described above will work equally well on tables containing an odd number of entries, since all tests have been made on a table containing an even number of entries. This test will be left to the reader.

The following flow chart is another example of a binary search routine.

T -	Top entry number
M -	Middle entry number
B -	Bottom entry number
TV -	Test value

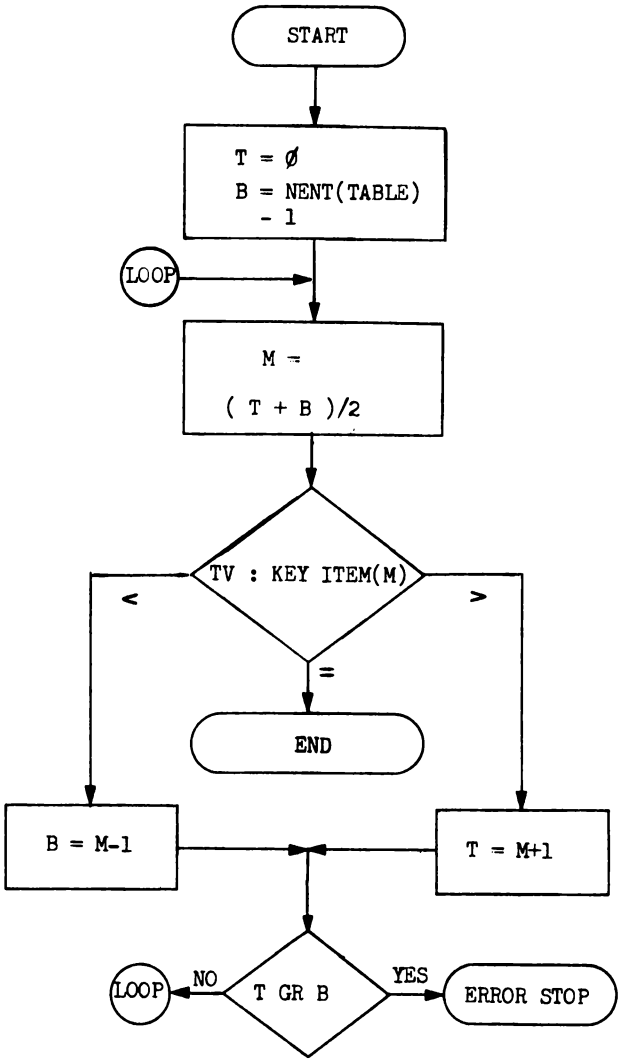


Figure 5-19.

Clearly, the most efficient division point in a binary search is not necessarily the middle of the table. Evidence of this fact exists in everyday occurrences . . . searching a table of information arranged alphabetically, the human does not start at the middle of the list if he is looking for something beginning with A. The best division point is that one at which it is equally probable that either part of the table thus divided will contain the desired value. If this point can be estimated before comparison with the table, using some rather simple computation, the binary search can be improved in its average number of accesses. It is obvious, however, that if the estimation of the division point is highly complicated, the estimation will itself outweight the advantage in time gained by this technique.

DIRECT LOOK-UP

At various other phases of the discussion of programming techniques, it has been mentioned that there is the possibility of organizing information so that there is a concrete relationship between the value in an entry and the entry number. This type of organization is used for the purposes of direct look-up. A typical case of this form of organization can be found in Example 5 of this chapter. In this case, the entry number is equal to the angle in degrees for which

the trigonometric functions are stored. Since the item ANGLE, if it did appear in the table would always equal the entry number, it is omitted from the table, thus conserving space.

The relationship between the entry number of an item and its identification (where the identification is considered to be the value of the key item, whether that item actually appears or not) need not be equality in order for the direct look-up scheme to be used. In fact, any arithmetic relationship (algorithm) will suffice for this purpose. The only requirement being that the identity of a key item be given so that the entry number of that item can be computed.

EXAMPLE 9

```
TABLE    MESSAGE    12Ø  1  12Ø    R     R      P
  ITEM   CHAR       MESSAGEØ1      1     6      H
ITEM     KIND       1  1  V       V(PCODE)     V(TCODE)
```

Given: Table MESSAGE, 12Ø entries long, contains the item CHAR. CHAR is a 6-bit item which will contain legal character codes. The table will be loaded with either typewriter codes, or with line printer codes. An item KIND, non-tabular, will be set to 1 if MESSAGE contains typewriter codes, and Ø if MESSAGE contains printer codes.

Required: If MESSAGE contains typewriter codes, replace each character with the corresponding line printer code. If MESSAGE contains line printer codes, replace each character with the corresponding typewriter code.

Solution: 1. Organization of Information

It can easily be seen that the solution to this problem will require a table or tables which will allow the program to find the corresponding typewriter code, given a line printer code, and vice versa. It is, then, necessary to organize these tables and to do so with an eye to the way in which they will be used.

It would be possible to construct a table containing both sets of codes, where each entry will contain the typewriter code for a given character, and the corresponding line printer code. If such a table is set up, it will have to be accessed by a slot-by-slot search in at least one of the two cases, since it is not possible to order the table on the basis of both these items. For the other case, it would be possible to access the table using a binary search, at least, or perhaps by direct look-up. The requirements of the problem, however, would be far better satisfied if it were possible to reference the table in the most efficient manner in both cases.

A solution to this consideration would be to construct two tables, one containing the typewriter codes in sequence, with their corresponding line printer codes, and the other containing the line printer codes in sequence, with their corresponding typewriter codes. These two tables might look like this:

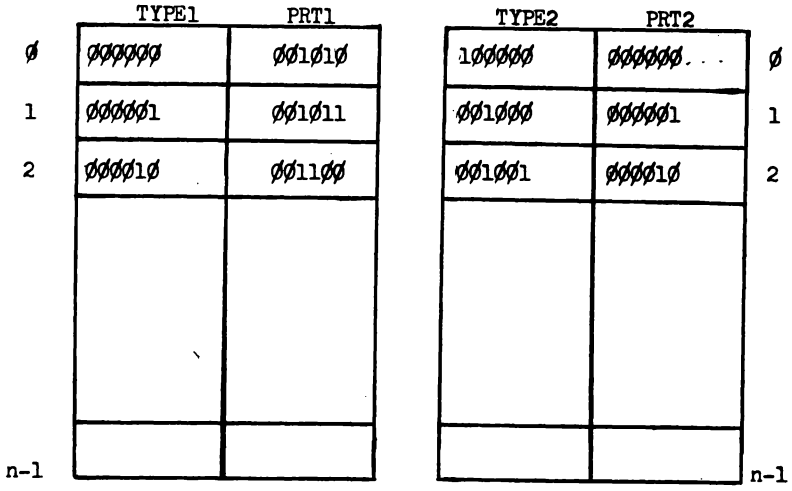


Figure 5-20.

The reader will observe that, in Figure 5-20, the table containing TYPE1 and PRT1, the binary values for TYPE1 are equal to the entry numbers. Similarly, in the table containing TYPE2 and PRT2, the values for PRT2 are equal to the entry numbers. The items TYPE1, and PRT2 would be removed from their respective tables with no loss in information content. Doing this will leave us with two tables, one containing the item PRT1, ordered so that in any entry its value is the printer code for the character for which the entry number is the typewriter code; the other containing the item TYPE2, ordered so that in any entry, n, the value of TYPE2 is the code for the character which is represented by the value n in printer code.

Assuming that the computer with which we are dealing has at least a 12-bit word, we can save memory space by combining these two tables into one table called COMBINE containing the items TYPE and PRT, ordered as given above, and packed. The table will look like the following:

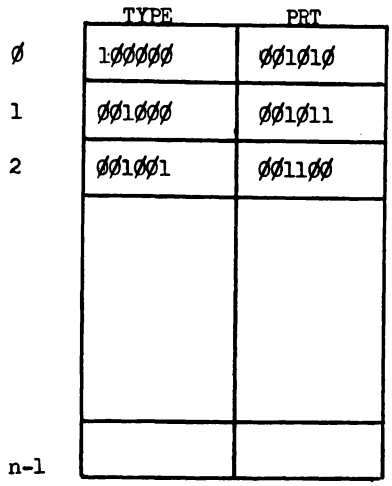


Figure 5-21.

2. Program Flow Chart

Because of the way in which table COMBINE discussed above has been organized, it should be clear to the reader that in order to access a typewriter code which corresponds to a given printer code, we need only address TYPE subscripted by the printer code. The reverse is, of course, also true. This is the technique of direct look-up . . . so called because we are able to go directly to the desired information with no search required. The flow chart which follows will use this technique for finding the codes, and will use a slot-by-slot processing of the table MESSAGE, since all entries must be processed.

Given:

```
TABLE    MESSAGE    12Ø  1  12Ø    R     R     P
  ITEM   CHART      MESSAGEØ1      1     6     H
TABLE    COMBINE    12Ø  1  12Ø    R     R     P
  ITEM   TYPE       COMBINEØ1      1     6     H
  ITEM   PRT        COMBINEØ1      1     6     H
ITEM     1  1  V    V(PCODE)       V(TCODE)
```

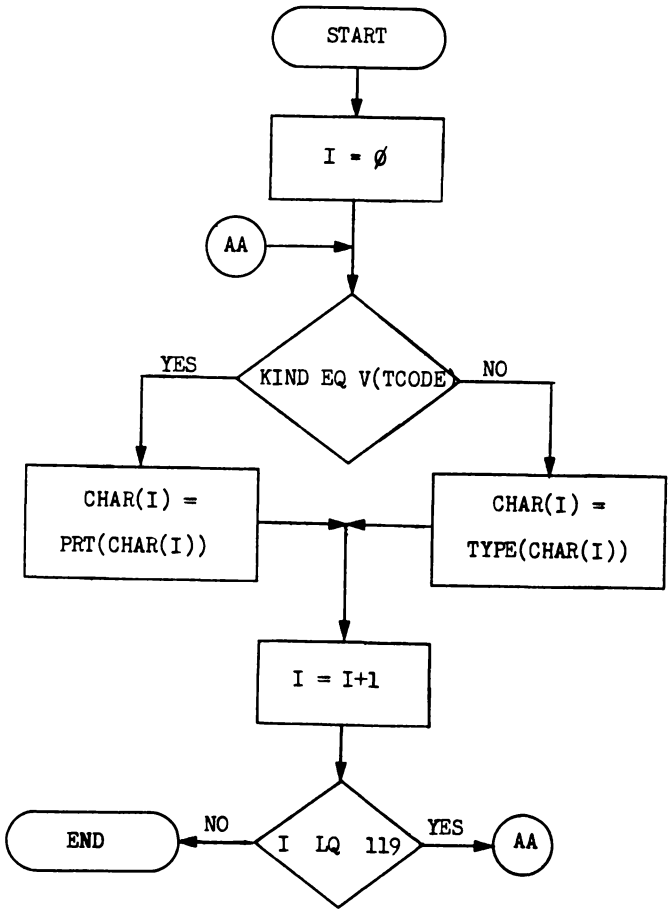


Figure 5-22

Notice that it is necessary to inspect the item KIND as each occurrence of CHAR is processed in order to provide that the correct code will be picked out.

EXAMPLE 10

	FIXED	VARY	
Ø	6ØØØ	3ØØØ	MAX'CAP = 5
1	12ØØØ	22ØØ	MAX'CAP = 1Ø
2	18ØØØ	15ØØ	MAX'CAP = 15
3	24ØØØ	1ØØØ	MAX'CAP = 2Ø
4	3ØØØØ	6ØØ	MAX'CAP = 25

Figure 5-23. Data Base.

Given: A variable length table containing the items MAX'CAP, FIXED and VARY, where MAX'CAP has been set, the other two items are unset.

Required: For each entry in the table containing MAX'CAP, set FIXED to the fixed cost for that capacity, and set VARY to the variable cost for that capacity.

Solution:

```
TABLE   PROD       5 1 5      R   R   P
  ITEM  FIXED      PRODØ1     1   16  U
  ITEM  VARY       PRODØ1     17  13  U
  ITEM  MAX'CAP    PRODØ1     31  18  U
```

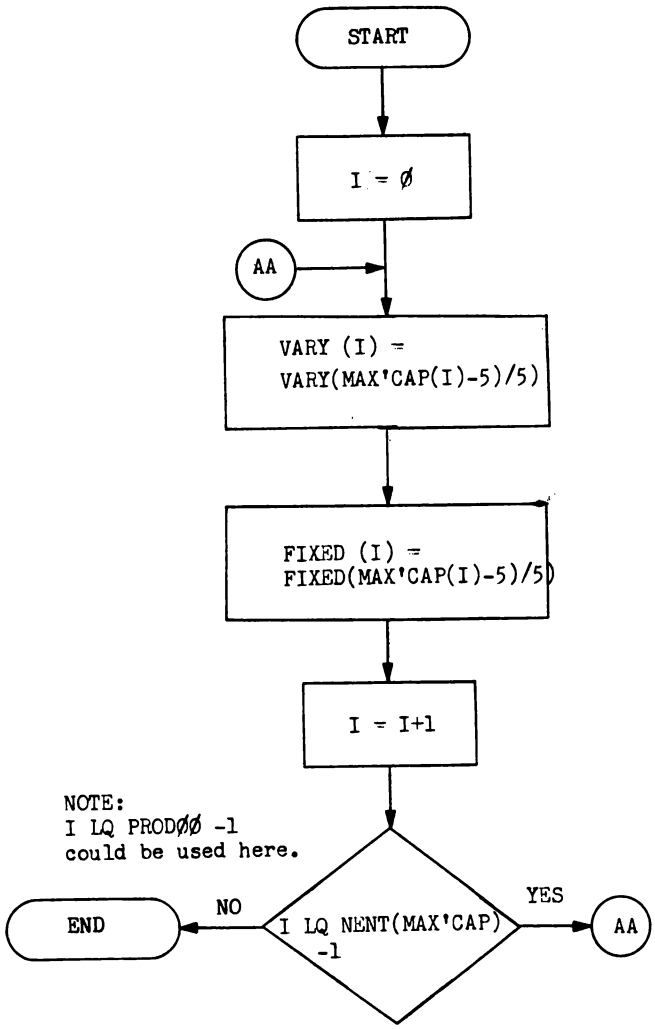


Figure 5-24.

Notice that the solution uses an algorithm for look-up into the given table. Notice, also, that slot-by-slot processing is used on the second given table.

The technique of direct look-up is a widely used one, and certainly presents great advantages where it can be used. The student should convince himself of the advantage, perhaps by flow charting a solution to one of the problems above using another technique. The organization of information for direct look-up is practical only so long as the relationship between the number of POSSIBLE values of the key and the number of the key and the number of USED values of they key is near one. As the number of unused values increases, the space wasted becomes prohibitive. Therefore as you can see, direct look-up can be used efficiently only with consecutive or successive tables.

From this point on, NENT will be referenced by using block ØØ of the particular table concerned. Instead of using NENT (table of item name), we will use TABLEØØ.

CHAPTER 6

DATA MANIPULATION

Computer programs are frequently involved in the processes of arranging or rearranging information stored in tables, or in files on external devices. These terms are used to indicate processes which place information in some specific order. The term rearranging refers to the process of transferring information from one state of order to another state of order, whereas arranging information implies transferring it from a state of disorder to a state of order.

It has been pointed out in other chapters that a table is designed in the light of two criteria:

1. The way in which the table is to be used.

2. The way in which the table is to be altered.

The particular considerations made in the determination of a suitable table structure are discussed in the chapter on information organization.

