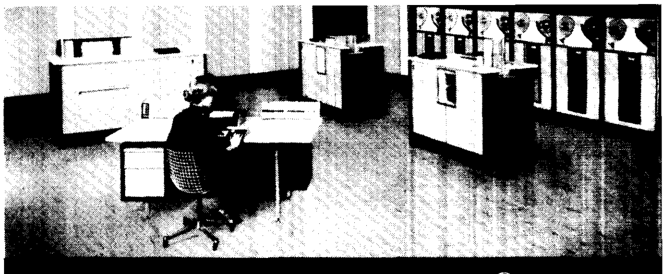


UNIVAC® III

GENERAL REFERENCE MANUAL

UTMOST

This manual is published by the UNIVAC® Division in loose leaf format as a rapid and complete means of keeping recipients apprised of UNIVAC Systems developments. The UNIVAC Division will issue updating packages, utilizing primarily a page-for-page or unit replacement technique. Such issuance will provide notification of hardware and/or software changes and refinements. The UNIVAC Division reserves the right to make such additions, corrections, and/or deletions as, in the judgment of the UNIVAC Division, are required by the development of its respective Systems.

PRINTED IN U.S.A.

December 3, 1963

UNIVAC III
UTMOST GENERAL REFERENCE MANUAL UP 3853

UPDATING PACKAGE A

SECTION	ADD NEW PAGES
Section 6	1 thru 7
Section 7	1 thru 4
Section 8	1 thru 39

The first addition to the "UTMOST Composite Manual," UP 3853 is Section 6 - "Variable Connectors," and suggestions on the use of UNIVAC III System for variable connectors are contained herein.

Since many programs involve looking up information in the memory, Section 7 - "Table Lookup" covers possible table lookup techniques. As an introduction to this subject, an example table lookup illustration is given.

Section 8 - "UTMOST," details the advanced features of the UTMOST Assembler. Of special note is the portion on the assembler directive PROC (pages 8-24 to 8-39), specifying how to construct and reference a procedure.

Place these pages in the manual in sequence of section numbers. For reference purposes, place this page directly after INDEX, until revised INDEX is issued.

UNIVAC III
UTMOST General Reference Manual, UP-3853

March 9, 1964

UPDATING PACKAGE "B"

The attached 79 pages contain additions to "UTMOST General Reference Manual", UP-3853.

This updating package should be utilized in the following manner:

	Title	Pages	Insert After TAB Labeled
SECTION 9	Tape File Handling	1 - 55	ITEM LEVEL TAPE FILE HANDLING
SECTION 13	Symbionts	1 - 24	SYMBIONTS

Please notice that no destruction is necessary with this updating package.

UNIVAC III
UTMOST General References Manual, UP3853

September 29, 1964

Updating Package "C"

This bulletin announces the release and availability of Updating Package "C" for the UTMOST General Reference Manual, UP3853, 49 pages. The pages should be utilized in the following manner:

	Destroy Former Pages Numbered	File New Pages Numbered
Section 17-F	N.A.	1 - 49 *

* These pages should be filed after the tab labeled INPUT/OUTPUT.

UNIVAC III
UTMOST General Reference Manual, UP 3853

December 10, 1964

UPDATING PACKAGE "D"

The attached material represents additions and changes for the UNIVAC III UTMOST General Reference Manual, UP 3853, and should be utilized in the following manner:

	DESTROY FORMER PAGES NUMBERED	FILE NEW PAGES NUMBERED
Table of Contents	N.A.	1 - 13
Section 2-A	21 and 22	21 and 22 Rev.1
Section 5	5 and 6	5 and 6 Rev.1
Appendix E	N.A.	1 - 3

CONTENTS

APPENDICES

TABLES AND ILLUSTRATIONS

FIGURES TABLES

2A. INTRODUCTION TO COMPUTER DATA PROCESSING

A. THE ELEMENTS OF DATA PROCESSING

In most data-processing, there is a set of data that is altered either infrequently or else in a known and invariable way. This type is referred to as *master data*. Names, addresses, badge numbers, pay rates, year-to-date gross, year-to-date withholding tax, and quarter-to-date social security tax are examples of master data representing the payroll area; stock numbers, descriptions, on-hand amounts, and unit of measure represent the inventory-control area.

Beyond the master data, there is another type of data to be fed into any data-processing system; this information differs in that its incidence is essentially random and unpredictable. This type is called *transaction data*. Hours worked, quantities shipped, and amounts invoiced are examples from, respectively, the areas of payroll, accounts receivable, and accounts payable.

Processing consists basically of applying the items of transaction data, either singly, as they come up, or in cumulative batches, to update the master data.

On the other hand, processing may also be constituted by information periodically being produced from the master data alone. An example, in the accounts-receivable area, is the production of a monthly statement.

There is one other major item in the general data-processing operation, the report. In essence, the report is a by-product of the processing operation in that it reflects in summary or other form updating of the master data, the latter being the chief function of the data-processing system. However, for most purposes, the report can be considered the end product and therefore the most important of the four elements. It abstracts and highlights critical aspects of the business picture that judicious processing of transaction and master data uncover, and it is looked to by management for necessary information for decisions in production, sales, purchasing, finance, and all other phases of business.

The schematic in Figure 2-1 relates the four basic elements in the general data-processing operation.

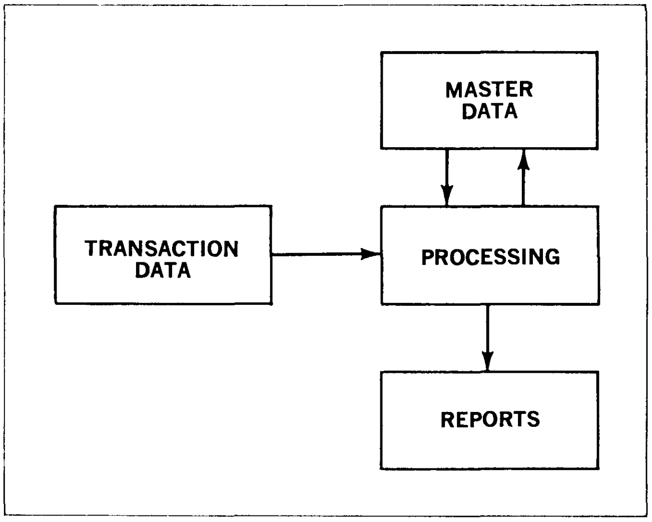
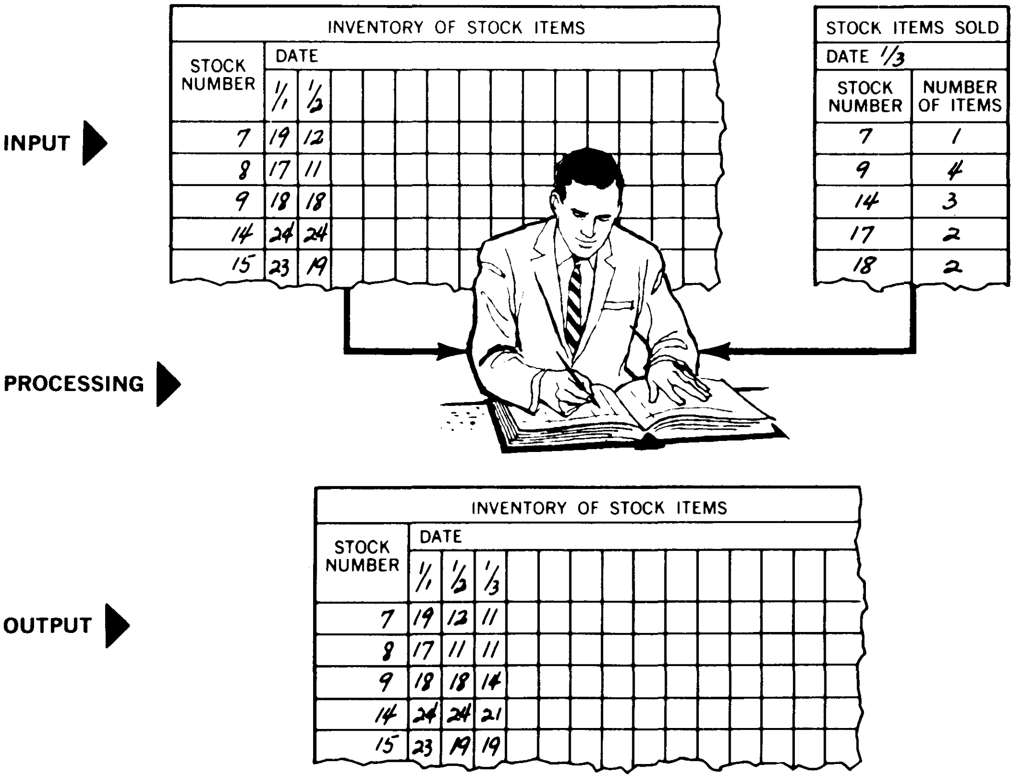


Figure 2-1. The General Data-Processing Operation

To further investigate the elements of a data-processing operation, examine the steps in the solution of a simplified processing application. Consider a company that keeps a record of its stock in a ledger. Each day a clerk is supplied with a sales form. On the basis of the form, the clerk brings the inventory up-to-date by writing a new column in the ledger. A representation of this data-processing operation is shown in Figure 2-2.



INVENTORY OF STOCK ITEMS

STOCK NUMBER	DATE 1/1	1/2
7	19	12
8	17	11
9	18	18
14	24	24
15	23	19

STOCK ITEMS SOLD

DATE 1/3

STOCK NUMBER	NUMBER OF ITEMS
7	1
9	4
14	3
17	2
18	2

INVENTORY OF STOCK ITEMS

STOCK NUMBER	DATE 1/1	1/2	1/3
7	19	12	11
8	17	11	11
9	18	18	14
14	24	24	21
15	23	19	19

Figure 2-2. A Data-Processing Operation

As indicated in Figure 2-2, this data processing operation breaks down into three broad parts:

- INPUT: the information to be processed.
- OUTPUT: the information produced by the processing.
- PROCESSING: the operations required to produce the output from the input.

To do the processing represented in Figure 2-2, the clerk must go through a certain sequence of steps. One possible sequence is represented in Figure 2-3.