The student is no doubt aware that the ways of using a table are numerous, and that they may involve the processing of all the entries in the table or the processing of just a few of the entries. The choice of a search technique is, of course, dependent on these factors. The student is also aware of the possibility of arranging the information in a table into a known order, such as ascending or descending order, in an effort to increase the ease and efficiency of using the table.

Although using the table can have several different meanings, altering the table most often refers to the deletion of one or more entries from the table or the insertion of a new entry or entries into the table. Given that the table is organized in one of the basic structures, namely parallel or serial structure, the mere possibility of these operations implies that the table is variable in length. The student will recall that when a table has variable length, the space will be allocated to contain the maximum number of entries possible, and a control word (NENT) will be created to indicate the current number of entries in the table. Obviously, altering the table will require altering the control item.

SORTING

The arranging of information into an order is generally referred to as sorting. Although the entry may contain many items, the table is usually sorted on the basis of one, or perhaps two of these items. The item which is used to sort the table is called the key item.

The most common techniques of sorting lists can be categorized as either MERGING, DISTRIBUTIONAL or EXCHANGE techniques. Of these, merging and distributional are generally the faster methods, although they require the use of extensive additional storage space. While exchanging is generally slower than these methods, it has the decided advantage of requiring no additional storage space; in other words, exchange sorting takes place entirely within the table to be sorted.

Since sorting, along with many of the other techniques to be used in this chapter, requires that all of the items in one entry be moved to the corresponding items in another entry, it becomes convenient at this point to provide some means of indicating the movement of the whole entry on the flow chart. This is accomplished by using the name of the table and the

subscript attached to that name, which indicates the proper entries. Thus, SAMBO (I) indicates entry I in table SAMBO. As in the case of NENT, there is no confusion provided by the possibility of either an item name or the name of a table within the parentheses. This is true because names for items and tables must be unique in each problem. Using this form, entire entries can be moved into one figure on a flow chart, and the programmer is spared the effort of listing each of the items to be transferred.

EXCHANGE SORTING

The exchange sorting methods, while they are slower than most of the sorting methods, are carried out entirely within the tables on which they operate. The exchange is of course basic to the exchange sorting techniques, and can be indicated on a flow chart by the use of the double equal sign (==) to indicate the double setting which is accomplished by an exchange.

In general, the exchange sorting techniques operate by inspecting the key items from adjacent entries in the table; the inspection finds them to be either in order (depending on the order into which the table is being sorted) or out of order. If two adjacent keys are found to be out of order, the entries containing them are exchanged. When all adjacent pairs of key items have been examined, a PASS through the table is said to be complete. Various characteristics of these passes, and the number of them required distinguish the exchange sorting methods from each other.

One of the more simple and less efficient methods of exchange sorting involves no concepts other than the comparison of adjacent pairs of key items, exchanging as required until a pass is complte. At the completion of one pass, one of the values in question has been placed in its correct position in the list. The second pass will also place a single value in its correct position. Remaining passes through the table, like the first two, will each place one value in its correct position. Since this is the case, and the reader should convince himself that it is, the maximum number of passes that are required through the table can be determined. If each pass places one value in its correct position, then the next to the last value is correctly placed in pass n-1, for an n-entry table. When this value and all others before it have been sorted, the remaining value must also be sorted. Therefore, for an n-entry table, n-1 passes are required to guarantee that the table is in order. In many cases, depending on the original condition of the table, not all of these passes are needed; however, if the table is in the reverse order from the desired order initially, n-1 passes will be required. If, then, n-1 passes are always used, the table will be sorted regardless of its initial condition. The SIMPLE EXCHANGE SORT, then, uses a pass counter to determine when it has finished and halts when this counter has reached the number of entries in the table minus one.

EXAMPLE 1

Given:

TABLE	LTC	1ØØ	1	1ØØ	R	V	P	
ITEM	KEY	TLCØ1			1	6	S	Ø5.ØØ
ITEM	VALUE	TLCØ1			7	11	S	1Ø.ØØ
ITEM	PASS'NO	1	7	μ				

Required: Sort the table into ascending sequence on the basis of item KEY.

Solution:

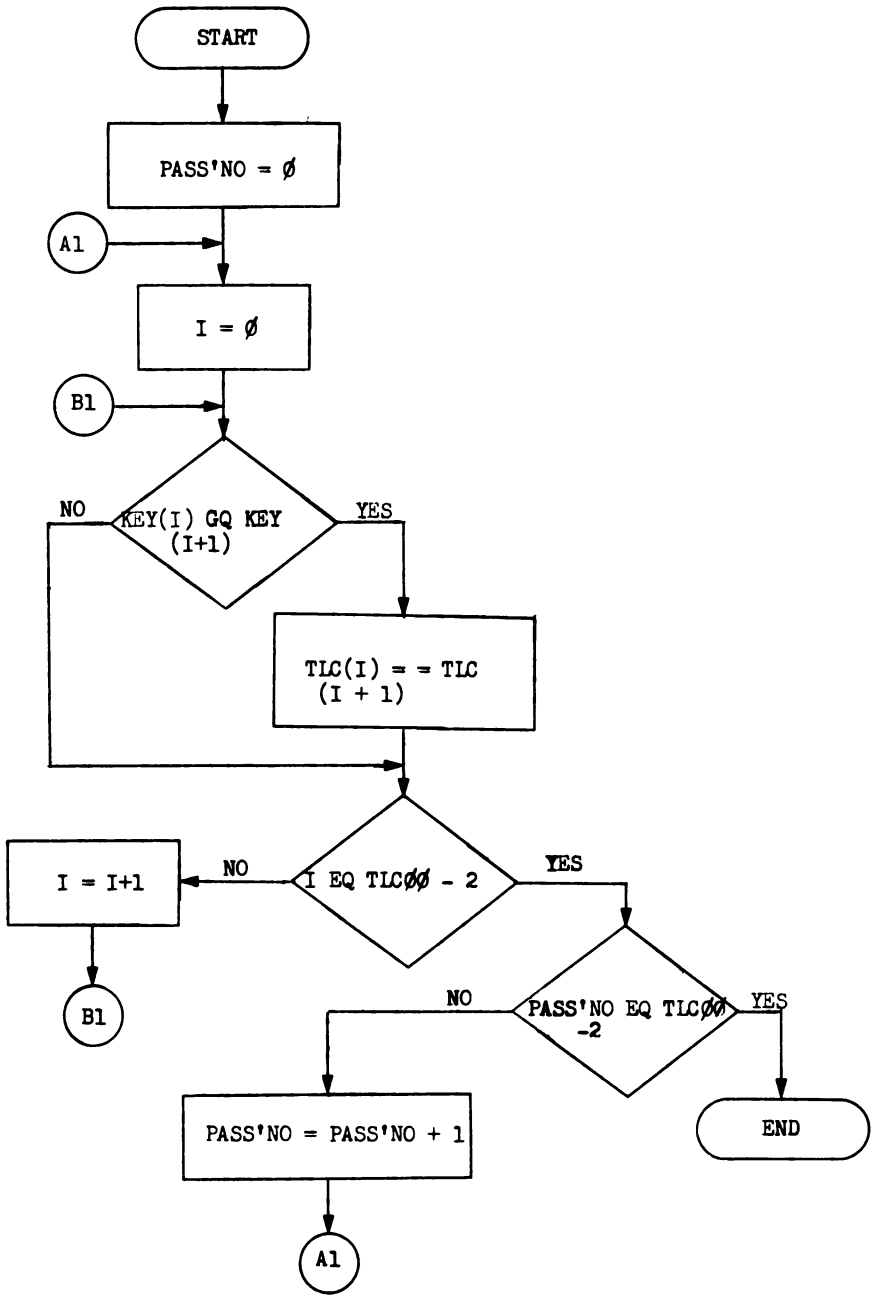


Figure 6-1

Example 1 illustrates the simple sort technique described on the previous page. The program stops on finding that PASS'NO, the pass counter, is equal to TLCØØ - 2. The passes, in the passes, in the flow chart, are numbered from zero to TLCØØ - 2, a total of TLCØØ - 1 passes. Notice that when equal keys are found, no exchanging is made. The program could certainly have been written to exchange in this case, but this would produce an obvious inefficiency.

VARIATION OF A SIMPLE EXCHANGE SORT

This sort routine is a slight variation of the simple exchange sort using a pass counter. It is more efficient in the fact that it orders the top of the table before proceeding on to the next entry. In other words, immediately after an exchange is made it starts the routine over again. Obviously this cuts down the number of passes especially when only a few items are out of order. Only one complete pass will be made which is only after the table is ordered. Upon the completion of the one pass the routine will end.

EXAMPLE 2

Solution:

Given:

TABLE	KIP	1ØØ	1	1ØØ	R	V	P
ITEM	KEY	KIPØ1			1	5	U
ITEM	VAL	KIPØ1			6	1Ø	U

Required: WAF to sort table KIP in ascending sequence.

Solution:

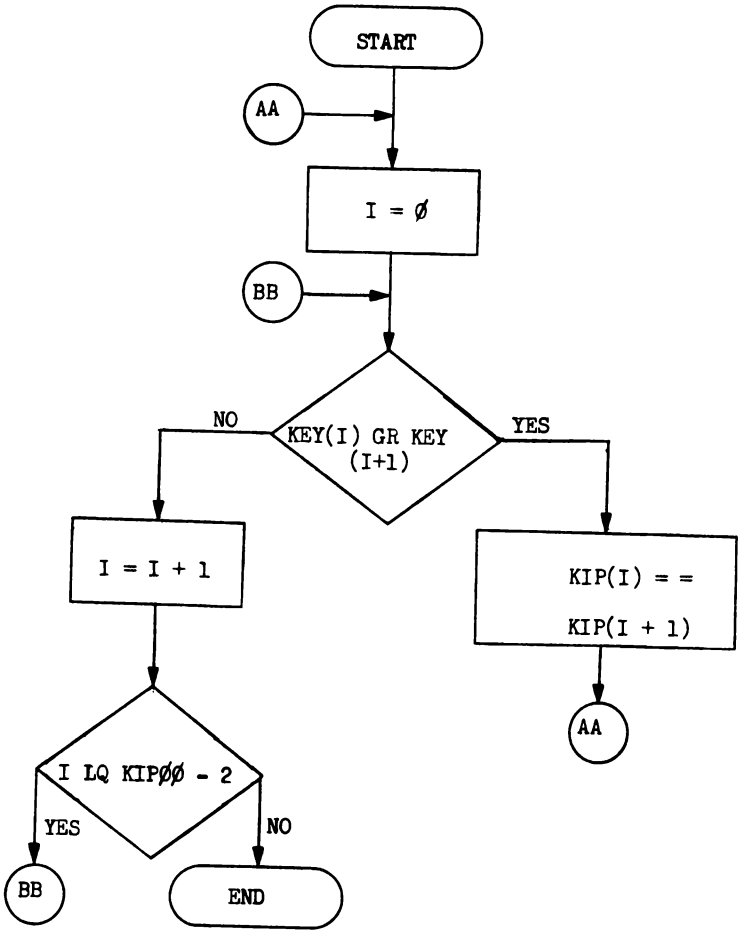


Figure 6-2.

Efficiency in terms of sorting time, as compared in the preceding two examples, leads us into a nebulous area. The condition of the table before it is sorted must be known before an accurate statement on efficiency of the two flows can be made.

If the original table is in a relatively ordered state, the variation flow is more efficient in that it handles fewer pieces of data. On the other hand, if the original table is in a relatively disordered state, the pass counter becomes the more efficient. Each sort had its advantages, but only at the extremes of disorder.

SINKING SORT

Another type of sort is the sinking sort routine. Since the determination of the end of the routine is based on the fact that one number is correctly positioned by each pass, would it not be sensible to ignore that value once it has been sorted, thus shortening the passes? In order to accomplish this, the program must set up and maintain an item against which the subscript governing the comparisons can be checked; as each pass is completed, this item is reduced by one. It is possible, further, to combine the function of the item described above with that of the pass counter in order to eliminate the need for both items. The sorting technique which operates in this fashion is called the SINKING SORT, probably because it sinks values to the bottom of the list and thereafter ignores them.

EXAMPLE 2

Given:

TABLE	GREEK	1ØØ 1 1ØØ	R	V	P
ITEM	ALPHA	GREEKØ1	1	5	U
ITEM	BETA	GREEKØ1	6	15	U
ITEM	GAMMA	GREEKØ1	21	7	U
ITEM	END	1 8 U			

Required: Sort the table into descending sequence on the basis of the value of GAMMA.

Solution:

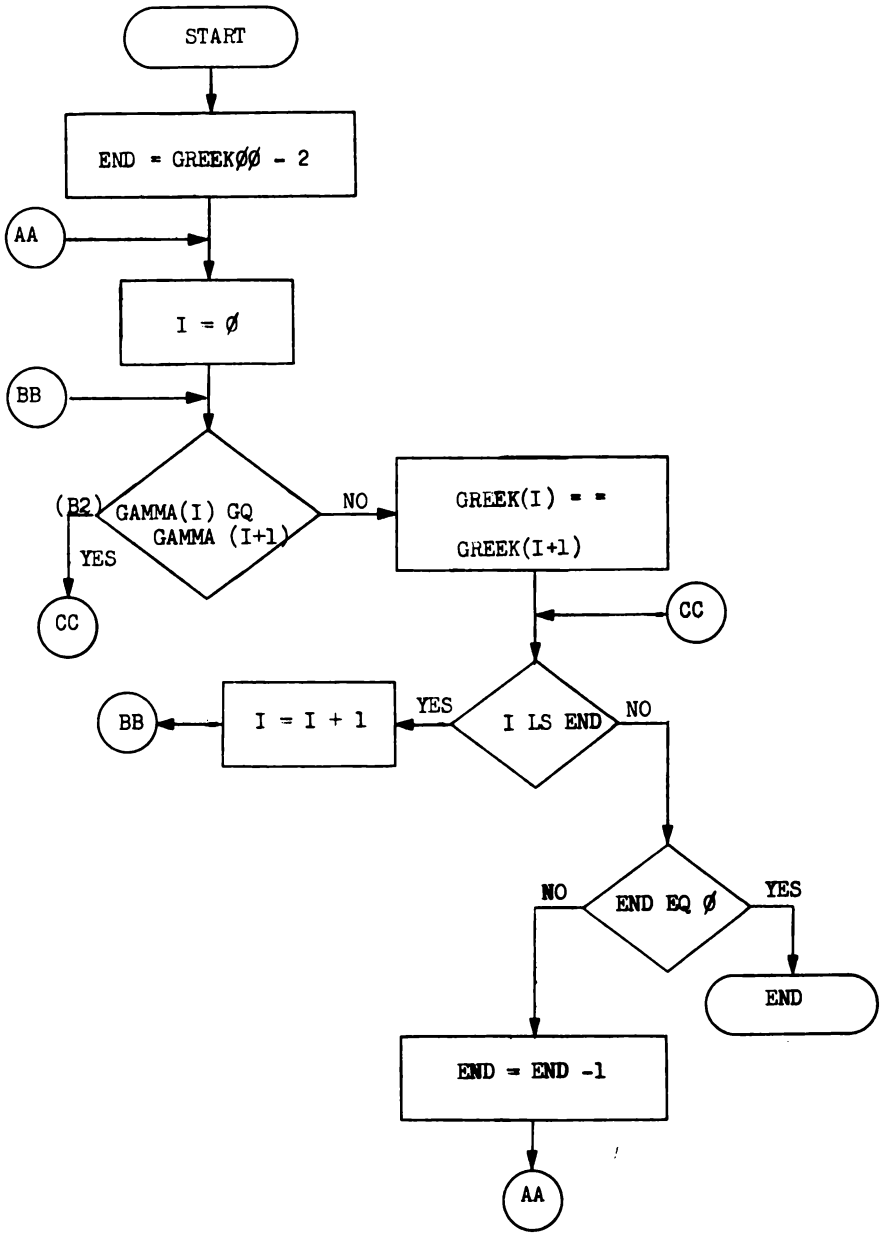


Figure 6-3.