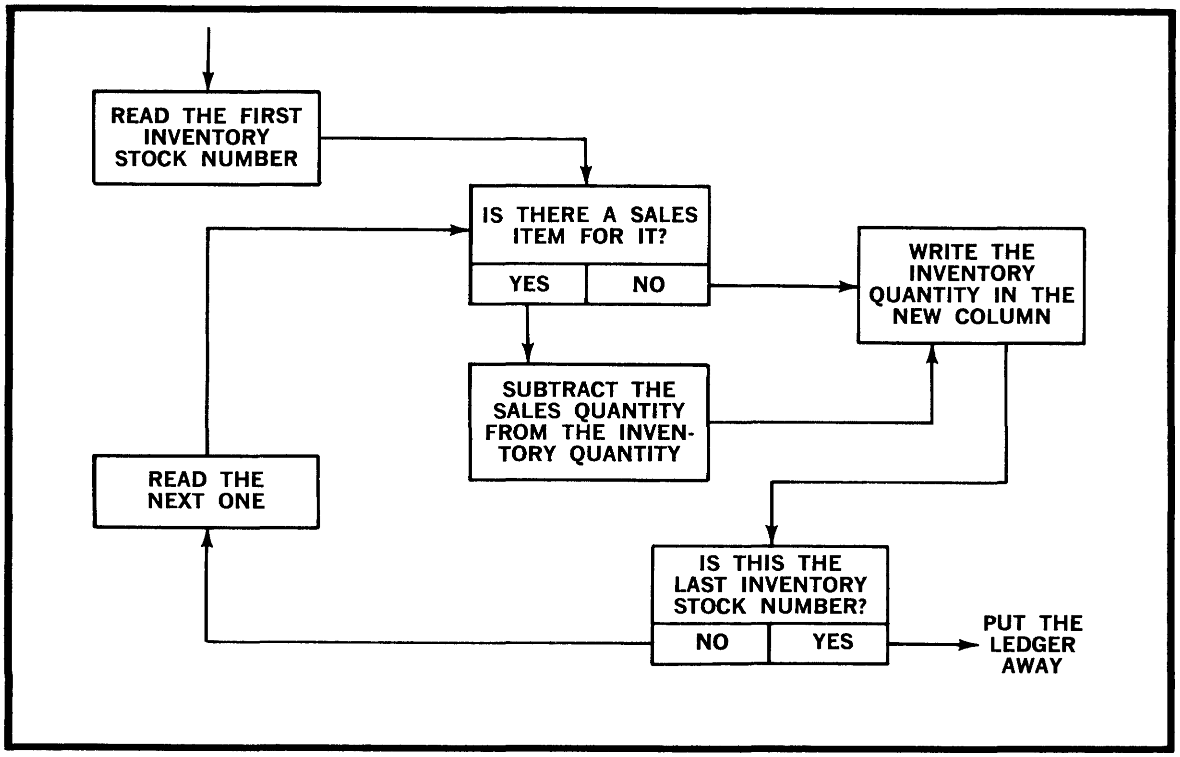

READ THE FIRST INVENTORY STOCK NUMBER
IS THERE A SALES ITEM FOR IT?
YES
NO
WRITE THE INVENTORY QUANTITY IN THE NEW COLUMN
SUBTRACT THE SALES QUANTITY FROM THE INVENTORY QUANTITY
READ THE NEXT ONE
IS THIS THE LAST INVENTORY STOCK NUMBER?
NO
YES
PUT THE LEDGER AWAY

Figure 2-3. The Sequence of Steps in the Data-Processing Operation

To do the steps shown in this Figure:

1. *The clerk must be able to do arithmetic (e.g., he must be able to subtract the sales quantity from the inventory quantity).*

2. *He must be able to make logical decisions (e.g., he must be able to determine whether or not there is a sales item for a given product).*

3. *He must be able to remember information (e.g., after he subtracts the sales quantity from the inventory quantity he must remember the difference at least until he writes it in the ledger).*

4. *Hemust do the steps in the sequence shown or do something logically equivalent to this sequence of steps.*

These four elements of processing are referred to, respectively, as:

1. *Arithmetic.*
2. *Logical Decision.*
3. *Memory or Storage.*
4. *Control.*

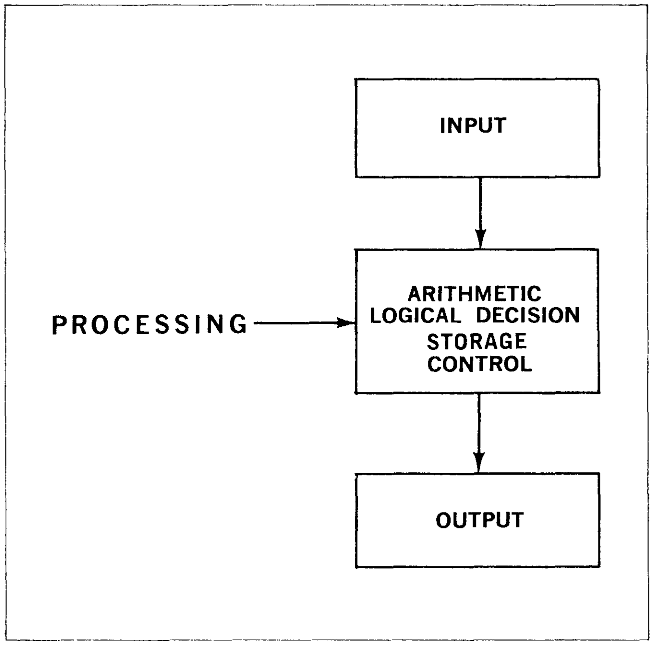


Figure 2-4. The Elements of a Data-Processing Operation

Experience has determined that to do the general data-processing operation, input, arithmetic logical decision, storage, control, and output are required. These six elements are shown in their logical relationship in Figure 2-4.

1. Manual Data-Processing

   The above example is a simplification. An actual inventory application is more complex. Moreover, even in the simplified form presented above, certain basic steps are left out. The question may be asked: How does the sales form originate? When a sale is made, a sales slip describing the commodity sold and the units of that commodity sold is prepared. Such a slip is prepared for each sale made during a day. At the end of the day, the clerk receives from the sales organization, not the sales form, but a bundle of sales slips, each representing a transaction. (For purposes of simplicity, assume that each transaction, and consequently, each sales slip, involves only one commodity). To prepare the sales form from the package of sales slips, the clerk has to first classify the sales slips by commodity, and at the same time, or as a separate operation, sort them into stock number order to put them in the same order as the commodities are listed in the inventory ledger. The clerk is then in a position to summarize the sales slips by commodity, and, as a final preparatory operation, prepare the sales form. With the resulting sales form, it is possible for the clerk to carry out the updating procedure described in the previous section.

For an operation of low enough volume, the approach described above is adequate. It is possible for one clerk to keep the inventory records for the simplified inventory application up to date. However, as the volume of the company's operations increases, the burden of keeping the inventory records up to date will become too heavy for one clerk. It will be necessary to add other clerks to handle the increased work load. With the advent of a number of people to maintain the inventory records, management may adopt the procedure of breaking the inventory maintenance down into a number of steps and of assigning one person to each one of the steps. Thus, one clerk might *read* the sales slips and *sort* them into the same order the inventory commodities are listed in the inventory ledger. Another clerk might then accept these sorted sales slips from the first clerk and record the sales on the sales form at the same time as he *summarizes* the sales slips by commodity. A third clerk might *subtract* the entries on the sales form from the balances on hand and record the differences on the sales form. Finally, a fourth clerk might *record* these new balances in the ledger. A schematic of this procedure is shown in Figure 2-5.

The approach shown in Figure 2-5 consists of breaking down the job into a number of simple steps. These steps fall into categories that constitute the functions of data-processing.

- *READING*
- *SORTING*
- *CALCULATING*
- *DECISION MAKING*
- *RECORDING*

When a job is simplified by breaking it down into a series of steps, the data to be processed are circulated through this series. Each step is the responsibility of a single person who performs the step repeatedly on the continuing flow of data.

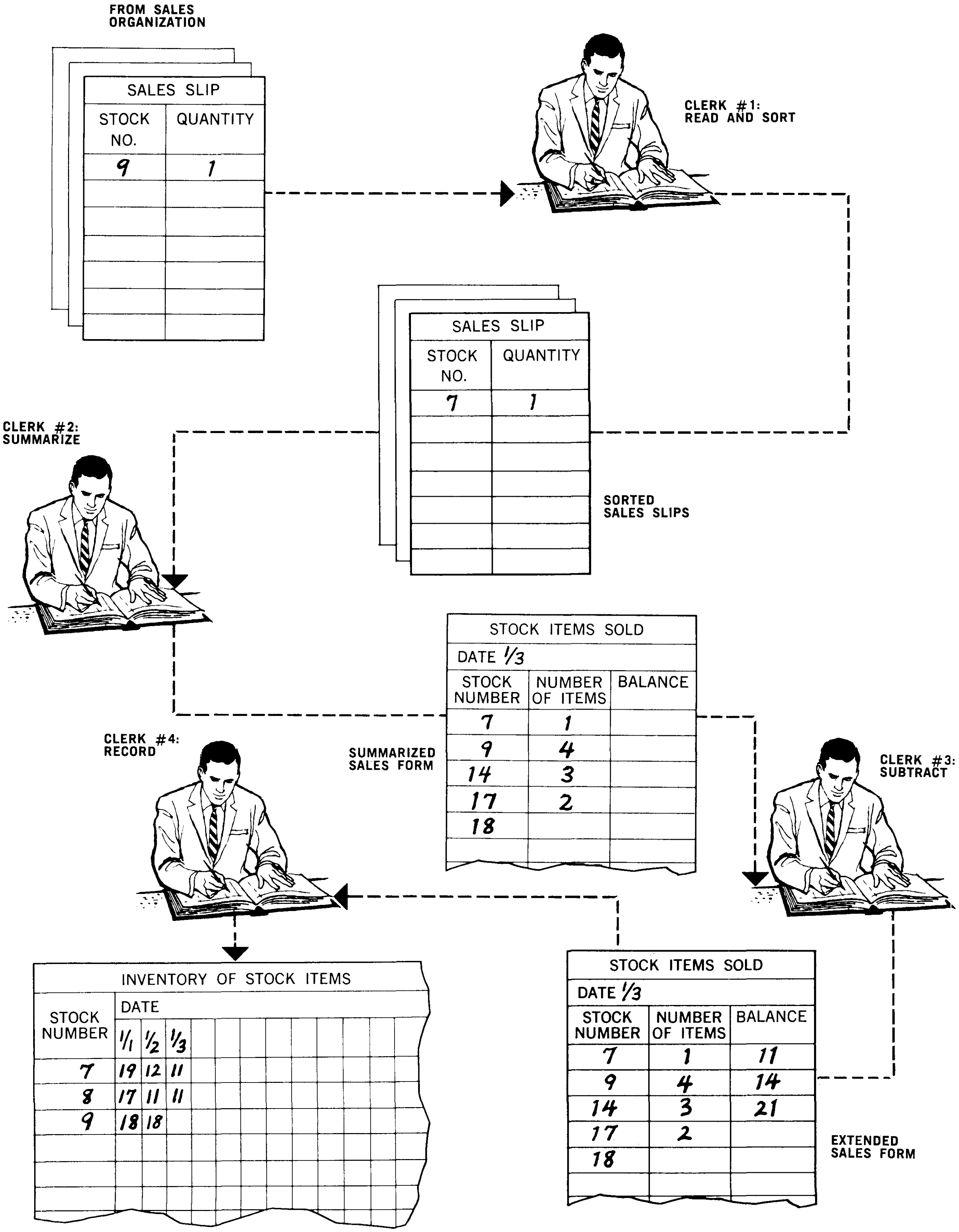
The approach just described is characteristic of manual data processing systems. Analyzing a job and dividing it into a series of steps is the first step in the development of a data-processing system.

2. Key-Driven Devices

Some of the functions of data-processing are mechanized in the typewriter and the adding machine. Each of these office machines performs one of the basic functions. Thus, the typewriter records, and the adding machine calculates. For example, in the simplified inventory application depicted in Figure 2-5, clerks two and three might use adding machines to summarize and subtract. Clerk number four might use a typewriter to record the updated inventory.

Since these machines perform only one data processing function, they are ''building block'' machines. They can fit into the pattern of analyzing a job into a series of tasks with no loss of flexibility. Their advantage lies in the fact that they increase both the speed and the accuracy of their operators.

The mechanization of data-processing functions is the second step in the development of processing systems.



SALES SLIP

STOCK NO.	QUANTITY
9	1

SALES SLIP

STOCK NO.	QUANTITY
7	1

STOCK ITEMS SOLD

DATE 1/3

STOCK NUMBER	NUMBER OF ITEMS	BALANCE
7	1	
9	4	
14	3	
17	2	
18		

INVENTORY OF STOCK ITEMS

STOCK NUMBER	DATE 1/1	1/2	1/3
7	19	12	11
8	17	11	11
9	18	18	

STOCK ITEMS SOLD

DATE 1/3

STOCK NUMBER	NUMBER OF ITEMS	BALANCE
7	1	11
9	4	14
14	3	21
17	2	
18		

Figure 2-5. Work Simplification

3. Punched Card Machines

With the keyboard-operated typewriter or adding machine, the operator must act as an interpreter, taking the results produced by one machine and transferring them, through the keyboard, to the other. For example, in the simplified inventory application shown in Figure 2-5 clerk number four must take the results produced on the adding machine by clerk number three and enter these figures on the keyboard of her typewriter to record them on the inventory records. The adding machine produces typed numbers on a paper tape but the typewriter only "understands" pressure on its keys. Hence, the clerk not only carries the messages from the adding machine to the typewriter, he also translates from one language to another.