The fact that this routine is written to sort into descending sequence alters only the decision at B2. The direction is, of course, reversed from that of an ascending sort. Note the use of the item END TO TERMINATE THE ROUTINE AS WELL AS TO SHORTEN THE PASSES. When END is equal to zero, the comparison which has just been made was between entries one and zero. When these are in order, the entire table is in order. Notice also the use of the double equal sign to indicate the exchange in the step following the decision at B2.

FLOATING SORT

Depending on their design, computers often show a preference for indexing up or for indexing down. For those computers whose preference is the latter, that is, starting at a high value and reducing to zero, the FLOATING SORT is an improvement over the sinking sort. As its name implies, this method involves floating numbers to the top of the table instead of sinking them to the bottom.

The exchange indicator, as added to the floating sort, has equal application to the sinking sort. Its purpose is one of efficiency. By eliminating unnecessary passes, it lessens the amount of data to be manipulated, thereby saving time. It is quite possible that a table could be sorted before a sufficient number of passes are completed to test the last two values (the function of item END in the sinking sort). If we assume this possibility occurs, a pass through the table will generate no exchanges. If, after each pass, we check an exchange indicator, which had been set prior to each pass, we could determine if an exchange were made and subsequently if the table happened to be sorted. Once we determine that no exchanges were made we know that the table is sorted and can stop the flow.

EXAMPLE 3

Given: (Floating Sort with Exchange Indicator)

```
TABLE   JOE      5Ø  1  5Ø   R    V   P
   ITEM KEY      JOEØ1       1    1Ø  U
   ITEM EXIND    JOEØ1      11    1Ø  U
```

Required: Sort table into ascending order based on KEY.

Solution:

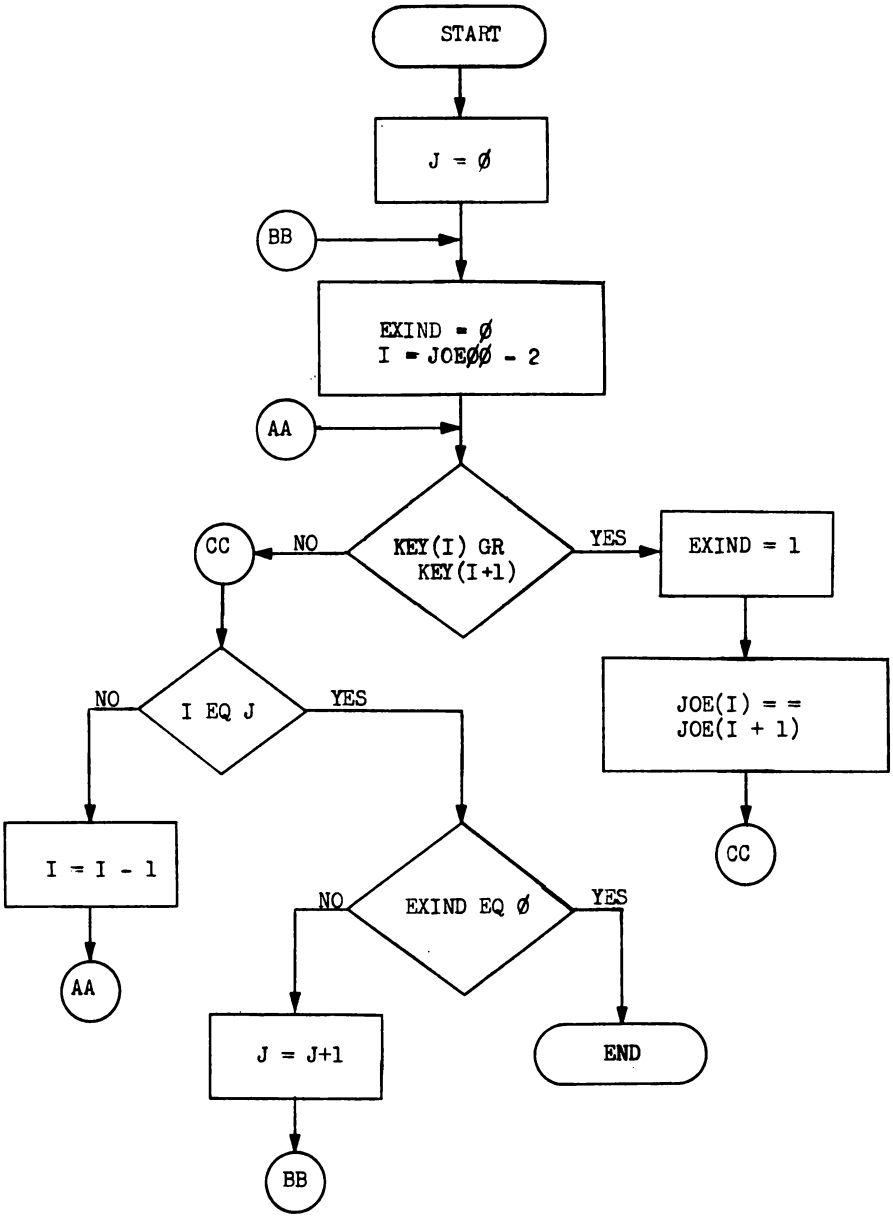


Figure 6-4.

SUBSTITUTE EXCHANGE SORT

The technique which follows could probably be best described as combining the characteristics of all the preceding methods. It is known as the SHUTTLE EXCHANGE sort, and is perhaps the best of the exchange sorting methods.

EXAMPLE 4

Given:

TABLE	POPULACE	1ØØ	1	1ØØ	R	V	S
ITEM	CITY	POPULACEØ1		1	48	H	
ITEM	STATE	POPULACEØ1		1	48	H	
ITEM	POPULATION	POPULACEØ1		1	3Ø	U	

Required: Sort the list into ascending sequence on the basis of POPULATION.

Solution:

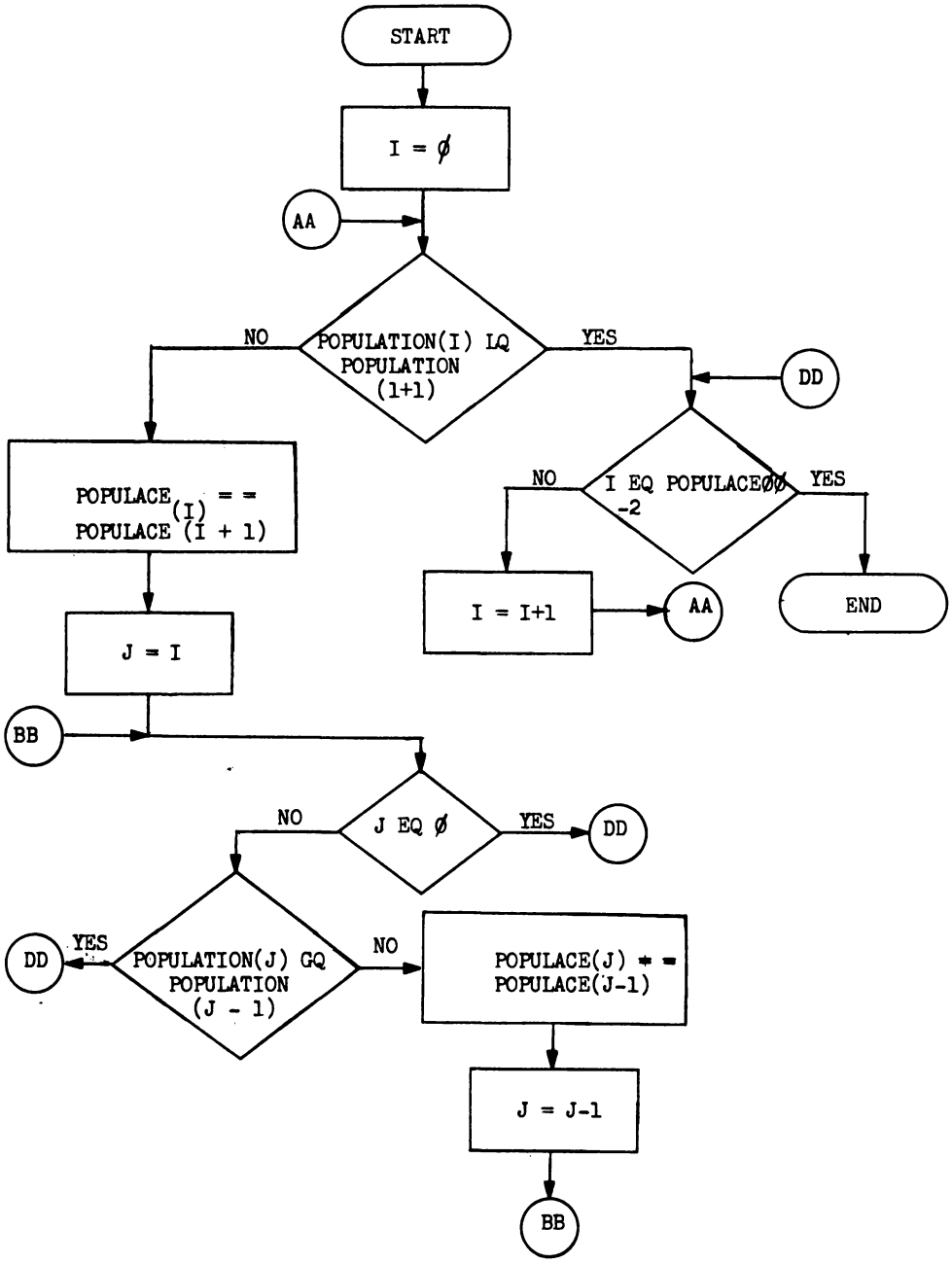


Figure 6-5.

As the reader can see, this method of sorting requires only one complete pass through the table, and it is governed by the subscript I. As many minor passes through the table as there are numbers out of order are required; they are governed by the subscript J. As each number which is out of sequence is found, it is not only exchanged, but "shuttled" upwards in the table until the entire table above I is in sequence.

DISTRIBUTIONAL SORTING

The distributional sorting techniques originate in the methods used to sort cards on EAM equipment. The number of passes required for sorting techniques of this kind is determined by the number of digits in the key, rather than the number of entries in the table. As a result, these methods are ideally suited to applications involving short keys.

Unlike the exchange sorting techniques, the distributional sorts require additional blocks of storage of the same size as the original table to be sorted. The number of blocks which are needed depends on the particular method used and the number base of the values being sorted, and ranges upward from one. Clearly, the old problem of time versus space is once again present. When space is available, the distributional methods are used; when it is not, time must be sacrificed and the exchange methods used.

In general, distributional methods operate by constructing "pockets" to receive the entries of the table. There must be as many such pockets as there are digits in the number system of the values being sorted, and each must be of the same length as the original table. The values of the table are distributed into these pockets on the basis of the values in one digit position. When this operation is complete, the values are reassembled in order, and the process is repeated for the next digit position of the value. The repetitions continue until all digit positions have been processed, at which time the reassembled table is sorted.

EXAMPLE 5

Problem: Sort the following list of decimal values into ascending sequence, using the distributional method:

25Ø	ØØ4	173	Ø9Ø	65Ø

Solution: Ten pockets must be constructed, each capable of containing five numbers. The pockets will be labeled with the values zero through nine. The passes through the list will begin with the right most digit and continue left until the third pass is completed, when the list should be sorted. The first pass produces in the zero pocket the values 25Ø, Ø9Ø, and 65Ø in that sequence; the three pocket contains 173; the four pocket contains ØØ4. All other pockets are empty. The list is now reassembled in order to produce 25Ø, Ø9Ø, 65Ø, 173, and ØØ4. The second pass, operating on the second digit position from the right, produces in the zero pocket ØØ4; in the five pocket 25Ø and 65Ø; in the seven pocket 173; in the nine pocket Ø9Ø. All other pockets are empty. Reassembly yields ØØ4, 25Ø, 65Ø, 173, and Ø9Ø. The final pass produces ØØ4 and Ø9Ø in the zero pocket, 173 in the one pocket, 25Ø in the two pocket, and 65Ø in the six pocket. When these are reassembled, the list is sorted and reads ØØ4, Ø9Ø, 173, 25Ø, and 65Ø.

While the use of the binary number system reduces the bookkeeping problem considerably, as well as the additional storage problem, it will, of course, introduce the necessity for many additional passes, since values represented in binary require more digit positions. The

example above was typical of the RADIX DISTRIBUTION sorting method for which a flow chart will be provided in the example which follows.

In order, however, to flow chart a method which requires access to individual bits of a value, it is necessary to introduce a method of indicating on a flow chart that access is being made to part of an item. As one might expect, this access is indicated using the word BIT. But, in addition to BIT, one must specify which bit or bits are being referenced. This is accomplished by using a subscript, which may be either a constant or a variable, to indicate the bit. If more than one bit is being reference, the first subscript indicates the beginning bit position and the second the number of bits. The bits in every register are numbered from one at the left of the item. And, of course the item itself must be specified, along with its subscript if it has one. This information is enclosed in parentheses after the bit subscript. Thus BIT(I,2) (ALPHA(A)) indicates 2 bits of the item ALPHA appearing in entry A. The two bits begin with bit I and end with bit I + 1.

EXAMPLE 6

Given:

TABLE	ORIG	1ØØ 1 1ØØ	R	V	P
ITEM	BEGIN	ORIGØ1	1	15	U
TABLE	OZR0	1ØØ 1 1ØØ	R	V	P
ITEM	JOKER	OZR0Ø1	1	15	U
TABLE	ONE	1ØØ 1 1ØØ	R	V	P
ITEM	POKER	ONEØ1	1	15	U

Required: Using the Radix Distribution Method, sort the table into ascending sequence.

Solution: 1. Data Description

Use the receiving tables, OZRO and ONE, which must be defined prior to program operation.