It is uneconomical for a person to do a substantial amount of this transferring, translating and copying when it can be done mechanically with more speed and accuracy. One solution to this problem is the punched-card machine, which approaches this problem of communications in the following way. The medium of communication in this type of system is a card on which one column is equivalent to one character of information. Holes punched in combinations of rows in a column represent these characters in coded form in the same way as the dots and dashes in Morse code represent characters. Figure 2-6 shows a card with some codes punched in it.

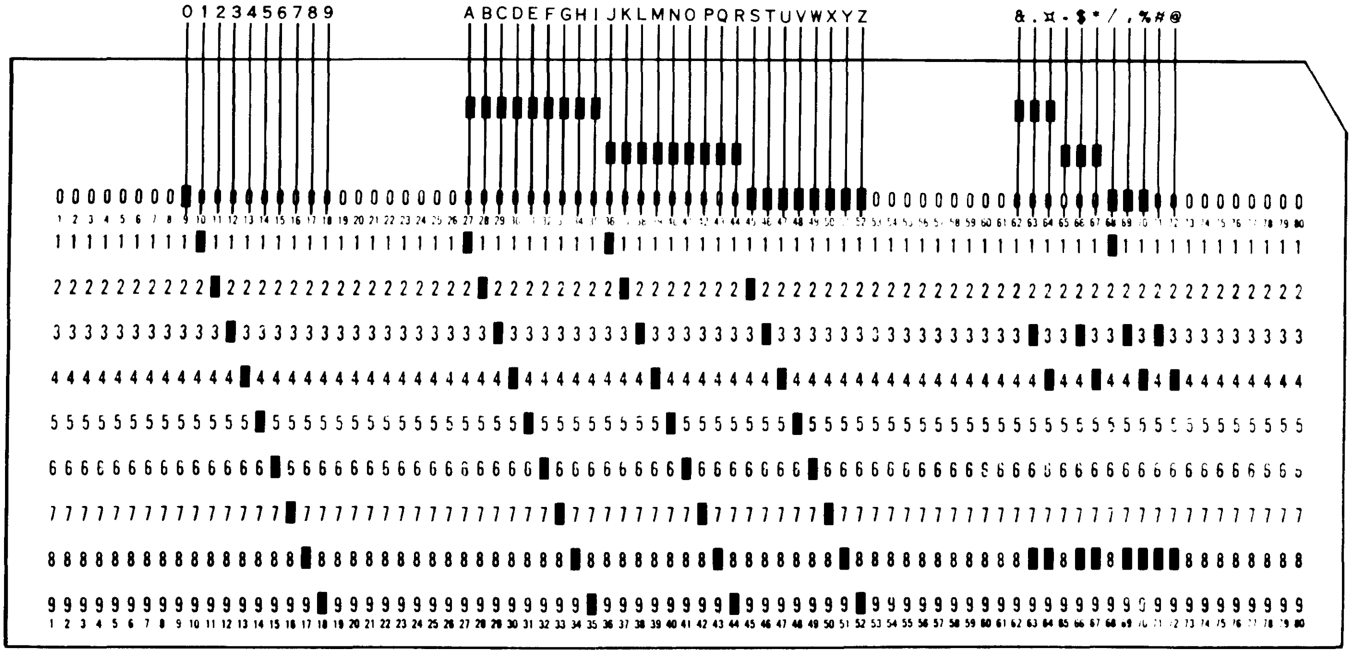


Figure 2-6. A Punched Card

By means of pins which make mechanical contact, a beam of light which activates a photoelectric cell, or brushes that make electrical contact through the holes punched in the card, punched-card machines can sense and "understand" the information punched in the card.

Thus, the machines "communicate" with each other through the medium of the holes in the punched card. All that is necessary is that, initially, all data to be processed be punched into cards in the common or "machine language" code. These cards are then used by an array of specialized punched-card machines: sorters, collators, card reproducers, calculators, punches, and tabulators. Each of these machines performs one of the data-processing functions. As a consequence, punched-card machines are also "building block" machines, able to incorporate a complex of operations formerly dealt with by manual means, and can be arranged in many ways to perform data-processing operations.

For example, the simplified inventory application might be done on punched-card equipment in the following way. Initially, the information in the inventory ledger has to be converted to punched-card form. This operation is executed on a key punch. One card is produced for each commodity in the inventory. Each such card contains, in coded form, the stock number of the commodity that this card represents and the current inventory balance for this commodity. The cards in this deck are kept in stock number order, the same way the stock numbers were listed in the ledger.

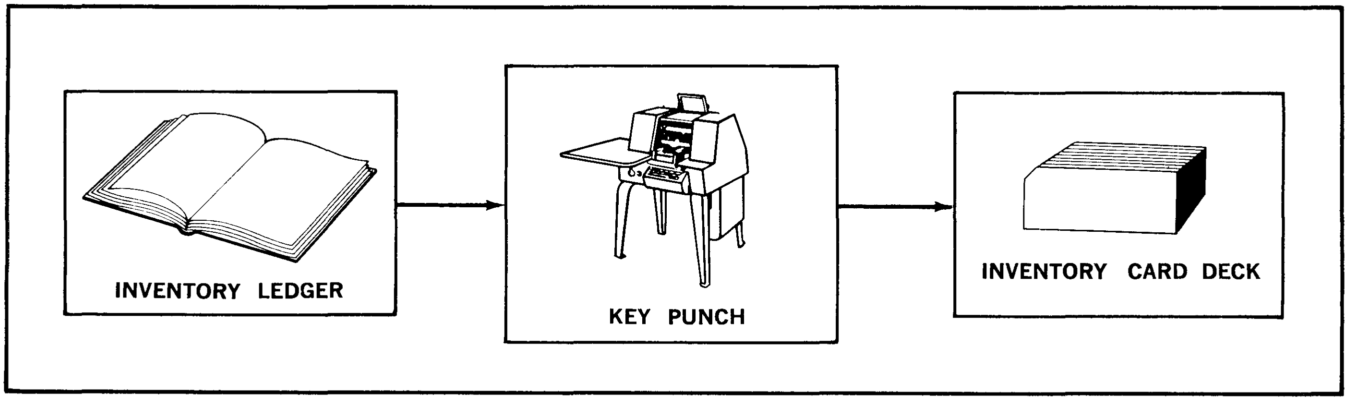


Figure 2-7. Punching the Inventory File into Cards

Once prepared, this inventory card deck never has to be prepared again, because the punched-card system maintains the deck in much the same way as the clerk maintained the ledger.

When the sales slips are received from the sales organization, they are punched into cards, one card for each sales slip, on the key punch. Each card in the sales deck now contains a stock number and a sales quantity. Another piece of card equipment called a sorter is then used to sort the cards into stock number order.

Now a piece of equipment called a collator is used. The collator capable of sensing or "reading" punched-cards has two input magazines. The inventory card deck is placed in one of these magazines. The sales deck is placed in the other. The collator also has a number of output stackers in which it stores cards which it has read. For the operation at hand, the collator is used to match the stock number of the inventory card in the bottom of the inventory deck magazine with the stock number of the sales card in the bottom of the sales magazine. If the stock numbers do not match, the inventory card is "not active" and is placed in one stacker. If the numbers match, the inventory card is active and it, together with the sales card and all sales cards following having the same stock number, are placed in another ("active") stacker. This operation of the collator is shown schematically in Figure 2-8.

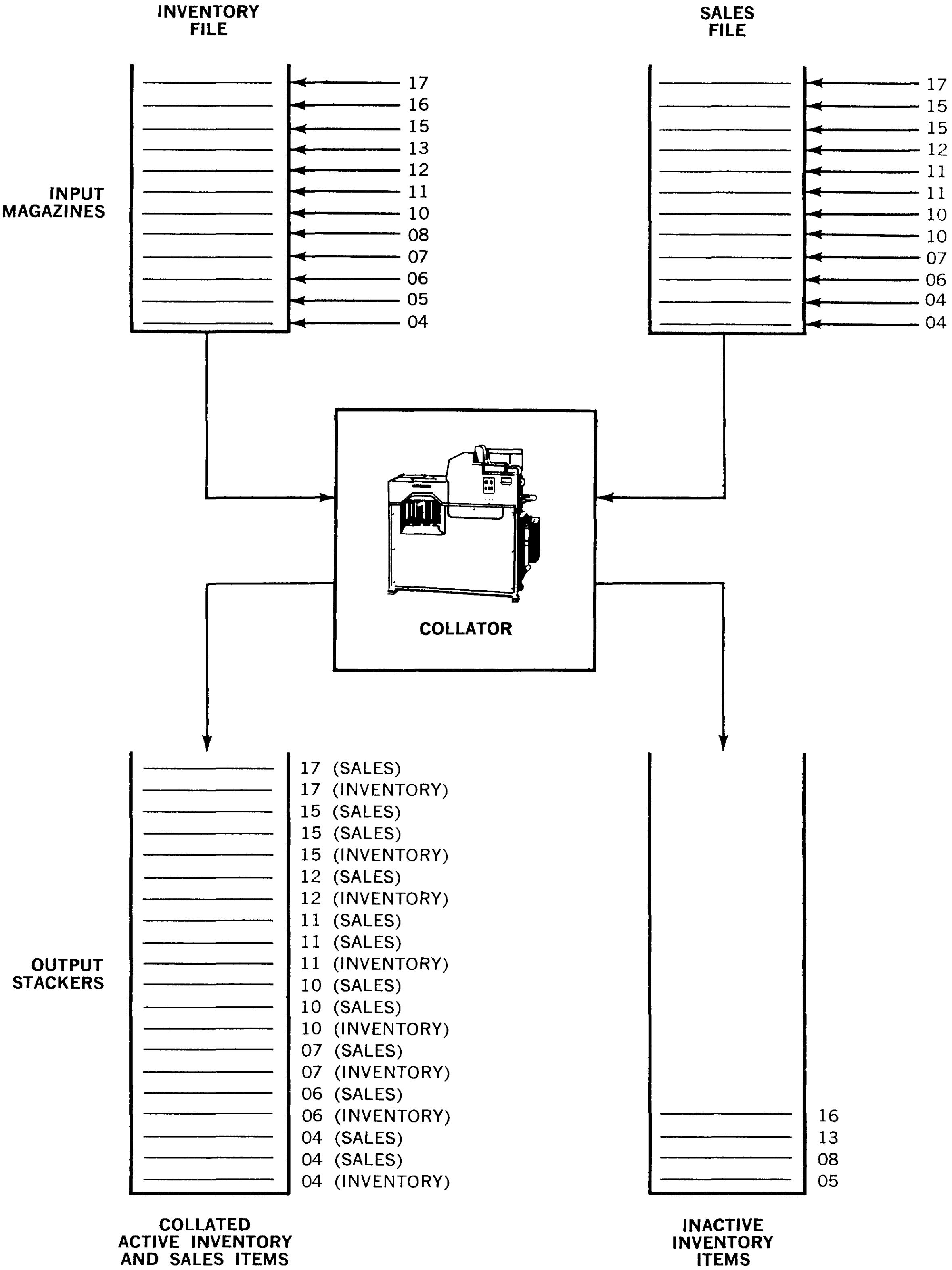


Figure 2-8. Collation of Inventory and Sales Items

At the completion of the collation operation, the collated active inventory and sales cards are run through a tabulator, which subtracts the sales quantities in the sales cards from the on-hand quantities of the associated inventory cards.

Attached to the tabulator is an automatic card punch. For every active inventory card read into the tabulator, the punch produces from a blank card a new inventory card with the same stock number and the new on-hand amount as supplied by the tabulator.