S controls table ORIG

A controls table OZRO

B and C control table ONE

I controls the bit number and the number of passes

Solution:

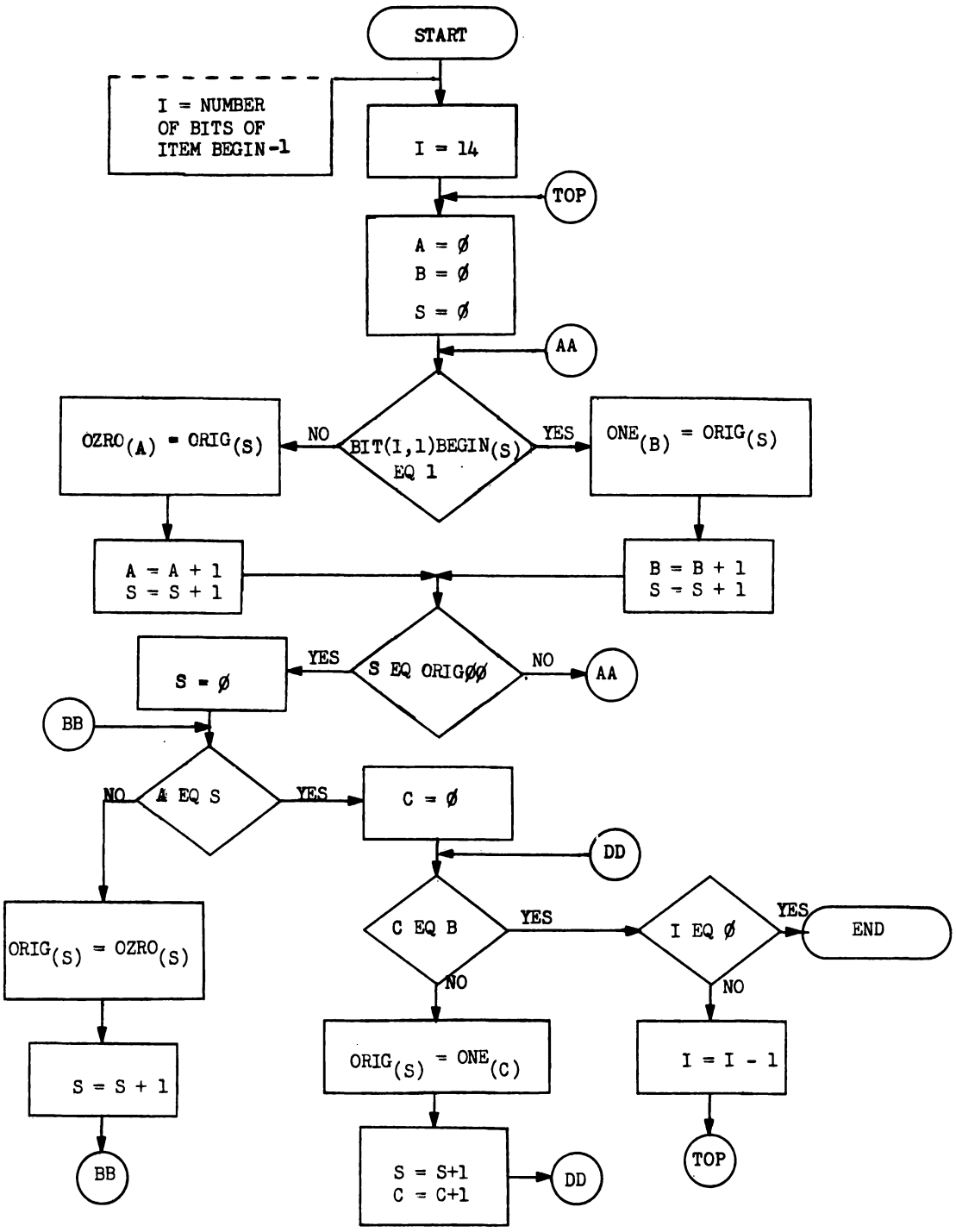


Figure 6-6.

During the operation of this program, the table containing the table, OZRO is used to contain the entries whose digit position under inspection contained a zero; similarly one contains those values of VALUE which contained a one

in the bit position being inspected. The subscript I is used to mark off the digit position being inspected as well as to count the passes, which will result in the termination of the program's operation. The subscript S is used to cycle through table ORIG in order to separate them into the two pockets, as well as to reassemble the values into the original table at the conclusion of each pass. The A, B, and C, subscripts are responsible, respectively, for the control of the storage into the tables containing zero and one. During the reassembly of the values, they control the number of values which are read out of their respective tables and back into ORIG.

BUCKET-LINK SORT

A bucket-link sort works like any other sort. It takes a group of items and sorts them into homogeneous groups or "buckets". A simple example of a bucket sort would best explain its workings. Given - 100_{10} random numbers ranging from Ø to 99. Required - Sort the 100 numbers so that all numbers ending in zero are grouped together; all numbers ending in one are grouped together; etc. A bucket sort would sort them based on their last digit (units position). Since there are 10 possible last digits (Ø - 9), there would be 10 groups or buckets. All numbers ending in zero would be in bucket Ø (Ø, 1Ø . . . 3Ø, 4Ø . . . 9Ø). All numbers ending in one would be in bucket 1 (1, . . . 51, . . . 81), etc. Each bucket would be ten registers in length. The buckets themselves could then be ordered in ascending or descending order. Of course, this would depend on the programmer writing the sort.

Making a bucket sort efficient (not wasting core storage) would require that the programmer know exactly how many items would be placed in each bucket, so that he could reserve sufficient registers. However, rarely does a programmer know the exact values of the data his sort will manipulate (how many Ø's, 1's, 2's, 3's, etc.). Therefore, he doesn't know how many registers to reserve for each bucket.

Suppose a bucket sort is going to sort 200_{10} numbers. To handle the most extreme case the programmer should make all buckets (and there should be ten) 200_{10} registers in length. This would handle the unusual case where all 200 numbers end in the same digit. It would also handle any other case. Under these conditions 2000_{10} registers would be reserved - 200 registers for bucket Ø, 200 for bucket 1, 200 for bucket 2, etc. As you can see this is an extremely inefficient sort.

However, a bucket sort can be made efficient if chaining of items is used. Chaining is the linking together of entries in a table. Figure 6-7 illustrates a parallel table structured around item BB. As can be see, the table is in ascending order based on BB.

Now look at Figure 6-8. This is also an ascending table structured around item BB. However, item AA is used as a chaining item. It tells the program processing the table where the next sequential item can be found. With this type of table, the using program has to be told where to find the first entry, since it doesn't necessarily appear in entry Ø of the table. This is done with a control item. The table in Figure 6-8 uses entry Ø as the control item. By inspection of the control item we can see that the first entry (or lowest number) is entry 4. At entry 4, BB = 6 and AA = 1. AA tells us where to look for the next number - entry 1. BB(1) = 1Ø which is the next sequential number. Now look at AA(1). It says entry 2 will contain the next number. It does - BB(2) = 12 and AA(2) = 7. You can go through the rest of the table on your own and verify that it is an ascending table. The entire table is held together by chaining. Logically this table is structured sequentially even though the numbers are scattered randomly. There must be an indication when the last entry in the table has been accessed. When AA = Ø the end of the table has been reached. Entry 5 is the end of the table.

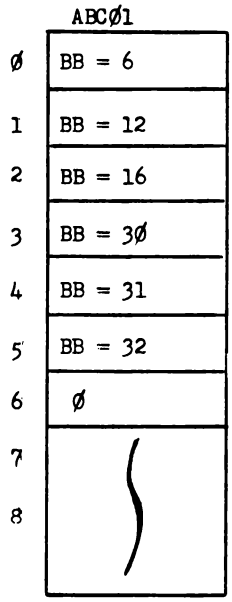


Figure 6-7.

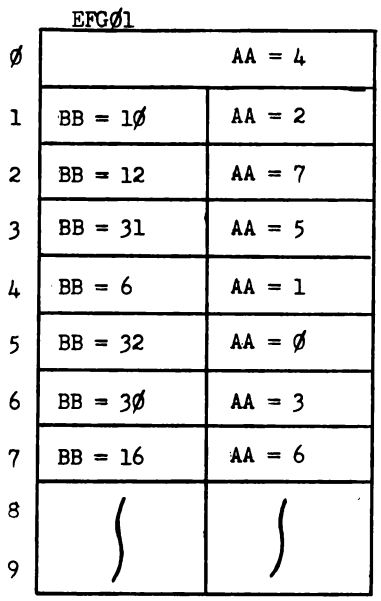


Figure 6-8.

The Air Defense Program for BUIC III combines the bucket sort and chaining into a very efficient bucket link sort and uses it as a track number conversion routine. The following is a simplified explanation of the Track Number Conversion routine.

Given:	TABLE	STN	81	1	81	R V P
	ITEM	SDLI	STN1	24	7	U
	ITEM	STRN	STN1	31	18	T
	*ITEM	STNO	STN1	1	48	U
	TABLE	LINK	1Ø	1	1Ø	RRP
	TABLE	TSD	81	1	81	R V P
	ITEM	TREN	TSD1	31	18	T

*STNO - Indicates next available channel in STNØ1.

STNØ1 - Sorted track number table.

SDLI - Refers to the channel (entry) number of the next track in this bucket. A value of zero indicates that this is the last track number in this bucket.

STRN - Contains the track number of the sorted track. Track numbers have the format LL/DD or L/DDD and are coded in unique Track format. "D" can range from Ø to 9. "L" can range from A to Z.

*STNO is unique to the STN table, it appears only once and occupies the entire register of the first register of STN.

LNKØ1 - Linkage table. This table contains the address of the first entry for each bucket in STNØ1. LNKØ1(Ø) contains the channel number of the first track whose last digit is zero. LNKØ1(1) contains the channel number of the first track whose last digit is one, etc.

TSDØØ - Contains number of meaningful entries in TSDØ1.

TSDØ1 - Tracking safe data table. Contains unsorted track numbers.

TTRN - Track numbers. Has identical format as STRN.

Initial Conditions:

TTRN - Contains unsorted track numbers.

LNKØ1 - is clear.

STNO - 1 (See Figure 6-9).

STNØ1 - is clear except for SDLI (Figure 6-9).

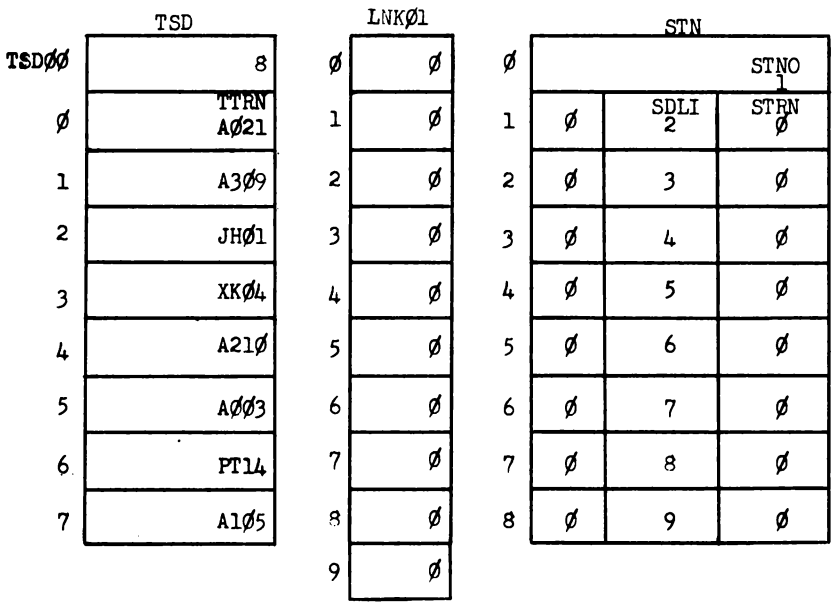


Figure 6-9.

Let's use the first three entries of TSD1 to illustrate the bucket sort. The first track number is AØ21. The last digit in $TTRN_{(1)}$ is a one. The value one is used as an index value into LNKØ1. The entry that we would inspect in LNKØ1 would be $LNKØ1_{(1)}$. The entry value is a zero which indicates that no previous track number has been assigned with a last digit of one. Now we wish to store AØ21 into the proper (or next available) channel in STNØ1, and place this channel number in $LNKØ1_{(1)}$. The value of the control word STNO is the next available channel in STNØ1. STNO equals one. We then set LNKØ1 = STNO. This means that the 1's bucket will start in $STNØ1_{(1)}$. $STRN_{(1)}$ = $TTRN_{(1)}$; $STNØØ = SDLI_{(1)}$ set the down link, SDLI, to zero. The three tables now appear.

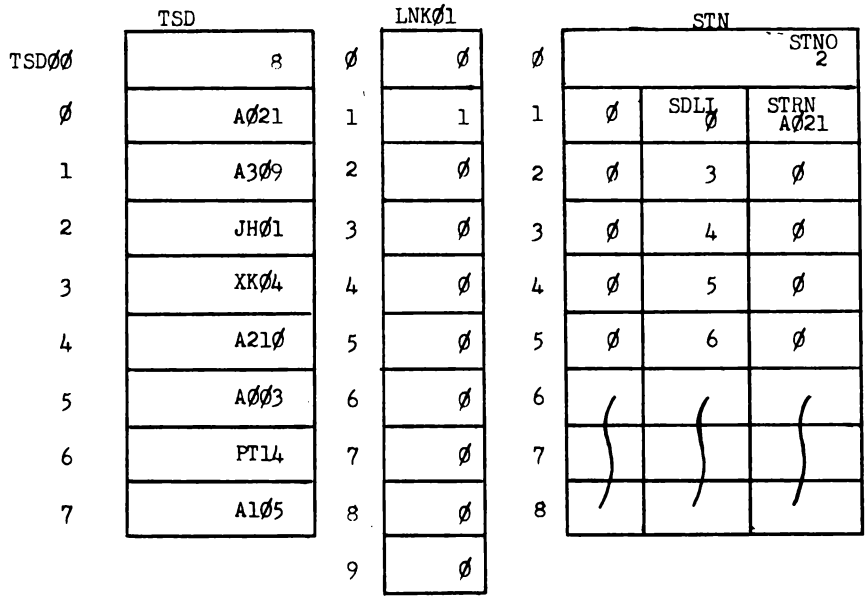


Figure 6-10.

The next track number in TSD1 is A3Ø9. Its last digit is nine. The value nine is used as an index value into LNKØ1. Inspection of $LNKØ1_{(9)}$ shows that this entry contains a zero. This indicates that this is the first track number with nine as the last digit. A new nine's buckets is now being started. We wish to save the address of the first entry of the 9's bucket. STNO contains the address of the next available channel in STNØ1. The following occurs. $LNK1_{(9)}$ = STNO; STNO = $SDLI_{(2)}$, $SDLI_{(2)}$ =Ø; $STRN_{(2)}$ = $TTRN_{(2)}$. At this point we have set up a 1's bucket and a 9's bucket. The three tables now appear.

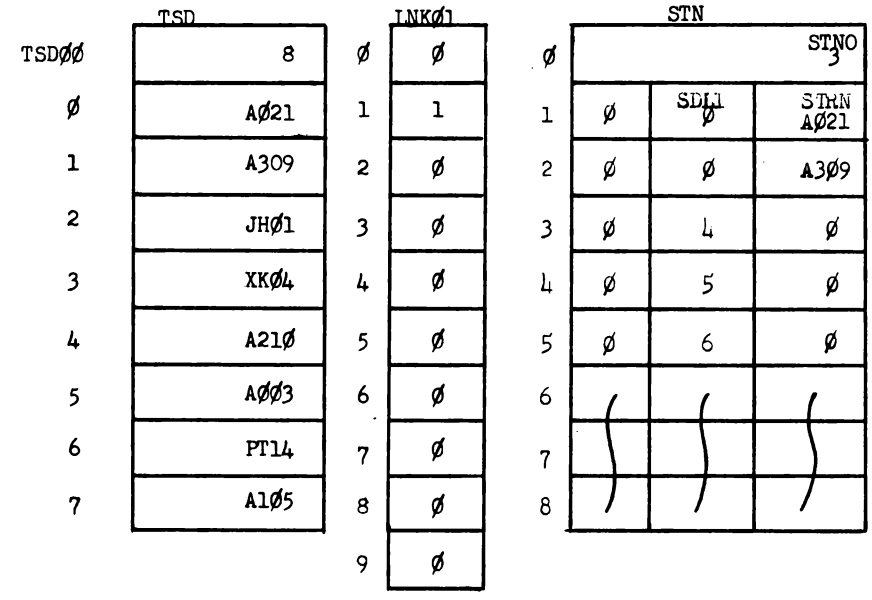


Figure 6-11.

Track JHØ1 is now ready to be sorted. The last digit of the track is one. This track belongs in the 1's bucket. $LNK1_{(1)}$ is examined to see if it equals zero - it doesn't. The value in $LNK1_{(1)}$ is the first entry in the 1's bucket. Using the value in $LNK1_{(1)}$ as an index value into STNØ1 we inspect $SDLI_{(1)}$ to see if it is the last entry in the 1's bucket. $SDLI_{(1)}$ equals zero which says this is the last entry in the 1's bucket. STNØØ is then checked for the next avaialable channel and that value is stored in $SDLI_{(1)}$. $STRN_{(3)} = TTRN_{(3)}$, $STNO = SDLI_{(3)}$, $SDLI_{(3)} = Ø$. $STNØ1_{(3)}$ is now the last entry in the 1's bucket. The three tables now appear.

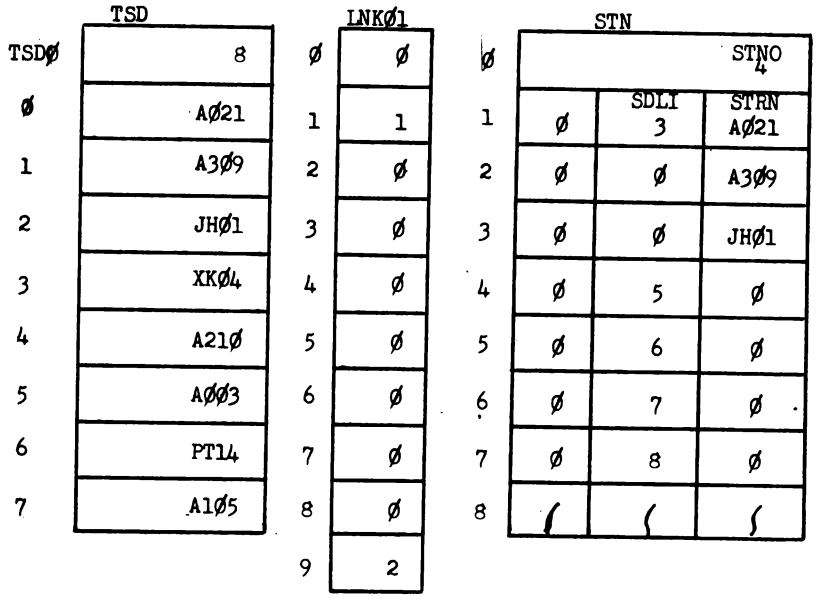


Figure 6-12.

Notice that LNKØ1 hasn't changed. It changes only when a new bucket is started.

On your own you can sort the rest of TSD1.

The above discussion has shown ordering a table. A more sophisticated sort could include uplinks as well as downlinks and could involve deleting track numbers. This sort will be discussed in class and incorporated into the more advanced areas of the course.

DELETION

When a variable length table exists, its control item, NENT, is expected to reflect the current number of meaningful entries in the table. Further, these entries are expected to be stored in the slots numbered from zero to n-1 in every case. If then, it becomes necessary to delete entries from the table, the process will involve moving entries around within the table. The criterion used to bring about the deletion of certain entries from a table is, of course, dependent on the specific requirements of the processing problem; the techniques employed for the actual moving of the records, however, can be generalized.

Deletion, and REPACKING, which is the moving up of MEANINGFUL entries to fill slots vacated by deleted entries, is a rather time consuming process, especially if the table is long. There are numerous methods of accomplishing the task, as is the case with most standard processes for computers. Each of the techniques, however, employs the principle of "covering" an entry which is to be deleted with an entry which is not to be deleted.

EXAMPLE 9

Given:

TABLE	PERSNL	5ØØ	1	5ØØ	R	V	S
ITEM	MAN'NO	PERSNLØ1		1	2Ø	U	
ITEM	HIRE'DATE	PERSNLØ1		21	2Ø	U	
ITEM	SALARY	PERSNLØ1		1	2Ø	U	16.Ø4

Required: Delete entries with MAN'NO = Ø. Repack the table maintaining original order.

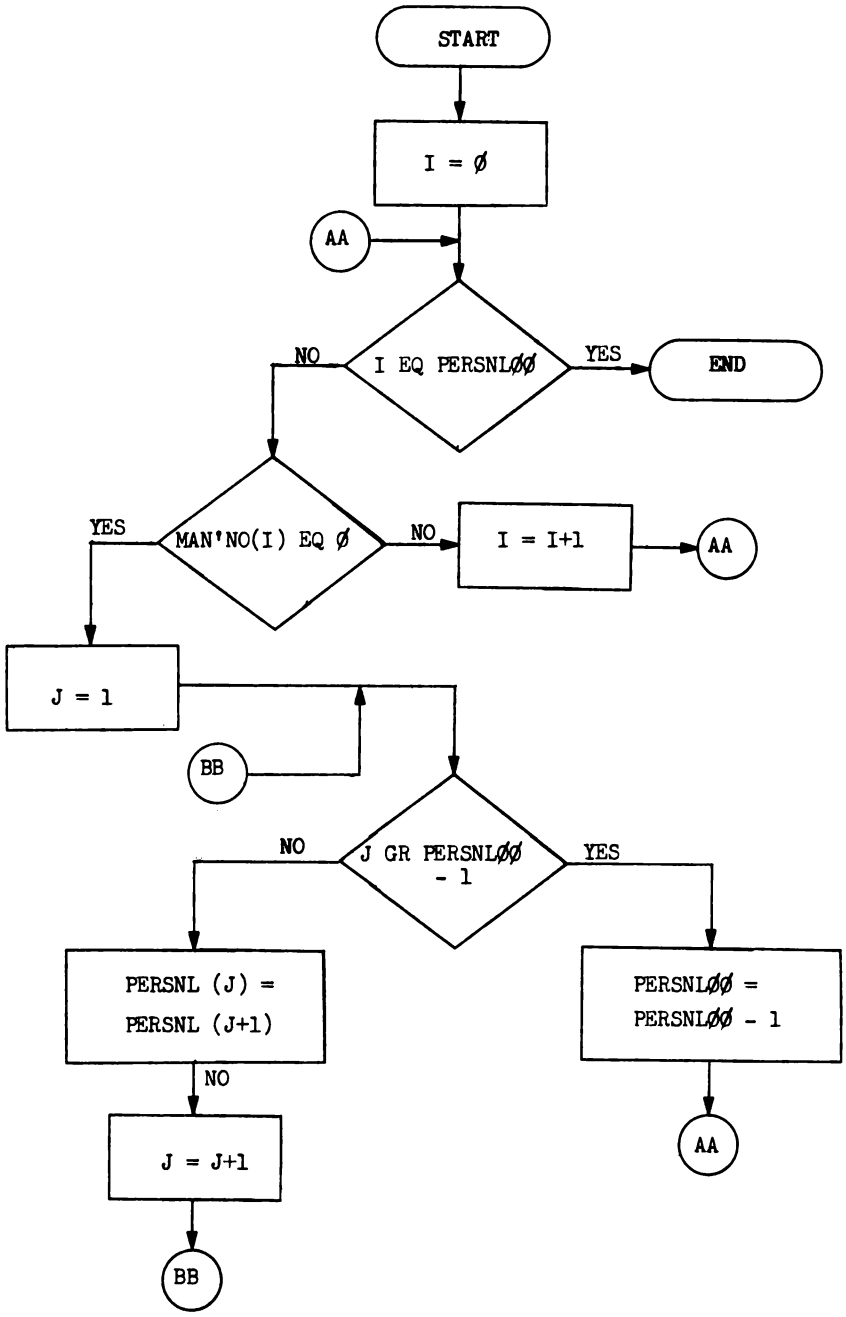


Figure 6-13.

The method used in Example 9 is perhaps the most obvious technique for accomplishing a deletion. When an entry containing MAN'NO equal to zero is found, all subsequent entries are moved up one slot to cover the meaningless entry which was found. This leaves, at the bottom of the table, a duplication of the last entry; however, when the last entry has been moved up, PERSNLøø is reduced by one so that that entry is no longer a part of the table.

In this example, the subscript I is used to cycle through the table inspecting for cases where MAN'NO is equal to zero. When such an entry is found, the subscript J is set to cycle

through the remainder of the table moving entries up. Notice that when the J cycle is completed, the program returns to step AA without increasing the subscript I. This procedure is necessary since it is possible to encounter two entries containing MAN'NO equal to zero in sequence. It is thus possible that we have moved another entry containing MAN'NO equal to zero into the slow occupied by the first encountered meaningless entry. We must now inspect the entry moved into that slot.

This technique for deleting meaningless entries, while it preserves the order of the table, moves information around a great deal . . . often unnecessarily, as in the case mentioned above. Therefore, while it is the most obvious method, it is certainly not the most efficient.

EXAMPLE 10

Given:	TABLE	PARTS	1ØØØ	1	1ØØØ	R	V	P	
	ITEM	STK'NO	PARTSØ1			1	15	U	
	ITEM	PRICE	PARTSØ1			16	20	U	13.Ø7

Required: Delete those entries containing STK'NO equal to zero.

Solution:

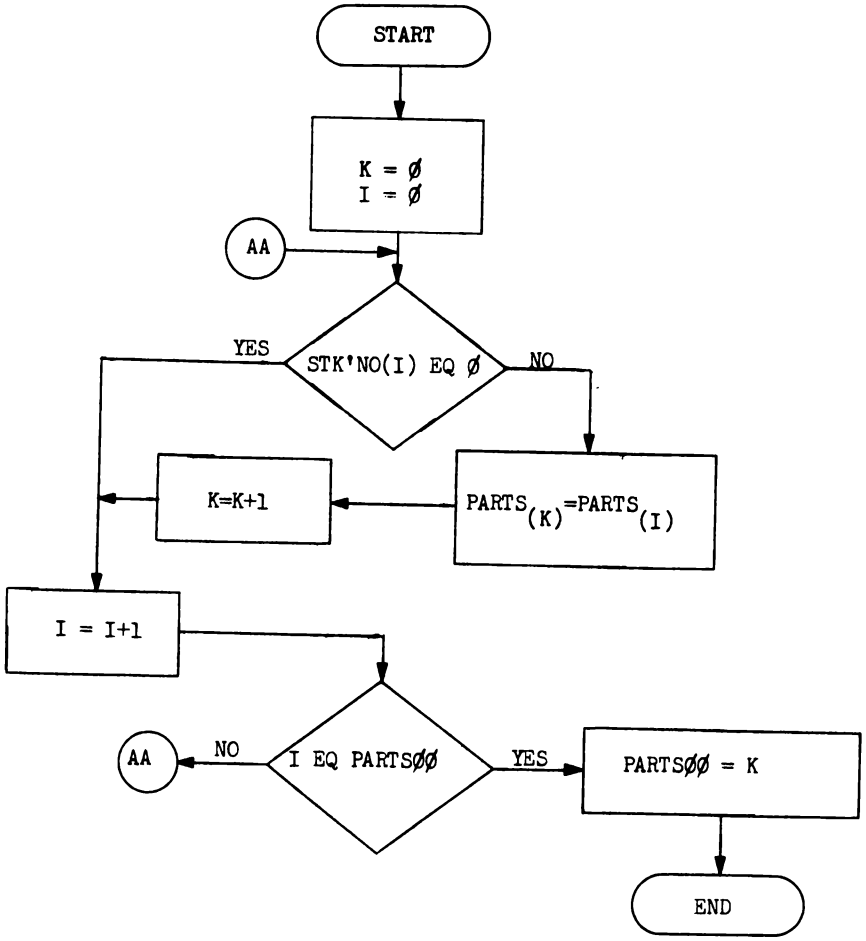


Figure 6-14.

Deletion methods have been presented with varying levels of efficiency based on the criteria mentioned above. To summarize, the deletion of entries from a table, first of all, requires that the table have a variable length and the control item which goes along with variable length. The addition or removal of entries from such a table requires the modification of this control item. Although this was not expressly pointed out in the examples above, it is important to notice in all of the routines illustrated the effect that the modification of NENT has on the testing of subscripts throughout the routines. Of course, if NENT has been modified during the execution of any part of a loop, its most recent value will be used in any test involving it. The criterion used to determine whether or not an entry is to be deleted depends, of course, on the specific problem. In most cases, however, the criterion does not involve all of the items in the entry. Nonetheless, when the entry is deleted, the ENTIRE entry is deleted and replaced by an entry which is judged to be meaningful. This process can be illustrated on the flow chart by specifying entry along with the name of an item or the name of the table and a subscript which specifies which entry is being referenced.

INSERTION

If an entry must be inserted into a variable length table an insertion method must be used. The same criteria apply to insertion as do to deletion; that is, the preservation of the original order of information in the list, the avoidance of unnecessary movement of entries, and the updating of NENT.

EXAMPLE 11

Given:	TABLE	ORIG	5ØØ	1	5ØØ	R	V	P
	ITEM	KEY	ORIGØ1			1	48	F
	ITEM	DATA	1		48	F		

Required: Insert item DATA into proper place in table.

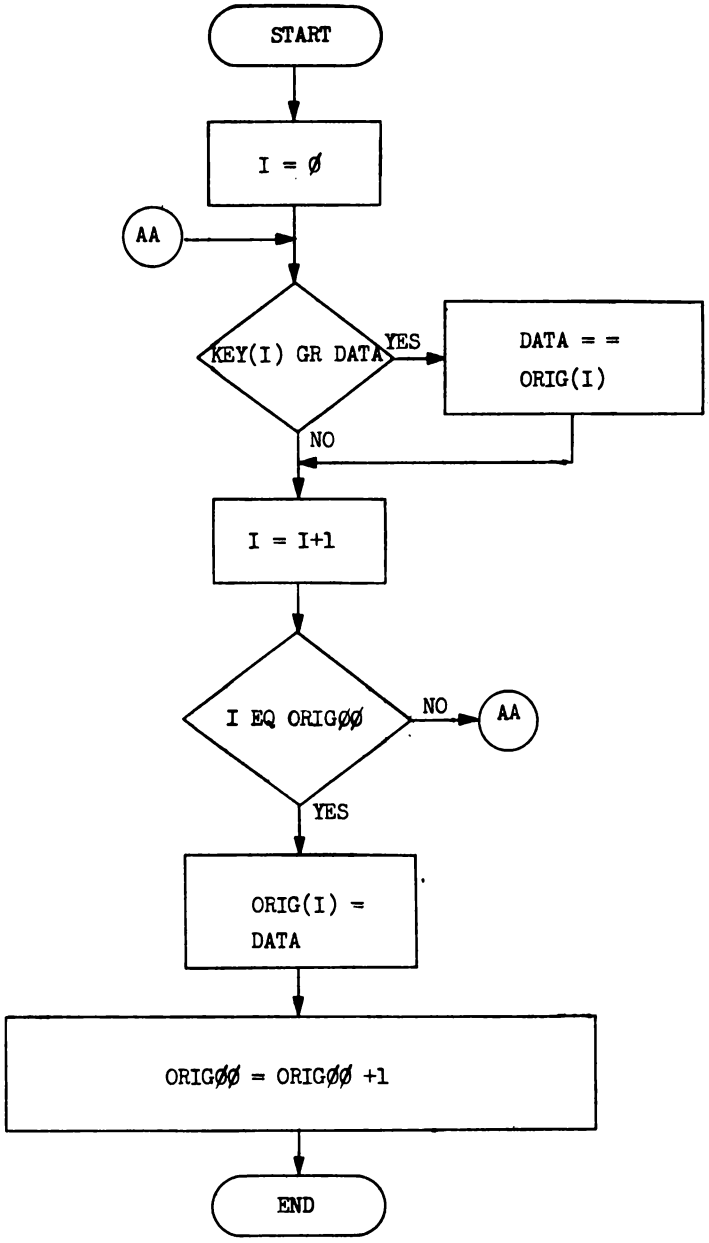


Figure 6-15.