Finally, the collator is used once more. This time the updated inventory cards are placed in one input magazine and the previously inactive inventory cards are placed in the other. For this operation, the collator compares the stock numbers of the two cards in the bottom of the two magazines and places the one with the lower stock number in an output stacker. The collator then repeats this process over and over until all the cards are in stock number sequence in the stacker. This operation creates the updated inventory deck, which can be used as the inventory deck for the next day's operation.

Holes punched in cards are not conveniently interpreted by persons using them. A machine called an interpreter performs this function for the convenience of operating personnel. In addition, it is necessary in a punched-card installation to have some printing facility for preparing reports for management. This printing facility is located in the tabulator. In the case of the simplified inventory application, any necessary reports can be printed by the tabulator at the same time that it is updating the inventory balances.

A schematic of this system is shown in Figure 2-9.

The punched-card also serves as a storage medium for information. In terms of the simplified inventory, this fact means that the inventory ledger has been replaced by the inventory card deck. The result of the communications and storage aspects of the punched-card is that data-processing becomes a materials handling job. The punched-cards are transferred from machine to machine and can be stored indefinitely for future use.

4. Punched Paper Tape

Punched paper tape is another form of ''machine language'' medium. The approach here is the same as in punched-cards: characters are represented in coded form, the code consisting of various combinations of punched holes. There are three basic differences between punched-cards and paper tape. First, the medium in which the punching is done is a paper tape of variable length rather then a fixed sized card. Second, card equipment handles information on cards a card at a time; paper tape equipment handles information a character at a time. Third, punched-card code is different from paper tape code.

Paper tape is used to a great extent in communications, the message being sent over wire and arriving in the form of a punched paper tape. However, the use of paper tape is not restricted to the field of communications. Paper tape punches and readers can be attached to conventional office equipment such as typewriters or accounting machines with the result that information entered on the keyboard of these machines can be taken off in the form of punched paper tape. This resulting tape can be read by the same or other equipment. This reading operation of an already prepared tape allows the processing of the information by the equipment without the necessity for re-entering the information on the equipment keyboard.

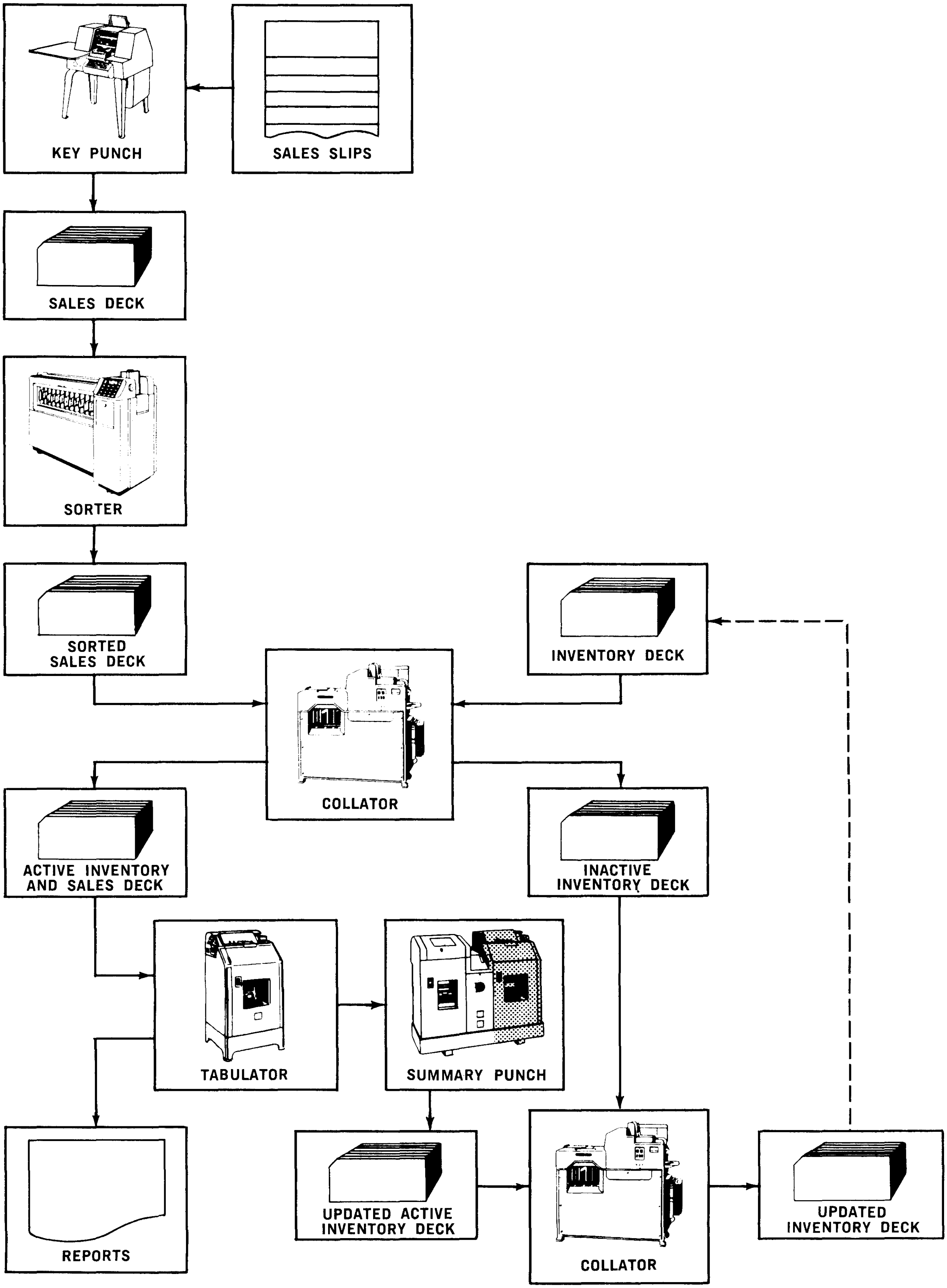


Figure 2-9. Punched Card Equipment

There are various types of equipment that read paper tape and produce punched cards and vice versa. Therefore, paper tape and punched-card equipment can be used cooperatively on the same batch of information without manually recording it in both paper tape and punched-card form. One recording in either form is sufficient. For example, in the simplified inventory application, if the sales organization were geographically widespread, the sales slip information might be sent over wire and arrive in the accounting office in the form of paper tape. This paper tape information can then be converted to punched-cards in which form it can enter the card system shown in Figure 2-9 at the point where the sales deck is sorted.

5. Magnetic Tape

Besides punched-cards and punched paper tape, a third type of bulk storage and communication medium is magnetic tape. Magnetic tape consists of a long strip of plastic material on which information is recorded in coded form. In this case, the code is a combination of magnetized spots rather than punchedholes. The data-processing equipment reads and writes the recorded information by means of tape handlers, each of which is similar to a household tape recorder. Equipment exists to convert information in magnetic tape form to or from either paper tape or punched card form.

The advantages of magnetic tape are that it allows (1) a denser packing of information than does either paper tape or punched-cards; (2) a higher rate of reading and recording. Consequently, more information can be stored in less space, and higher speed data-processing devices can be utilized.

6. Computers

A common language medium in the form of punched-cards and paper and magnetic tape is the third step in the development of data-processing systems. Except for data origination and the handling of bulk data (decks of cards and reels of tape), the human function in a common language data-processing system is reduced to following the right procedure. The following of such procedures is handled automatically by the computer, the latest step in the development of data-processing systems.

a. Real Time Computers

In general, computers are divided into two broad categories. The first is used to apply transaction data to the master file as the transaction data occurs. In the other type, the transaction data is batched over a period of time and is applied by the computer to the master file data in the resulting batches. The first type is known as a real-time computer; the second, as an offline computer.

To get some idea of the operation of a real time computer, consider the simplified inventory application. If the computer is to reduce the on hand quantity for any commodity at the time that the commodity is being sold and if any commodity in the line can be sold at any time, then the computer must have immediate access to the whole inventory record at all times. This need can be met by recording the inventory information on some type of ‘‘storage’’ device. One such device resembles a juke box, in which the information is recorded on the records and the computerhas the power to place a record arm on any part of any record that it desires. Another consists of a drum and has the information recorded in tracks around the surface of the drum. The computer has the ability to read information from or record information on any part of any track that it desires. Such devices are referred to as *mass storage* devices.

Secondly, to perform the operations required, there must be some mechanism to allow each transaction to be entered into the computer for processing as it occurs. This requirement can be met by some type of keyboard device that allows the salesman to send the necessary information to the computer at the same time as he is recording the sale for the customer.

Finally, since all inventory information is stored on the mass storage device and is not accessible to management, a printer must be available to the computer so that all required reports can be printed. The computer has the ability to post a transaction to the proper inventory record when it occurs and to select the proper information for reports. A schematic of the real-time computer system described above is shown in Figure 2-10.

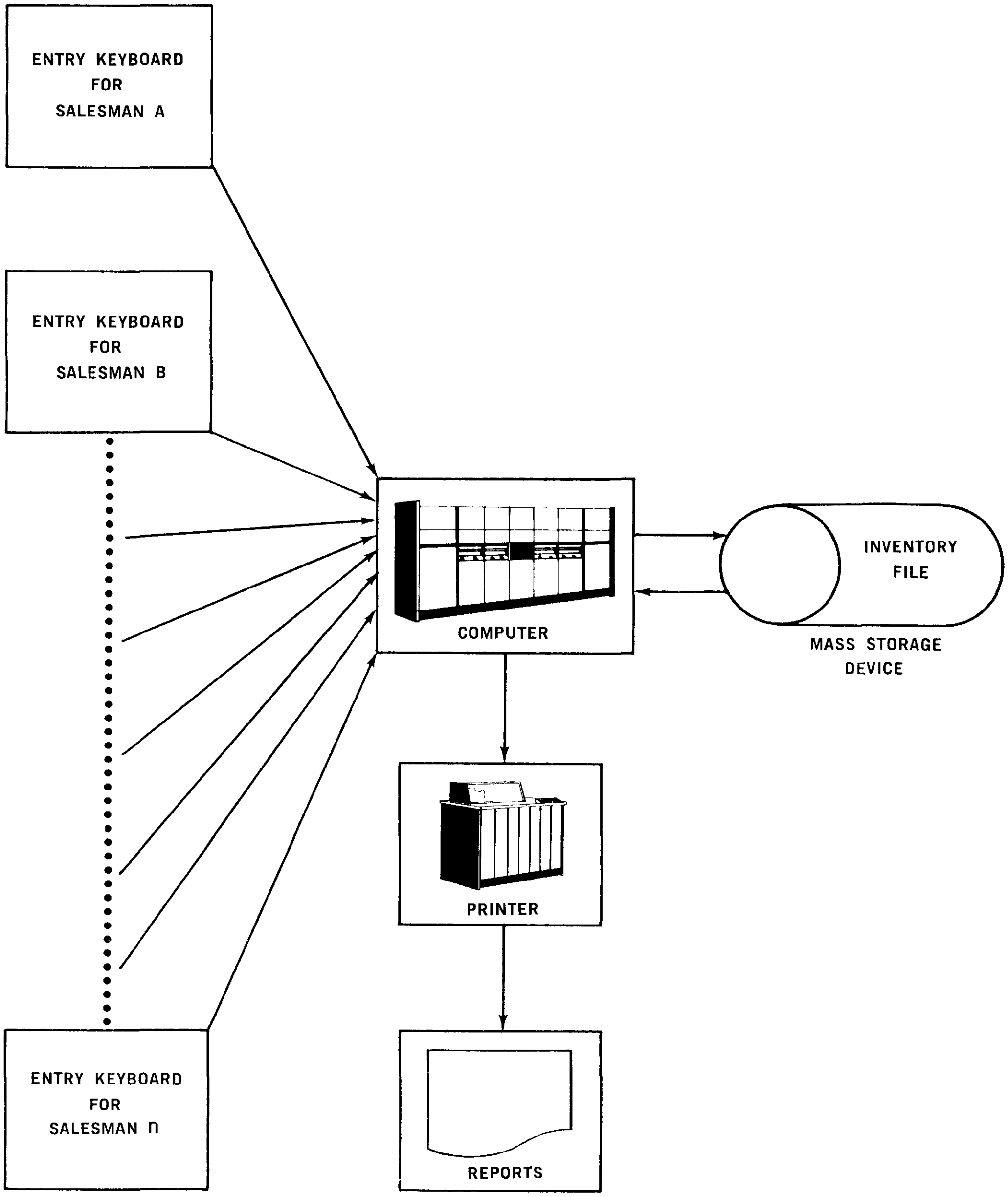

ENTRY KEYBOARD FOR SALESMAN A
ENTRY KEYBOARD FOR SALESMAN B
ENTRY KEYBOARD FOR SALESMAN n
COMPUTER
INVENTORY FILE
MASS STORAGE DEVICE
PRINTER
REPORTS