In this insertion routine a slot-by-slot search is made, comparing DATA with KEY(I). As soon as the proper place is found for DATA it is exchanged with ORIG(I). After that, each following ORIG(I) is exchanged with new DATA items being put there by previous exchanges until the end of the table is reached. At this point, the new item has been inserted and the last entry of the original table is in item DATA. This is then put at the end of the table, NENT is updated and the insertion routine is complete (note at this point item DATA and the last entry in the table are identical).

MERGING

Merging, since it is a technique which processes tables which are ordered, is perhaps the simplest of the sorting methods. In general, merging involves the combination of two or more similarly sorted tables into one larger table sorted in the same sequence as the original tables. It may be used with ascending and/or descending tables, the resultant table being ascending or descending.

The technique is based, as are all the sorting techniques, on the comparison of key items from two entries to determine which belongs first in the sorted table. In the specific case of merging, the comparison is performed between the "top" entries of the two tables to determine which belongs first. When the correct entry is found, it is stored in the new table, and the index for the table from which it came is increased to indicate a new "top" for that table.

Again a comparison is made between the top keys of the two tables to find the one which belongs next in the new table. This process is repeated until one of the tables is exhaused, when the remainder of the new table is filled with the remaining entries in the unexhausted table. When both tables have been exhausted, the process is complete.

EXAMPLE

Problem: Merge the following tables of sorted values into a third table also sorted into ascending sequence.

TABLE 1	TABLE 2
12	3Ø
15	31
45	6Ø
92	73
	81

Solution: The first comparison is between the values 12 and 30; 12 is found to be the smaller, and is therefore the first value in Table 3. The next comparison is between the values 15 and 30, since the top of Table 1 has now been changed, while the top of Table 2 is the same. Again, the desired entry comes from Table 1, causing Table 3 to contain, in sequence, 12 and 15. The next comparison is between the values 45 and 3Ø. The value in Table 2 is selected, increasing Table 3 to contain 12, 15, and 3Ø. The values 45 and 31 are now compared. Since 31 is chosen, Table 3 will contain 12, 15, 3Ø and 31. This process is repeated, always selecting the smaller of the two values compared since the end result is to be in ascending sequence, until both tables are exhausted, when the following has been produced:

TABLE 3

12
15
3Ø
31
45
6Ø
73
81
92

Notice that more than once several selections in sequence were made from the same table for the final table. Refer to Figure 6-16 for the Flow Chart.

Given:	TABLE	ONE	5Ø 1 5Ø	R	R	P
	ITEM	KEY 1	ONEØ1	1	1Ø	U
	TABLE	TWO	1ØØ 1 1ØØ	R	R	P
	ITEM	KEY 2	TWOØ1	1	1Ø	U
	TABLE	THREE	15Ø 1 15Ø	R	R	P
	ITEM	KEY 3	THREEØ1	1	16	U

Solution:

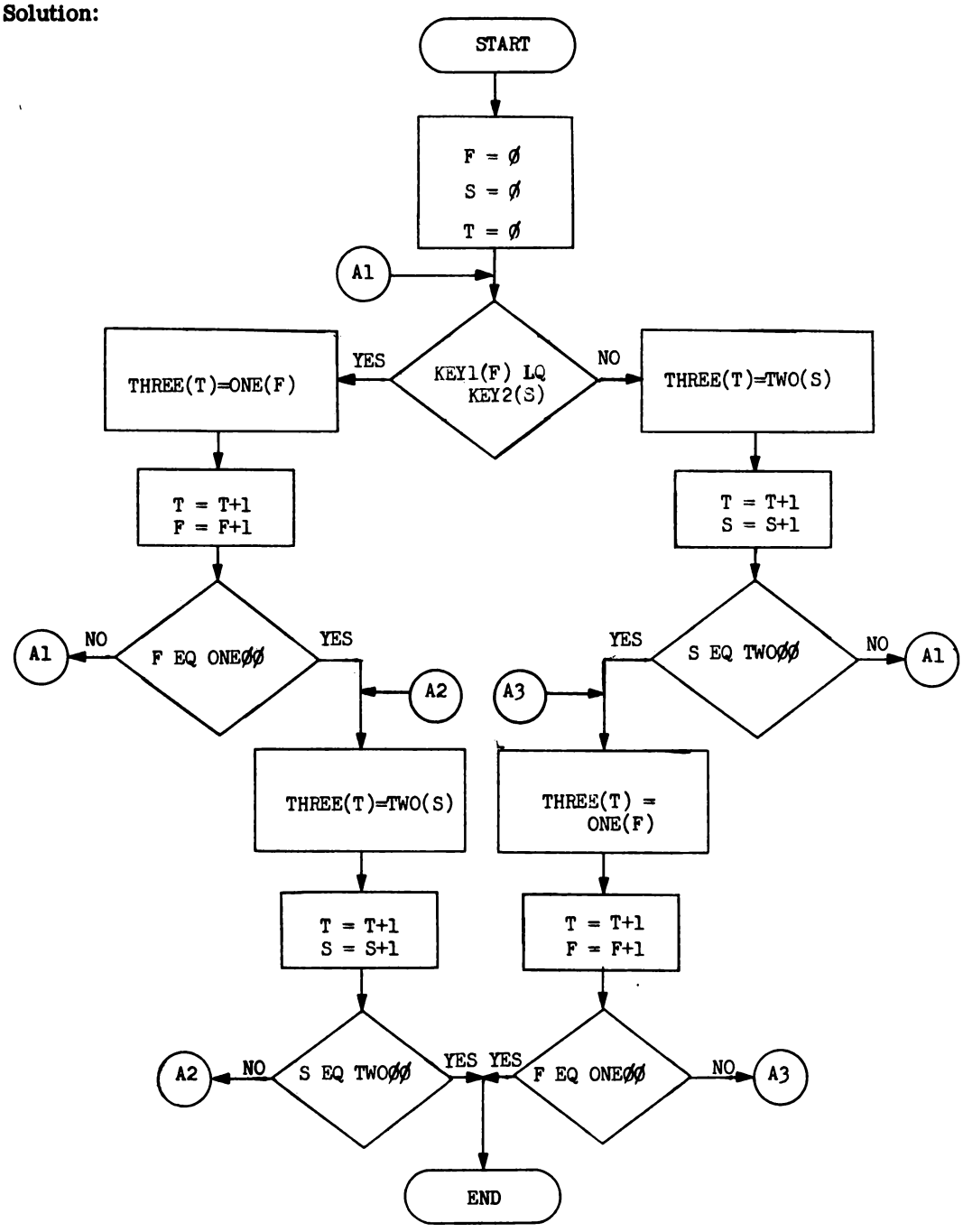


Figure 6-16.

SUMMARY

The sorting methods described and flow charted in this document are by no means all of the methods available. They have been presented both for their intrinsic values as sorting methods and as examples of data processing techniques. The student is encouraged to look at them in this light, and to consider the details of them as representative of the problems to be encountered in data processing.

CHAPTER 7

SUBROUTINES AND SUBROUTINE LIBRARIES

INTRODUCTION

The computer programmer, when setting down the solution to a problem, often finds that the same task or several similar tasks must be repeated often in the program. Frequently, the only variation to be found among these tasks is in the data on which they operate. It's obvious that both programmer time and program space would be wasted if the sequence of instructions needed to perform these tasks had to be rewritten every time the occasion arose to use them. Indeed, if the task to be performed is an often used one, the programmer would certainly prefer not to write the code at all, but to incorporate into his program a set of instructions which have been prepared beforehand to perform the desired function.

It is easy to imagine that certan mathematical operations, such as sine or cosine computation, square root, cube root, or certain data processing functions, such as sorting, deleting, or merging might readily fall into the category of frequently used functions for which the computer instructions could be pre-prepared. The need for such routines is so universal, in fact, that computer manufacturers frequently provide many of them with the computer when it is purchased; in addition, individual programmers, when writing such routines, write them so that they can be used by others, and "users groups" are formed along the owners or users of individual computers for the express purpose of exchanging information of this kind.

DEFINITION

A subroutine is a set of instructions which may be used and reused by a single program or by many programs, to perform a well-defined task.

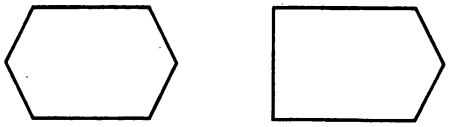
The essential difference between a subroutine and a program designed to perform the same function is that the subroutine must contain extra instructions within it to link it to the using program. This linkage provides (1) that the subroutine can be called and operated by any program, (2) the values that the subroutine needs to operate, (3) a location for the computed results, and (4) a return location to the calling program. Subroutines can usually be located somewhere in memory.

There are two types of subroutines. They are classified on the basis of their availability to the programmer. An open subroutine is one created by a programmer for specialized use only by his program. Since the writer of the program and the subroutine are the same person, the subroutine can be changed and is "open" to modification by the programmer. Keep in mind that the open subroutine is usually so specialized that its use is limited only to the program of which it is part.

The closed subroutine, on the other hand, is very general in nature with a wide range of applications. It is written by one programmer for use by other programmers. From the outlook of the using programmer, the subroutine is "closed" to modification. Some common closed subroutines are binary to octal conversions, rewinding of the tape drives, as well as trigonometric functions. The using programmer must take the pains to insure that he utilizes the subroutine within the limitations set forth by the programmer who created the subroutine.

The subroutine is called upon to operate by means of a jump to its beginning address. This jump instruction, located in the main routine, is known as the SUBROUTINE CALL. For

the purposes of the flow chart, the subroutine call as well as the operation of the subroutine is indicated by the symbol in Figure 7-1, containing the name of the subroutine:



CLOSED OPEN

Figure 7-1. Subroutine Linkage Symbols

The operation of the subroutine itself usually is flow charted elsewhere in detail.

When a subroutine has finished operating, it must know the return address for giving control back to the calling program. As a result of hardware design, this return address is available to the subroutine.

The method by which the return instruction(s) is set up is largely dependent on specific computer characteristics, but generally falls into one of two categories.

1. The address of the subroutine call can, under the control of the subroutine, be modified and stored into the instruction which will be used to return to the main routine.

2. The computer may recognize the subroutine call as a special instruction which, when it is encountered, will cause the return instruction to be set up automatically.

This process, though it may take several computer instructions to accomplish, is seldom illustrated on the flow chart of the subroutine.

CALLING PARAMETERS

Depending on the functions which they are designed to perform, subroutines may require no information from the main program, or they may require a great deal of information. Similarly, they may provide information to the main routine when they return to it, or they may simply perform a "service", such as rewinding a tape, which requires no return of information, or at the most, an indication of successful or unsuccessful completion of the operation.

When information is required by the subroutine, this information must be located at an address which is known to the subroutine or which can be computed by the subroutine. The information is known as the CALLING PARAMETERS to the subroutine. Where only one calling parameter is involved, the subroutine might expect it to be located at a fixed place in memory, or perhaps, in some arithmetic register. Even where more than one calling parameter is required, it is sometimes possible to locate these data in non-memory registers by making use of the CPU registers and the index registers. In any case, the locations of the calling parameters are specified by the writer of the subroutine since, of course, his program must know where to find the information with which it is to work. The responsibility of the writer of the main program is simply to insure that the require data is in the correct place when the subroutine is called.

At the flow charting level of programming, reference is seldom, if ever, made to the arithmetic registers of the computer or to the addresses of memory registers. Information, as the student is aware, is handled in items. As a consequence, the transfer of information

from the main program to the subroutine must be illustrated in terms of items and their settings. A subroutine manipulates items which it has defined, but whose values will be supplied by the main program. The main program, then has the responsibility, as was stated above, for insuring that the right values are in the right items. If, for example, a square root subroutine manipulated the item NUMB to produce the square root of its value, the main program would be obliged to set NUMB. In the main program, the item whose square root is to be taken might be called anything, perhaps XX. In order to distinguish between the transfer of information from the main program to the subroutine and the ordinary main program settings of items, the word ESTABLISH is placed in the function box. Figure 7-2 is used to illustrate such a transfer.

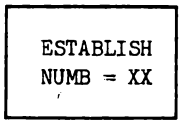


Figure 7-2. Setting Function for Parameters

CALLING SEQUENCE

When the number of calling parameters to a subroutine becomes so large that it is impossible or impractical to transmit them via the arithmetic registers of the computer, the need arises to set up the calling input parameters in the form of a CALLING SEQUENCE. It has been pointed out that the address of the subroutine call is made available to the subroutine in some way. Based on this fact, and the fact that the addresses immediately following the call may be easily computed by the subroutine, it has become common practice to place the calling sequence in the registers of memory which immediately follow the subroutine call. Once again, the order in which the parameters will appear in this sequence is determined by the writer of the subroutine, while the writer of the main program simply supplies them in the specified order.

This type of information transfer from the main program to the subroutine is illustrated in the flow chart using the word "establish".

OUTPUT PARAMETERS

The results of the operation of the subroutine must, of course, be made available to the main program. They, too, may be left in the arithmetic or control registers of the computer, or perhaps in memory locations designated by the main program. If the latter is true, the main program will have had to designate these locations by means of the input (calling) parameters. For example, the writer of a square root subroutine might specify that the number whose root is to be taken will appear in a particular register, and that the address into which the result should be stored will appear in another register. The subroutine can operate on the data, and, when finished, store the result in the specified register.

Mention has been made previously of subroutines whose functions do not require that they transmit any information back to the main routine. In such cases, of course, the subroutine simply returns to the main routine having set nothing. It is also frequently necessary that the subroutine have the capacity to indicate error conditions to the main program. This can be done in a number of ways, one of which is to transmit an output parameter whose value has special meaning in terms of error codes. Another technique used is to return to a different location in the main program when an error occurs.

EXAMPLE 1

The following subroutine is designed to convert an unsigned binary integer to its decimal equivalent, represented in Hollerith code. The subroutine will produce a value with a maximum of eight Hollerith characters. Zeroes preceding the first significant digit will be replaced by blanks. The value zero will be indicated by one zero right justified.

1. Subroutine data

```
TABLE    HOLE       1Ø  1  1Ø      R     R    P
   ITEM  HOLRTH     HOLEØ1         1     6    H
```

The table has been loaded with values such that HOLRTH(Ø) contains the Hollerith code for zero, HOLRTH(1) contains the Hollerith code for one, etc.

Non-tabular, ITEM ARITH 1 48 U. It is set up to contain the integer input parameter to the subroutine.

Non-tabular, ITEM ABC 1 48 H, set up to contain the output parameter from the subroutine.

Non-tabular, ITEM EFG 1 48 U It is set up to contain intermediate values for the subroutine.

ITEM EFT'I 1 1 B P Ø, is set up as an indicator for the internal operations of the subroutine. It is initially set to zero.

2. **Subroutine flow chart.**

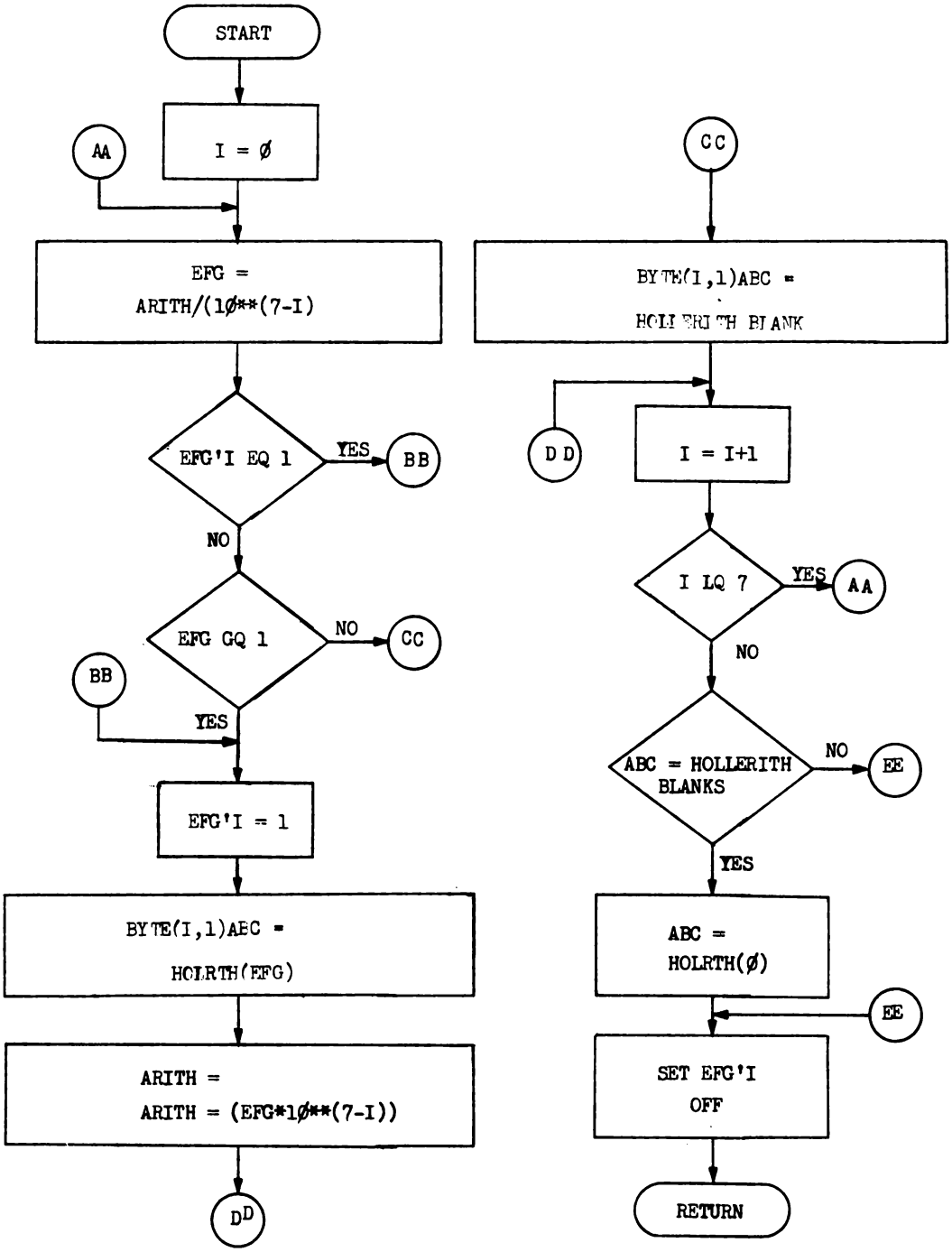


Figure 7-3.

The main program which follows is designed to use the subroutine CONVERT in order to convert the values in a table containing the item INT to Hollerith code, and to store the converted values in a table containing the Hollerith item OUT for eventual output. Both tables are variable length.

3. Main program flow chart

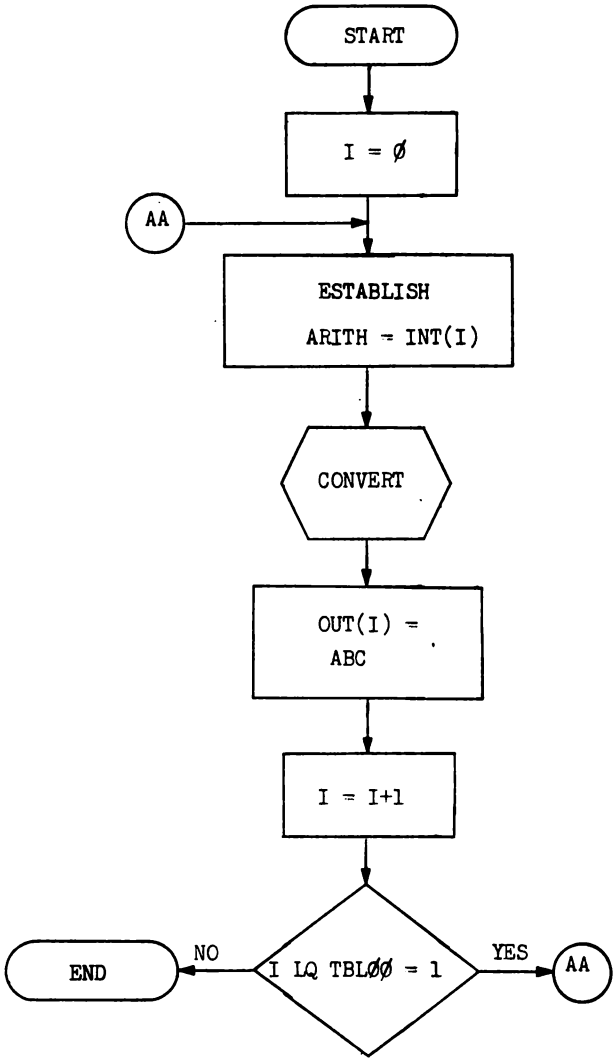


Figure 7-4.

SUBROUTINE LIBRARIES

A means of increasing the efficiency of problem solution and program preparation is the subroutine library. As its name implies, the subroutine library is a collection of subroutines. These subroutines are written so that they can be incorporated into any program. Those subroutines that constitutes the library are considered closed subroutines. The term external subroutine can also be applied to them, since they are external to the using program. The library of subroutines is normally stored with the Utility System on a magnetic tap, or apart from the Utility System, but on a tape which can be accessed by the System.

INCORPORATION OF A SUBROUTINE INTO A PROGRAM

Subroutines taken from a subroutine library may be incorporated into a main program in a number of ways. The most common methods are the following:

1. The subroutine is assembled with the main program. In this case, the symbolic program makes reference to the subroutine by its symbolic name. The assembly program, on recognizing this name finds the correct subroutine on the subroutine library tape, and attaches it to the program being assembled. Although the main program may refer to the subroutine more than once, the assembler is astute enough to attach the subroutine only once, usually at the end of the program. The binary deck or tape, then, which is produced by the assembler will include both the program as written and all subroutines used by it.

2. The subroutine is used directly from memory. In this instance, the subroutine is one which is normally in the computer memory during the operation of any program. Since this is the case, it is not necessary to repeat the subroutine with the main program, and it is therefore accessed directly at its normal address in memory. It is of course necessary, in this case, that the assembler recognize the subroutine to be one which falls into this category. When it does so, it will respond to the situation by replacing the symbolic reference to it with a reference to its fixed memory location.

TYPES OF SUBROUTINES

The programs usually contained in a Subroutine Library may be grouped according to the following classification:

1. Mathematical functions and processing procedures
2. Modes of operation
3. Service routines
4. Executive programs

The first three groups represent sets of subroutines, whereas the executive routines control the execution of the subroutines and the main programs which use them.

Examples of mathematical functions are trigonometric functions, matrix operations, square roots, etc.; processing procedures may include many kinds of file processing such as sorting, deleting, mergining, etc. Modes of operation include such things as floating point operations (for a computer which is not equipped with floating point, such that these operations must be simulated by a subroutine) or double precision operations. Service routines may perform such operations as input or output, conversions from binary to Hollerith and vice versa, or testing and checking.

SUMMARY

The purpose of subroutines is to save time and effort in the coding, programming, and testing phases of program production. A subroutine is a program written to perform a well defined task.

CHAPTER 8

INTPUT/OUTPUT PROGRAMMING

INTRODUCTION

Communication between the outside world and the central computer is accomplished through the computer's input-output (I/O) system. The input-output equipment is the means of communications between the user and the machine. In general, the I/O system is used to translate the coder's marks and symbols into internal machine language and vice versa. The coder or operator inserts a series of numbers and/or alphabetic information into an input device, which results in bits of information being stored in memory. Similarly, information in memory is converted by an output device to symbols which the user can interpret directly.

Figure 8-1 is a simplified version of the peripheral equipment central computer relationship.

Figure 8-1.

The computer INPUT section is capable of accepting data in a variety of forms and converting it to the standard format that is used by the central processing unit (CPU). The types of inputs which constitute the input section for different computers vary a great deal. Some of the more common types of input devices are punched card readers, magnetic tapes, paper tapes, and typewriters, CRT displays, teletypes, drums, discs, and flexowriters. The INPUT section obtains the information from punched cards, or other media, and places it in CORE MEMORY. This information may be data required in the solution of a problem, or instructions that tell the computer what to do. In effect, the input section provides one-way communications between you and the central computer. You can apply information to the input section, but no data will be returned to you from that section.

As might be expected, the OUTPUT section has the function of recording the results of computer processing. Actually, the output section is capable of recording anything located in core memory. The results may be presented in the form of punched cards or printed pages or CRT displays or drums, or disc, paper or magnetic tapes. The term recording, mentioned above is usually referred to as writing. Figure 8-2 is a detailed diagram of the exchange of information between the I/O system and the internal computer.

Figure 8-2. Exchange of Information Between the I/O System and the Internal Computer

Note that some devices have only output capability, some have only input capability and while others have the capability for inputting or outputting information.

I/O DEVICES

The programs which make up a computer-based system, as well as the data which the programs use must be readily available to the operating elements of the computer. Vast amounts of storage are required by the programs that operate as a system. Therefore an entire system cannot be stored in core memory at one time. Only those programs needed at any given instant of time can be stored in core. The rest of the system must be stored external to the computer and called into core memory as needed.

Although many I/O storage devices are available, punched cards, drums and magnetic tapes are the most frequently used, and a major portion of any I/O programming will be concerned with these devices. Punched cards provide you with an easily modified card deck which can be visually examined for possible changes or errors in the program. Magnetic tape and drums, although not as handy or as alterable as the card decks, provide high-speed data transfers and minimizes the amount of computer time used in transferring data in and out of core memory. This chapter will discuss the I/O devices used to store data external to the computer and how they can be programmed using flow charts.

There are two basic kinds of I/O storage devices: magnetic and nonmagnetic. Magnetic I/O devices hold information in the form of magnetized particles which generally represent binary digits, e.g., magnetic tapes, drums, and discs. Non-magnetic I/O devices represent information in the form of printed characters, punched holes or light sensitive marks, e.g., punched cards and paper tape.

A storage device is nothing more than a place to save something. A library, warehouse, filing cabinets, or suit pockets are, after all, nothing more than storage devices. To use these storage devices, one needs to know only two things: what is stored and where it is stored. These same considerations apply for I/O storage devices. To make use of information stored in a data storage device, it is necessary to know the form in which the data is stored and the location at which it is stored.

Data are stored in a storage device as a bit, a character, or a word. A bit can be stored as a magnetized spot on a magnetic medium or it can be stored as a hole or a mark on a non-magnetic medium. Any medium that can store a bit can be used to store a character. A collection of characters or bits which has a fixed pre-determined length can be stored as a word on any medium which will accept a character. To complete this brief review of data-form relationship, it is necessary to mention fields, records, files, and blocks. A collection of bits which are to be considered as a logical unit, but whose length can vary, is known as a field. The next level in the hierachy is that of a record. A related set of words or fields combine to form a record. The length of a record may or may not be fixed. Records combine to form a file. A file is a collection of related sets of information 9records) which can be treated as a logical unit. The file and record concept of data stratification is particularly useful and important when the data are stored in a magnetic medium. A block of information, on the other hand, is a general term used to describe two or more words, fields, or files and is not particular to any specific device or medium.

Now that we know what can be stored, we should consider where it is stored and how it can be found. Data are stored by words or fields at an address. An address is the location of a given word or field. The address may be fixed location pre-determined by the manufacturer of the storage device, e.g., magnetic drums, and discs; or it may be a location whose position is relative to the data structure, e.g., magnetic tape. Data are READ from the I/O device by the read element of the read-write head and stored in memory. The read operation is an input operation performed by the read element of the read-write head. Data are also WRITTEN from memory onto a storage medium. This operation is known as an output operation and is performed by the write element of the read-write head. When a word is read from a storage device and the act of reading destroys the contents of the address, the read operation is known as destructive read-out. Generally, for I/O devices the read operation is non-destructive. When a word is written or stored at an address, the original or previous contents of the address are destroyed.

The flow chart symbol for indicating any I/O operation is shown in Figure 8-3.

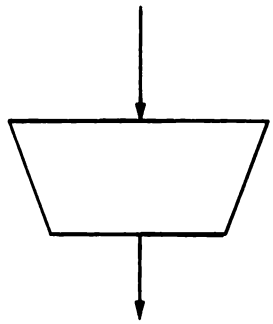


Figure 8-3.

This I/O*symbol has one input and one output flow line. When an I/O symbol appears in a flow chart it means an I/O transfer is occurring, i.e., information is being transferred from memory to the card punch, or drum, or tape drive (TD) or to memory from a drum, card reader, or tape. The text inside the symbol describes the specific I/O operation.

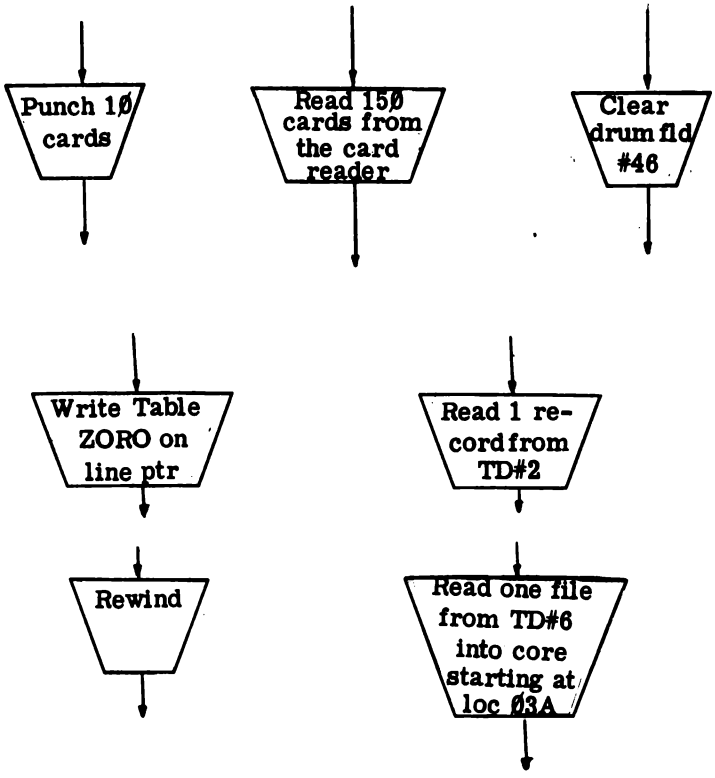


Figure 8-4.

The following is an example of a simplified flow using I/O programming.