Figure 2-10. Realtime Computer

The advantage of the real-time computer is that the application in which it is used involves master data that is up to date. For example, in the simplified inventory application, the inventory data reflects the current actual inventory situation. However, if the computer is to apply transaction data to the master data randomly as the transaction data occurs, all of the master data must always be stored on the mass storage device, and the computer must constantly be available for the updating calculations without interference from other uses of the computer. For example, in the inventory application, the inventory data must always be stored on the mass storage device. This fact means that the mass storage device can be used only for those data-processing applications for which the master data permanently stored on it is applicable. Mass storage devices are currently not inexpensive and, as of now, the applications that can justify such a device on the merits of one application are relatively few.

b. Off-Line Computers

In the off-line computer, the master data is stored not on a mass storage device, but on magnetic tape. Consequently, addition of more applications with various master files to the computer data-processing system is a matter of recording the master files involved. A consequence of this approach is that no master file is available to the computer in its entirety. Therefore, transaction data cannot be applied as it occurs. Instead, transaction data is batched over a period of time and is applied by the computer to the master data in the resulting batches on a cyclical basis. This approach results in the fact that no master file is ever completely up to date. However, the speed and accuracy with which the computer operates allows a cyclical updating period of short duration. This short cycle results in master files, the timeliness and accuracy of which cannot be approached by any other kind of equipment. The off-line computer is the standard computer used in data-processing today.

As an example of an off-line computer operation, consider the simplified inventory application as it might be applied to an off-line computer.

When the computer is first introduced as the data processor, the inventory tape would have to be prepared in some way. For example, it might be punched into cards, the punched-card deck then being converted to tape as shown in Figure 2-11.

Once prepared, this inventory tape would never have to be prepared again, because the computer would maintain it in much the same way as the punched-card system maintained the information in punched-card form.

The company operation generates the sales form for the computer system the same way as before. However, before the computer system can use the information on the sales form, it must first be converted to tape in a manner similar to the way in which the inventory tape is initially prepared.

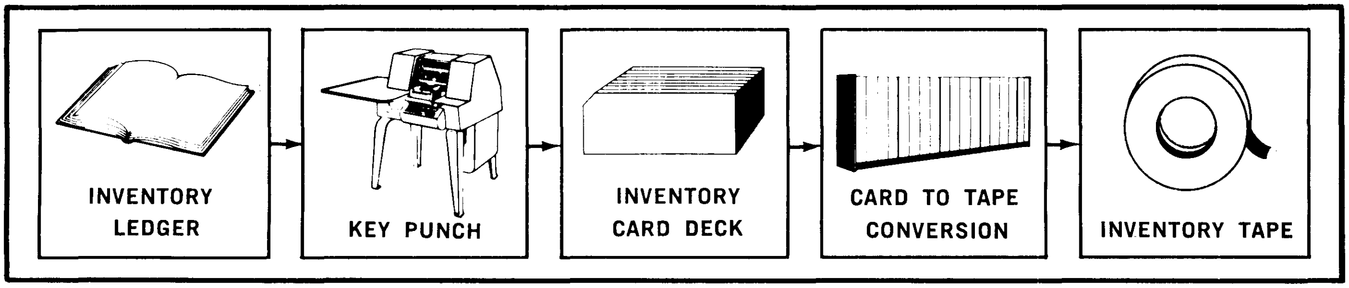


Figure 2-11. Converting the Inventory File to Magnetic Tape

The computer then reads this sales data from this sales tape by means of a tape handler (Figure 2-12). The computer sorts the sales data into stock number order and summarizes it. This sorted and summarized sales data is then written by the computer on a blank tape mounted on another tape handler.

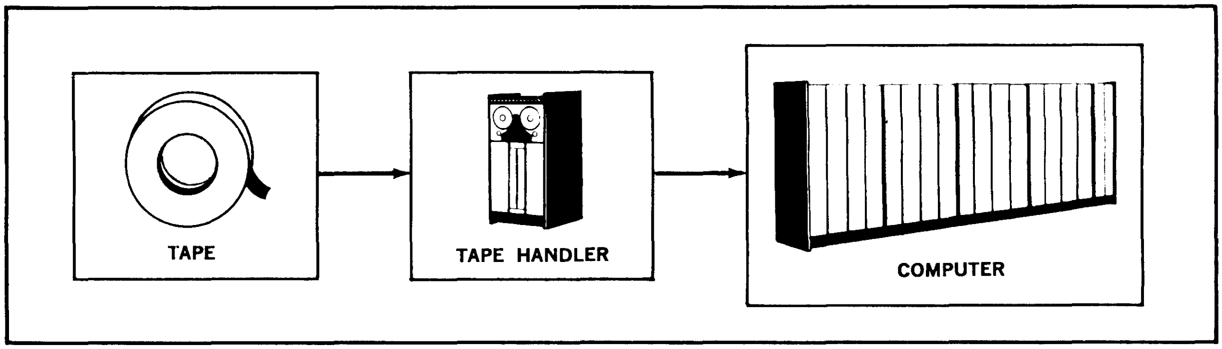


Figure 2-12. Reading the Information from a Tape into the Computer via a Tape Handler

The processing operations of arithmetic, logical decision, storage, and control, which are necessary to produce the updated inventory from the information given on the current inventory tape and on the sorted sales tape, are done by the computer. The inventory tape is read by means of a tape handler. The sales tape is read by means of a second tape handler.

In the computer system, the computer brings the inventory up to date by producing an updated inventory tape which is an exact reproduction of the current inventory tape, except that those changes in stock level required by the information on the sales tape have been made. The computer records the updated inventory information on a blank tape already mounted on a third tape handler.

The updated inventory tape produced on one day becomes the inventory input tape on the next day, while the sales tape continues to originate from without the system.

In any data-processing system, it becomes necessary from time to time to inspect the results of the processing. Thus, for example, in the manual inventory system previously described, management will want to see the stock levels for various stock items. Although many of the purposes for which management would want to make this inspection will be handled automatically by the computer, with the result that manual reference to files in a computer system should be significantly less than such reference in any other kind of system, there will still be occasions when it will be necessary for management to view the records maintained by the computer. Since tape-recording is neither visible nor legible, it isnecessary in a computer system to have some type of printing equipment to produce the reports required by management. The computer records the information to appear in the report on a blank tape mounted on a fourth tape handler. This report tape is used as input the printer in the production of the report.

A schematic of this computer system is shown in Figure 2-13.

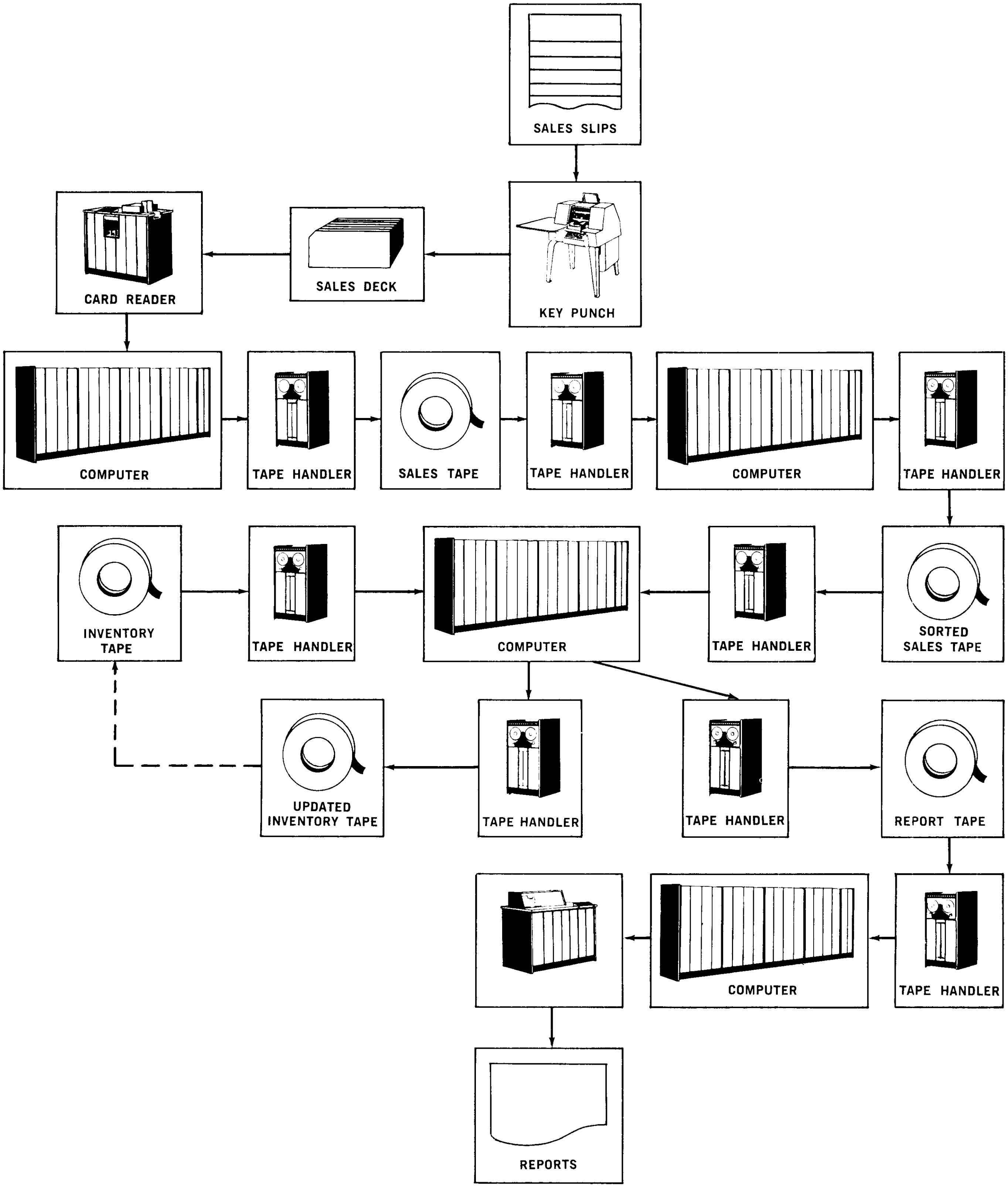


Figure 2-13. Offline Computer

c. Concurrent Processing

The power of computers currently being marketed is such that it is possible for the computer to do different operations at once. For example, the computer can be doing a processing operation such as was described for updating the inventory tape on the basis of the information on the sales tape, convert information punched on cards to magnetic tape, and print information read from a tape onto paper via the printer all at the same time. This approach to computer data processing is known as concurrent processing and is shown schematically in Figure 2-14.