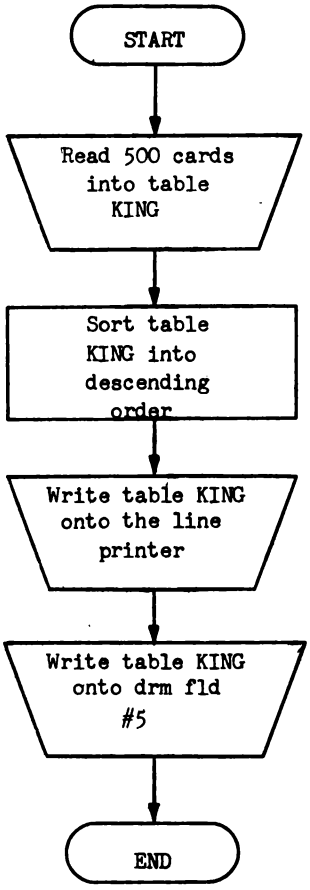


Figure 8-5.

You will notice in the preceeding example that the program was solved using a MACRO flow. By this time you should be able to look at the function box that calls for sorting KING and visualize the micro flow that would be necessary at that point. The following examples of I/O operations are macro flows but they give special attention to some problems encountered in I/O programming. Some special considerations for I/O programming involve the following:

1. Testing to see if the I/O device is ready to operate.
2. Determining completion of the I/O operation.
3. Positioning of tape drives before operating them.

The next example is a more detailed flow that gives attention to specific I/O limitations.

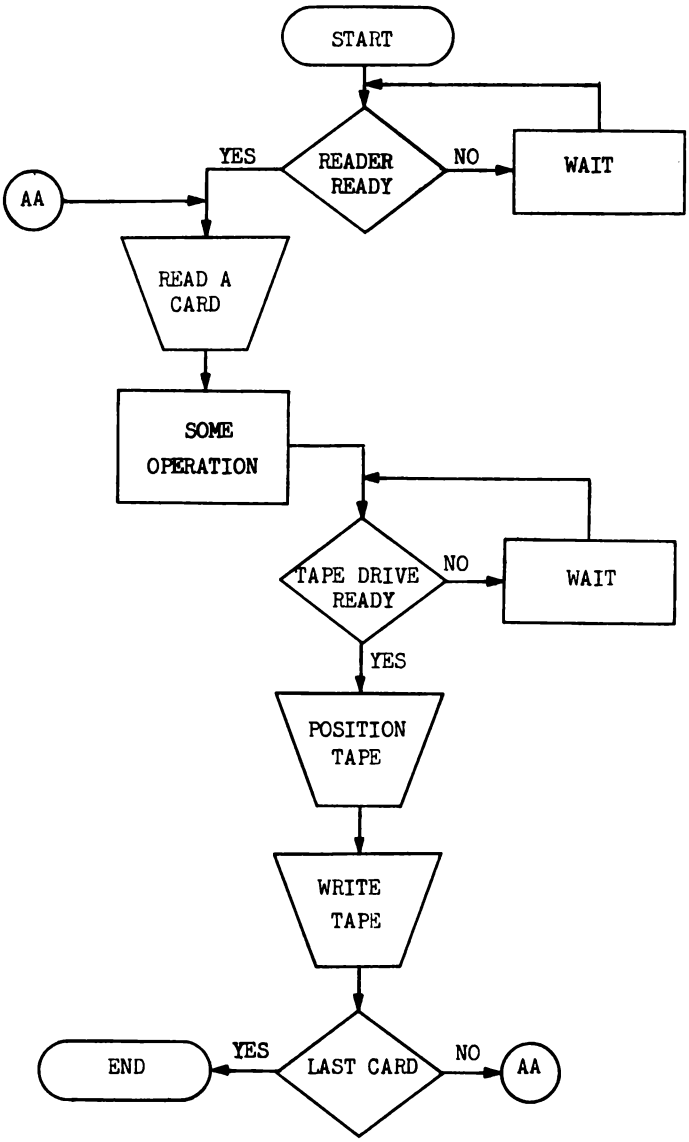


Figure 8-6.

The End of File (EOF) check is made with tape operations to determine the end of the tape read operation. The check for reader ready condition is not made in the following examples. Keep in mind, however, that this check is made by the hardware. Usually if an I/O device is not ready (turned on), the computer will alert the operator to this condition. The hardware cannot, however, detect an improperly positioned tape.

The following examples give some insight into the problems involved with specific I/O functions. The "initialize software" step includes steps 1 and 2 as listed for Figure 8-6.

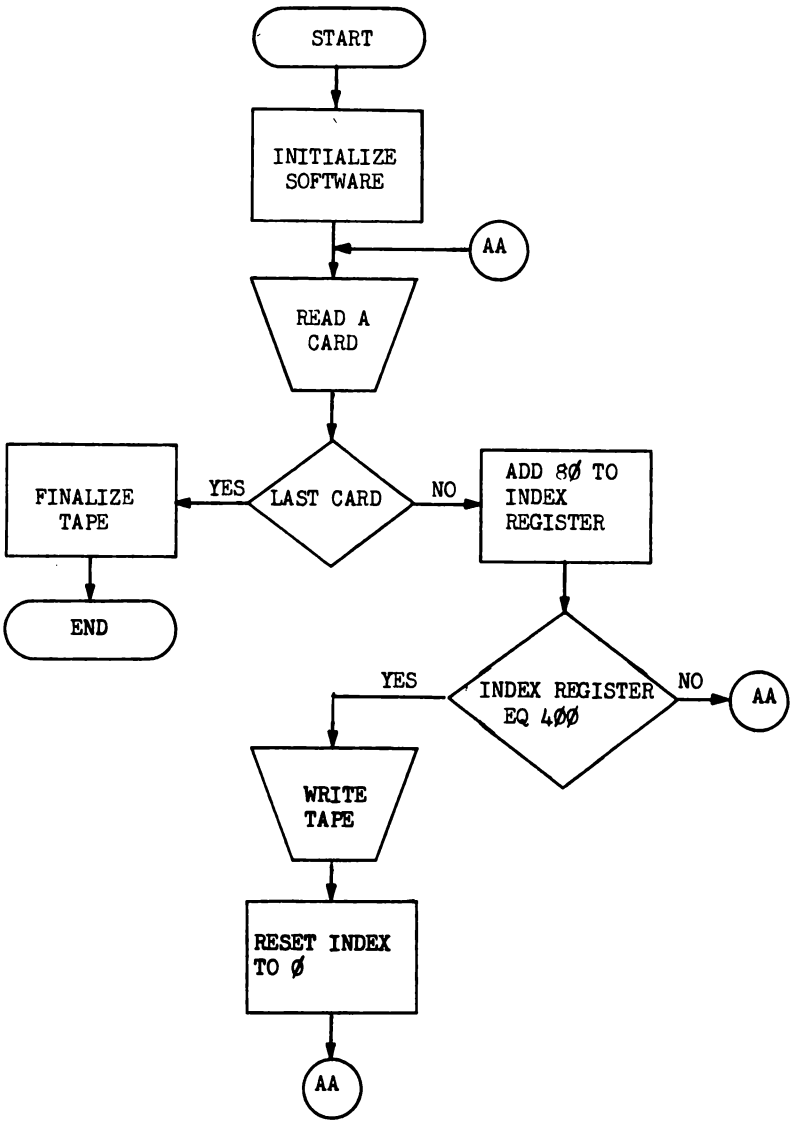


Figure 8-7.

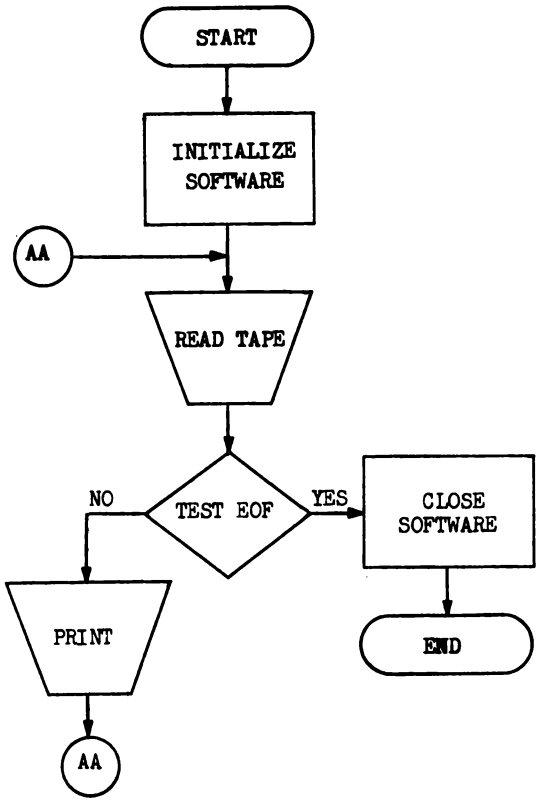


Figure 8-8. Tape to Printer

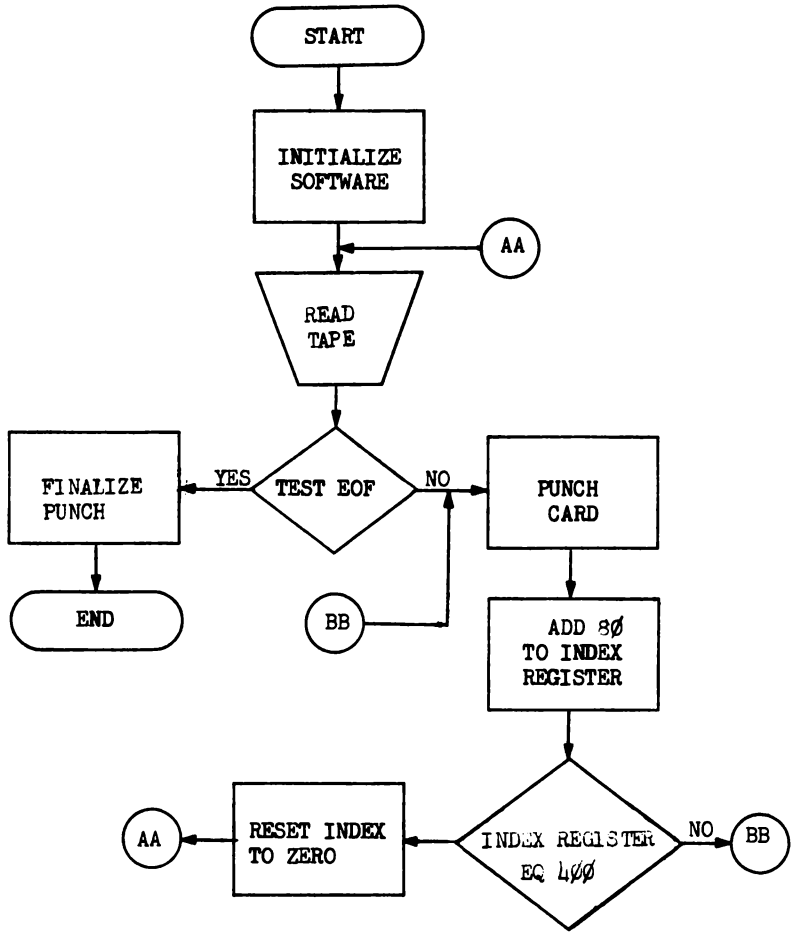


Figure 8-9. Tape to Card

The following flow includes provision for error detection and error halt. It introduces the necessity of housekeeping to insure that no extraneous data are present to cause the operation to function improperly. Note also that this example specifies which tape drive is used. The End of File (EOF) check is made. The determination of the flow function is left to the student. This flow is generally representative of a finished flow that would be sent to a coder for coding.

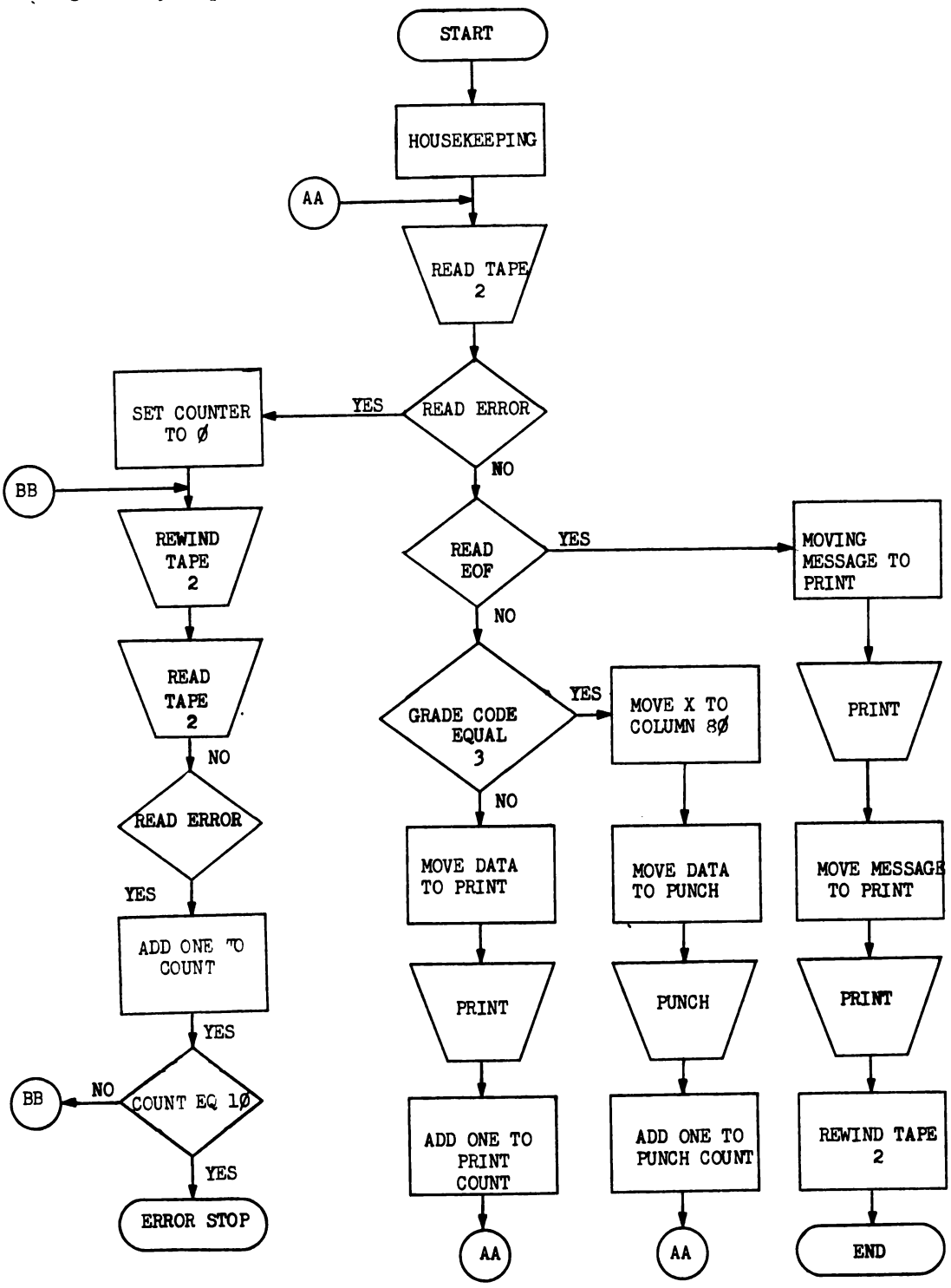


Figure 8-10. MACRO I/O

SUMMARY

Due to the limited internal storage available only those programs needed at any given instant of time are stored in core memory. The majority of data for a computer system is stored on cards, drums, tapes, etc. Therefore, programs and data must be constantly read into core and operated and/or used and then written back out onto external storage. Learning to use and program the I/O system effectively is a major part of a programmer's job.