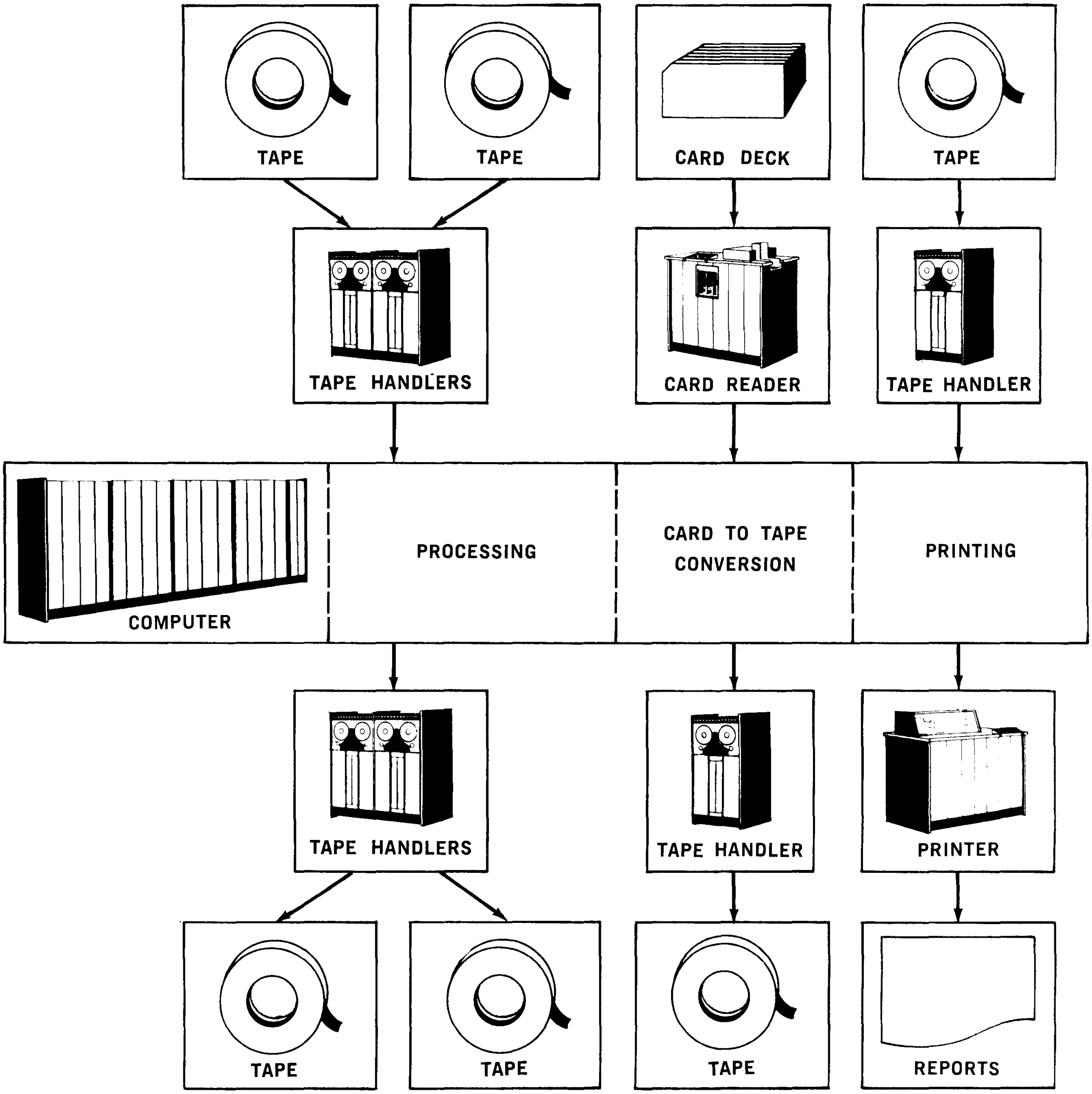


Figure 2-14. Concurrent Processing

B. BASIC OFF-LINE DATA-PROCESSING

The reading of input data from input tapes, the processing of that data to produce output data, and the writing of the output data on output tapes is known as a computer *run*. A computer data-processing system is always made up of one or more – generally more – computer runs.

The input and output of a computer run can always be classified into a number of files. For example, in the simplified inventory run there are two input files, the inventory file and the sales file; and one output file, the updated inventory file. (Actually, there are two output files, since the report file must also be considered an output file, but for purposes of this discussion this file will be ignored). The unit of information in a file is called an *item*, and characteristically, a file is made up of a series of items. For example, in the simplified inventory run, the inventory file would be made up of a series of inventory items, each inventory item referring to one commodity in the company's stock line, and one inventory item appearing on the inventory file for each such commodity. Similarly, the sales file would be made up of a series of sales items, each referring to a commodity on which there has been activity during the day for which the sales file has been prepared.

Each item in a file is made up of a set of *fields* (Figure 2-15). A field is a subunit of information which describes some aspect of whatever the item containing the field refers to. For example, each inventory item contains a stock number field, which specifies the particular number by which the commodity to which the item refers is known, and a quantity field, which specifies the number of units of this commodity that the company has on hand. In reality, of course, an inventory item in an actual data-processing application would contain many more fields than are specified above, but for the simplified inventory run described here, these fields are adequate.

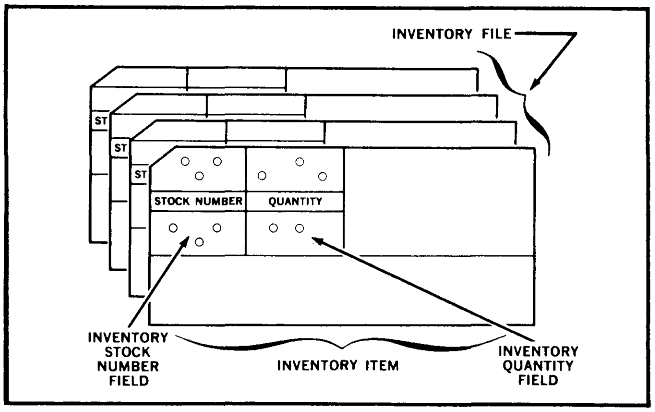


Figure 2-15. Files, Items, and Fields

Now that the terms, *run, file, item,* and *field* have been introduced, the fundamental nature of the way in which an off-line computer processes data can be described in more detail. This description is presented within the framework of the example of the simplified inventory updating run. The inventory file is the master file of this run. The sorted sales file is the transaction file. The items in the sorted sales file are items that have been batched over a period of time and are now going to be applied to the inventory file to update it. The computer reads information from a tape by means of the reading head of the tape handler on which the tape is mounted. This fact means that the computer has access to only one item on the inventory file and one sales item at a time.

The computer reads the first sales item, and on the basis of the stock-number field which it finds in the item, starts a search on the inventory file for the proper item to be updated. If the items on the inventory and sales files are arranged in random order, the computer is going to spend a good deal of time passing the inventory tape over the reading head to accomplish this search. No processing is done during search time. However, if both the inventory and sales items are arranged in the same order, say ascending order, by stock number, the time spent in searching for the correct inventory item is minimized. The first sales item is read. Items are then read from the inventory file until the item with the matching stock number is located, at which point updating occurs. The next sales item is then read. Because of the way the files are ordered, the inventory item to be updated by this sales item is either the one updated by the last sales item or the next active inventory item to be found in moving down the inventory file. In this manner, the next inventory item to be updated is always the one that is closest to the reading head (Figure 2-16).

A general characteristic of off-line data processing is that the items in the files involved are ordered by the field on which searching for updating is to be done. This field is known as the *key* of the item. It is not necessary to reorder the master file each time, since the updated master file from the last cycle becomes the master file for the current cycle. Once the master file is put in order, the updating process produces the updated master file in the same order in which the current master file is read. However, the transaction files, such as the sales file, are generated in random order and must be ordered before being used in a run.

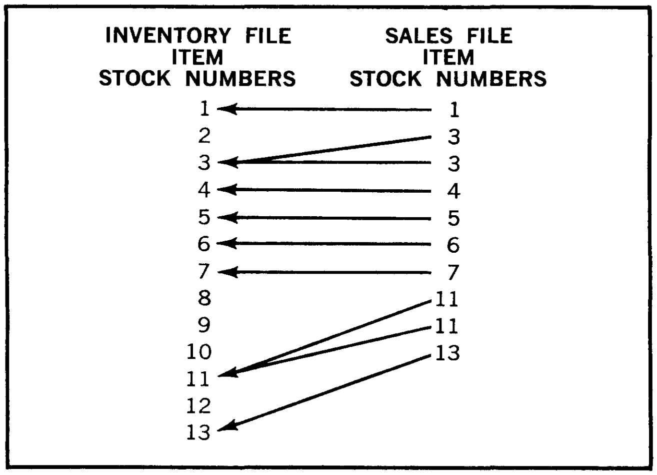


Figure 2-16. Minimizing Search Time by Ordering Files

C. PROGRAMMING THE COMPUTER

The computer, as a data processor, has the ability to read, remember, and write information; do arithmetic; and make logical decisions. It also has the ability to follow a series of instructions that tell it to perform these operations in a particular sequence. However, it is incapable of preparing this series of instructions for itself; this job must be done by a man who both understands what output is to be prepared from what input and what sequence of processing operations is required to form the output from the input. This series of instructions that a computer follows in processing data is known as a *program*, and the man who prepares programs for the computer is known as a *programmer*.

When a programmer is assigned to a computer run, he is generally told by the designer of the data-processing system into which this run fits what input files will be fed into the run and what output files are expected from it. This information is usually described in terms of a *process chart*. A process chart is the laying out of a data-processing system in terms of input, processing and output. For example, Figure 2-17 is a process chart of the computer inventory system shown in Figure 2-13. (Production of the report is eliminated in Figure 2-17.) In this manual, programming exercises will be given in problem – statement form. However, each problem will specify the same three things as a process chart: input, processing and output.

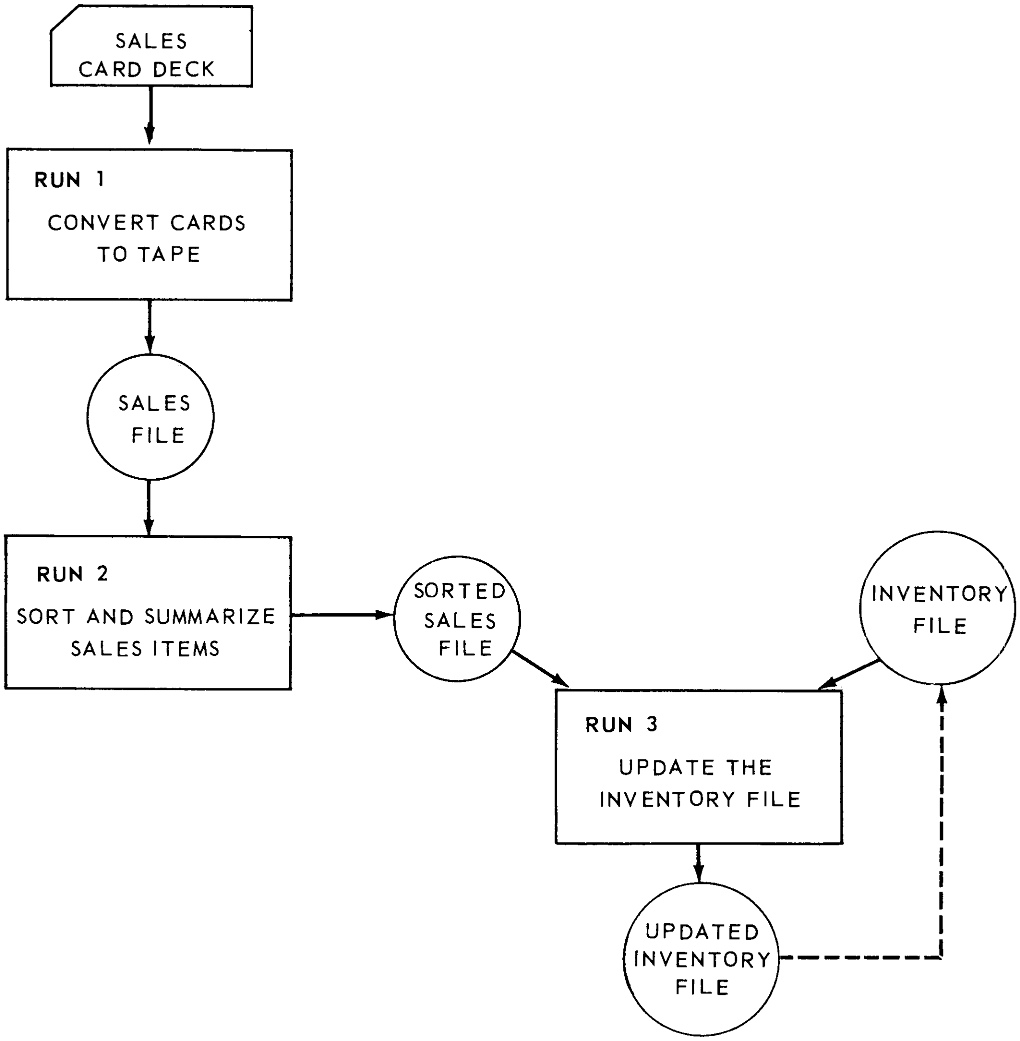


Figure 2-17. Example of a Process Chart

To a greater or lesser extent the programmer will also be given information about the items and fields in the input and output files he is to process in his run. It is the programmer's responsibility to complete this job of *item design*. For each item in each file, the programmer must specify in complete detail what fields make up the item. He must specify the order in which the fields are to be recorded and how many characters are to be allocated to each field. For example, for the simplified inventory run (run 3 in Figure 2-17), the programmer must specify the following file, item and field information.

The simplified inventory run involves three files:

1. *The inventory file.*
2. *The (sorted) sales file.*
3. *The updated inventory file.*

Each inventory-file item consists of two fields:

1. *The stock-number field.*
2. *The quantity-on-hand field.*

Each sales-file item consists of two fields:

1. *The stock-number field.*
2. *The quantity-sold field.*

Finally, each updated inventory file item consists of two fields:

1. *The stock-number field.*
2. *The quantity-on-hand field.*

Notice that the fields for the updated inventory item are identical to the fields of the inventory item; this situation is as it should be, since the updated inventory file is basically a copy of the inventory file, the only changes made being those specified by the sales file.

In this manual, item design will be one of the givens in the programming exercises.

Besides completing the item design for the run he has been assigned to, the programmer must also figure out the sequence of logical steps that must be gone through to produce output files from the input files. This job is called the *logical analysis* of the run. For example, for the simplified inventory updating run, the logical analysis might be as follows.

The first inventory item and the first sales item are read from their respective files into the computer's store. The stock number field of the inventory item currently in the store is then compared with the stock number field of the current sales item. If the two stock numbers are not equal, the current inventory item is not altered, but instead becomes the current updated inventory item, which is written on the updated inventory file; the next inventory item is read from the inventory file into the store; and the comparison of stock numbers is once more made. As long as this comparison does not check out for equality, this process continues, since the arrangement of the items in the files determines that the current sales item refers to an inventory item which is further down the inventory file and that all items preceding this inventory item on the inventory file were not active during the period the current sales file was being compiled. When the stock numbers of the current inventory and sales items prove to be equal, the quantity-on-hand field of the current inventory item is reduced by the quantity-sold field of the current sales item, the next sales item from the sales tape is read, and the comparison of stock numbers is resumed. Notice that the inventory item just updated does not immediately become the updated inventory item, since more than one sales item may refer to it. This process is continued until there are no more sales items, at which point there are no more inventory items to be updated, and the inventory items remaining in the inventory file are moved to the updated inventory file. When there are no more inventory items, the run is complete.

A more formal statement of this logical analysis is shown below.

```
 1. Read The First Inventory Item.
 2. Read The First Sales Item.
 3. Compare The Stock-Number Field Of The Current Inventory Item With The Stock-Number Field Of The Current Sales Item; If They Are Equal, Go To Step 8.
 4. Make The Current Inventory Item The Current Updated Inventory Item.
 5. Write The Current Updated Inventory Item.
 6. Read The Next Inventory Item; If There Are No More, Go To Step 13.
 7. Go To Step 3.
 8. Subtract The Quantity-Sold Field Of The Current Sales Item From The Quantity-On-Hand Field Of The Current Inventory Item.
 9. Read The Next Sales Item; If There Are No More, Go To Step 11.
10. Go To Step 3.
11. Change Step 7 To Go To Step 4.
12. Go To Step 4.
13. Stop.
```

2B. INTRODUCTION TO PROGRAMMING

A. REPRESENTATION OF INFORMATION

Any positive number can be repesented by a row of marks such as 111111111 (or 9), although all but the smallest numbers become unwieldy in such notation. For ease of manipulation a positional notation using symbols to represent different rows of marks is more convenient. One such notation is the Arabic, which uses ten different symbols or digits, 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

The number of different digits used in a positional notation or system is known as the base of the system. Using one digit position, quantities as large as nine can be represented in the decimal system. To represent a quantity larger than nine another digit position must be used. Thus, to represent the quantity ten a carry is made into the digit position to the left and the original digit position reverts to zero. The expansion of this system is exemplified by the odometer of a car. In positional notation each digit position, or column, implies a power of the base as a multiplier of the digit in the column. The decimal number 1076 is positional notation for the expression,

$$(1 \times 1000) + (0 \times 100) + (7 \times 10) + (6 \times 1)$$

The columns imply powers of ten,

$$\begin{aligned} 1 &= 1 &&= 10^0 \\ 10 &= 10 &&= 10^1 \\ 100 &= 10 \times 10 &&= 10^2 \\ 1000 &= 10 \times 10 \times 10 &&= 10^3 \end{aligned}$$

and, appropriately enough, are named the units column, the tens column, the hundreds column, and so on.

A computer that represents numbers in decimal notation must have storage elements capable of assuming ten easily distinguishable stable states, one for each possible digit. While such elements exist, their cost prohibits the construction of a computer that represents numbers in decimal notation. Electronic elements lend themselves most naturally to two-stable-state devices. Thus, computers usually represent numbers in the base two or binary system. The binary system can be built up in a way ânalogous to the decimal. There are two possible digits, 0 and 1, used in conjunction with successive powers of two.

$$\begin{array}{l} 2^0 = 1 \\ 2^1 = 2 \\ 2^2 = 4 \\ 2^3 = 8 \\ \cdot \quad \cdot \\ \cdot \quad \cdot \\ \cdot \quad \cdot \end{array}$$

Thus, the binary equivalent of a decimal nine is 1001, which is binary notation for the expression–

$$(1 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1)$$

1. Student Exercises

Write the binary equivalents of the decimal numbers 6, 13, 15, 27 and 43.

2. Binary Addition

The addition table for the binary system is

$$\begin{array}{r} 0 + 0 = 0 \\ 0 + 1 = 1 \\ 1 + 1 = 10 \\ 1 + 1 + 1 = 11 \end{array}$$

The sum of two binary ones is the binary number 10, the binary equivalent of a decimal two. The binary number 10 is not what is called ten, which is a decimal, not a binary number. Similar remarks hold for the sum of three binary ones, which is the binary number 11, not the decimal number eleven.

a. Example:

DECIMAL	BINARY
13	1101
14	1110
27	11011

b. Student Exercises

Add the following:

$$\begin{array}{r} 1011 \\ \underline{1111} \end{array} \qquad \begin{array}{r} 1010 \\ \underline{10111} \end{array} \qquad \begin{array}{r} 11001 \\ \underline{10111} \end{array}$$

3. Addition of Two Numbers with Opposite Signs

While addition of two numbers with opposite sign could be done by use of a *subtraction* table, computers use the method of complementation. For any given number there exists a second number which when added to the first will produce a sum consisting of a one followed by as many zeros as there are digits in the first number. The second number is the complement of the first.

- To get the complement of a binary number

  1. Replace the ones with zeros and the zeros with one and,

  2. Add a binary one to the result.

  For example, given 1101 replace ones with zeros and zeros with ones, and add a binary one

```
                 0010
                    1
Complement       0011
                 ----
Proof:           1101 + 0011 = 10000
```

- To add two numbers with opposite signs:

  1. Equalize the number of digit positions by inserting non significant zeros in the number with the smaller absolute value.
  2. Take the complement of the absolute value of the smaller in absolute value.
  3. Add the absolute value of the other number to the result.
  4. Drop the most significant carry and,
  5. Prefix the sign of the number with the larger absolute value to the sum.

a. Example:

```
Add       −101101
          +   1011
          --------
Step 1.   −101101
          +001011

Step 2.   The smaller in absolute value is

            001011

            110100
          +      1
          --------
            110101

Step 3.     101101
          + 110101
          --------
           1100010

Step 4.     100010

Step 5.   −100010
```

b. Student Exercises

Add the following:

$$\begin{array}{r} -1011 \\ +1111 \\ \hline \end{array} \qquad \begin{array}{r} +\ 1010 \\ -10111 \\ \hline \end{array} \qquad \begin{array}{r} -11001 \\ -10111 \\ \hline \end{array}$$

4. Subtraction

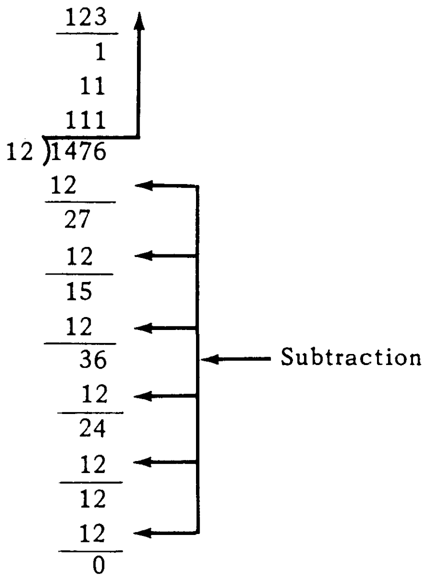
Subtraction can be accomplished through the use of addition by changing the sign of the subtrahend and adding the subtrahend to the minuend.

5. Multiplication

Binary multiplication, like decimal multiplication, can be done by a series of additions.

6. Division

Binary division, like decimal division, can be done by a series of additions and subtractions.



Thus, division can be done by addition.

7. Coded Binary

Binary representation is used in computers in one of two forms. The first is the binary notation just described called pure binary representation. The other is called coded binary representation. In this representation, only the pure binary equivalents of the ten decimal digits are used.

DECIMAL	PURE BINARY
	8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Any decimal number greater than nine is represented by a combination of the above codes. For example, the decimal number 147 would be represented as

0001 0100 0111

One modification of coded binary representation is called the excess-three representation. The excess-three expression of a decimal number is equal to the pure binary representation of a decimal that is three greater than the number being represented. For example, the excess-three representation of decimal 5 is 1000, which in pure binary represents decimal 8 —— or 3 greater than 5.

DECIMAL	EXCESS-THREE (XS-3)
	8421
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

Excess-three representation has two advantages over straight coded binary.

1. *The addition of two excess-three numbers produces a carry if the addition of their decimal equivalents produces a carry.*
2. *An excess-three number can be complemented in the same way as a pure binary number.*

8. Excess-Three Arithmetic

If two like signed excess-three digits are added, the decimal equivalent of their sum is not equal to the sum of the decimal equivalents of the digits.

$$\begin{array}{ccc} \text{DECIMAL} & \qquad\qquad & \text{EXCESS-THREE} \\ 5 & & 1000 \\ 1 & & 0100 \\ \hline 6 & & 1100 \end{array}$$

The decimal equivalent of the excess-three digit 1100 is not six, but nine. The reason for this fact is that if the addition does not produce a carry the sum is not in excess-three representation, but in excess-six notation. To convert the sum to excess-three representation it is necessary to subtract the pure binary equivalent of a decimal three, 0011. The complement of the pure binary number 0011 is 1101. Thus, to correct the sum of two excess-three digits that do not produce a carry, add the pure binary number 1101 to the sum.

$$\begin{array}{c} 1100 \\ 1101 \\ \hline 1001 \end{array}$$

The excess-three digit 1001 is the equivalent of a decimal six. However, if the addition of two excess-three digits produces a carry, the sum is not represented in excess-six.

$$\begin{array}{rcr} 5 & \qquad\qquad & 1000 \\ 6 & & 1001 \\ \hline \swarrow 1 & & \swarrow 0001 \end{array}$$

The reason for the above fact is that the carry in the excess-three addition carries the equivalent of a decimal 16 out of the sum. Ten of this 16 is the decimal carry to the next column, which is desired; three of the 16 is what previously produced an excess-six sum, and its carry is of no concern; but the last three is what was necessary to produce an excess-three sum. Thus, the sum comes out in pure binary representation. To convert the sum to excess-three representation it is necessary to add the pure binary equivalent of a decimal three, 0011, to the sum.

$$\begin{array}{c} 0001 \\ 0011 \\ \hline 0100 \end{array}$$

The excess-three digit 0100 is the equivalent of a decimal one. In summary, to add two like signed excess-three numbers:

1. *Add the numbers according to the rules of pure binary addition and*
2. *Apply "correction factors" to each digit in the sum.*

The correction factors are as follows:

- If the column in which the digit appears did not produce a carry, add the correction factor 1101.
- If the column produced a carry, add 0011.

a. Example :

	0100	0111	1010
	1001	0110	1000
Intermediate sum	1101	1110	↙0010
Correction factors	1101	1101	0011
Excess-three sum	1010	1011	0101

Since the correction factors apply only to individual digits and not the entire sum, any carry produced is ignored.

b. Student Exercises

Add	0110	1010	0011
	0111	1000	1010
	1000	0101	1100
	1010	0111	0101

c. Addition of Two Excess-Three Numbers with Opposite Signs

Two excess-three numbers with opposite sign are added in the same way as two pure binary numbers with opposite signs.

1) Example:

Add	−1010	0111	1001	0100
	+	0100	0110	1100
Step 1.	−1010	0111	1001	0100
	+0011	0100	0110	1100
Step 2.	1100	1011	1001	0011
				1
	1100	1011	1001	0100
Step 3.	1010	0111	1001	0100
	1100	1011	1001	0100
	↙0111	↙0011	↙0010	1000
	0011	0011	0011	1101
	↙1010	0110	0101	0101
Step 4.	1010	0110	0101	0101
Step 5.	−1010	0110	0101	0101

2) Student Exercises

Add	+1010	0011	0111	PROBLEM 1
	−0101	0011	1010	
	−0111	1011	1100	PROBLEM 2
	+1000	0110	0101	

9. Decimal Representation

The four-bit binary coded excess-three notation is referred to as decimal representation. This allows for sixteen characters whose values range from 0000 to 1111. In this group are included all the numerics and some special characters, but no alphabetics.

10. Alphanumeric Representation

To represent the twenty-six alphabetic characters the sixteen possibilities of decimal representation are not sufficient. Therefore, in this representation, a two-bit zone is added and precedes each binary coded excess-three notation. The other four bits are called the numeric portion.

Each representation in the numeric portion may be preceded by one of four possible zones: 00, 01, 10 or 11. A total of sixty-four different characters may be designated in this format.

Alphanumeric information may be distinguished from decimal information by the zone. It is possible to present numeric characters in both decimal format and alphanumeric format. A complete table of character representation can be found in the Character Code Chart, Figure 2-18.

11. Octal Representation

Consider the following pure binary number.

001101000101011001111100

The length of this binary number written in this notation makes it difficult to both read the number and transcribe it correctly. Reading and transcription is eased by breaking the number into groups, as follows:

001 101 000 101 011 001 111 000

Nevertheless, the number is still difficult to handle. To ease binary number manipulation, a convention of writing binary numbers in a code is generally adopted. The code used is *octal*. Octal notation is a number system with a base of eight. Thus, the coefficients in the octal number system are 0, 1, 2, 3, 4, 5, 6 and 7. The binary equivalents of these octal numbers are as follows:

OCTAL	BINARY
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

COBOL – FORTRAN SET

	00	01	10	11
0000	Δ blank	+		
0001	;	)	*	(
0010	–	.	$	,
0011	0			' (Apos.)
0100	1	A	J	/
0101	2	B	K	S
0110	3	C	L	T
0111	4	D	M	U
1000	5	E	N	V
1001	6	F	O	W
1010	7	G	P	X
1011	8	H	Q	Y
1100	9	I	R	Z
1101	:	=		
1110	<			
1111	>			

Figure 2-18. 6-Bit Printable Character Codes

Thus, the octal numbers have as their binary equivalents all binary numbers that can be represented in three bit positions. As a result, octal notation is a natural as a code for binary notation, each octal digit standing for a three bit group of binary numbers. For example, in octal code, the above binary number would be written as follows:

153053170

B. THE CENTRAL PROCESSOR

The focus of the UNIVAC III computer is the Central Processor. The Processor accepts information from input units, stores that information, does arithmetic operations, makes logical decisions, and produces new – or updated – information for output. The information handled in this way is called *data*. Stock numbers, inventory quantities, and sales quantities are examples of data. The arithmetic operations and logical decisions done on the data is called *processing*.

1. The Storage Unit

To process data, the Processor must have stored in an accessible place three types of information:

1. *the data itself,*
2. *instructions, and*
3. *constants.*

Instructions are coded units of information which are used to direct the Processor in the processing of data. A program must include an instruction for each operation that the Processor is to do, such as the subtraction of one quantity from another. Constants are units of information which the Processor must have to perform certain operations. For example, if a salesman's commission on a sale is ten percent of the sales price, the processor must multiply the sales price by ten percent to develop this commission amount. To do this the Processor must have available to it a constant of ten percent.

Data, instructions and constants are made immediately available to the Processor by storing them in the Processor's *store*. The store is made up of *storage locations*. The amount of store available on the UNIVAC III is variable at the user's option. Store is obtainable in the following amounts:

- 8192 storage locations,
- 16,384 storage locations,
- 24,576 storage locations, and
- 32,768 storage locations.

For ease in reference, these store sizes are spoken of as *8K, 16K, 24K* and *32K*, respectively. Any storage location may be used to store data, instructions or constants. The amount of information that can be stored in one storage location is fixed. This fixed amount of information stored in any one storage location is called a *word*. A word consists of 27 binary bits of information. The bit positions, for reference, are numbered 1 through 27, from right to left. The rightmost bit (bit 1) is the Least Significant Bit (LSB) and the leftmost bit is the Most Significant Bit (MSB). This 27-bit word is divided into three portions:

1. *The Data Portion (bits 1 through 24).*
2. *The Sign (bit 25).*
3. *The Checking Portion (bits 26 and 27).*

CHECKING		S	DATA	
27	26	25	24	1

The sign of the word is used in arithmetic and comparison operations. In this position a binary 0 represents a positive value while a binary 1 indicates a negative value.

The checking bits are used by the checking circuitry of the system to assure that there is no change in the information content of the words because of some malfunction or electronic phenomenon as the words are transferred through the circuitry of the system. The type of checking used is modulo 3, residue zero. Since there is no possible access to these bit positions through programming and they have no value as data, no further reference will be made to them and only bits 1 through 25 will be considered.

a. Addressing

Each storage location has a label or address by which its contents may be referenced so that each may be distinguished from any other. The addresses used within the Central Processor are in pure binary form. For ease of representation, only their decimal equivalents will be used here. The numbering of these addresses is sequential and the first storage address of any system is 00000. The highest storage address of a system having the maximum size store is 32767. This allows for the addressing of a possible 32,768 words.

	STORAGE ADDRESS	CONTENTS
Word 1	00000	25 bit positions
Word 32768	32767	25 bit positions

A given word in a storage location may be referenced as often as desired, because when it has been stored in that location it remains there until it is "erased" by the transfer of another word into the same location.

b. Data Word Format

In the UNIVAC III there are three types of data words. All three types use bit position 25 to represent the sign of the word.

1) The Decimal Data Word

S	DIGIT 6	DIGIT 5	DIGIT 4	DIGIT 3	DIGIT 2	DIGIT 1
25	24 21	20 17	16 13	12 9	8 5	4 1

The data that is represented in bit positions 1-24 is arranged in six, fixed, 4-bit groups. These groups are coded in binary excess-3 representation. Bit positions 1 through 4 represent the LSD and bit positions 21 through 24 the MSD.

Example:

0	1 1 0 0	1 0 0 0	1 0 0 1	0 1 1 0	0 1 0 0	0 1 0 1	= + 956312

2) The Alphanumeric Data Word

S	DIGIT 4	DIGIT 3	DIGIT 2	DIGIT 1
25	24 19	18 13	12 7	6 1

The data is arranged in 4, fixed, 6-bit groups. These groups are coded in binary excess-3 representation with zone. Bit positions 1-6 represent the LSD and bit positions 19-24 the MSD.

Example:

0	0 1 0 1 0 1	1 0 0 1 1 0	1 1 0 1 1 1	0 0 0 1 1 1	= + BLU4

3) The Binary Data Word

S	24-BIT BINARY VALUE
25	24 1

The data is represented as one pure binary 24-bit value ranging from 0 through plus or minus 16,777,215. (2^{24}-1).

Example:

0	0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 1 0 1	= + 532637

2. The Arithmetic Unit

The Processor does arithmetic operations and makes logical decisions by means of its internal arithmetic unit. This unit has characteristics in common with a desk calculator, since it contains an adder to produce the sum or difference of two quantities, and circuitry to use the adder in the development of a product of two quantities or the quotient of one quantity divided by the other. Additional circuitry allows the Processor to use the adder to make logical decisions concerning the equality or relative magnitude of two quantities.

To operate on a quantity, the Processor must transfer it from the store to the arithmetic unit. Four *arithmetic registers* provide temporary storage for such quantities within the arithmetic unit. These arithmetic registers are designated by a four bit positional representation. The designations are 1000, 0100, 0010 and 0001. Since these designations are the pure binary representations of the decimal numbers 8, 4, 2 and 1, these registers are referred to as AR8, AR4, AR2 and AR1, where AR stands for arithmetic register. Each register is similar to a storage location in that it has the capacity to store one word of information.

3. Control Unit

The function of the control unit is to select, in the proper sequence, each instruction from storage, interpret, and execute it.

The selection of an instruction is performed in a sequential manner. If the instruction just executed were located in storage address 00698, the next instruction would come from storage address 00699, and so on sequentially through the store.

Each instruction is a UNIVAC III word and, as such, is 25 bits in length. Its structure as shown in the diagram below, is different from any of the data word formats.

IA FS	X	OP CODE	AR	m
25	24 21	20 15	14 11	10 1

Of the sections shown only three will be considered for the present time.

m — The contents of bit positions 1 through 10 represent the binary storage address of the instruction operand. It is the contents of this storage location which will be operated upon by the instruction. With binary 1's in each of these bit positions the highest storage location that could be indicated in the m portion is 1023. The method of addressing storage locations greater than 1023 will be discussed later. For the present, consider a store whose size is 1024 words with an address range of 0000 - 1023.

AR The contents of bit positions 11 through 14 represent the address of an arithmetic register in the arithmetic unit. Each arithmetic register is a temporary storage location for one UNIVAC III word and therefore is 25 bits in length. Utilizing bit positions 11-14, respectively, the addresses are:

AR 8421	DECIMAL EQUIVALENT
1000	8
0100	4
0010	2
0001	1

OP CODE The contents of bit positions 15 through 20 represent the OPeration Code which indicates the operation to be performed. Although the Central Processor only recognizes a binary configuration as the OP Code, for ease of coding, a mnemonic OP Code will be used to designate each instruction.

C. CODING

Coding is the translation of a logical analysis of a run into an organized series of instructions that are intelligible to the Processor. An instruction must be given to the Processor for each operation it is to do. In the UNIVAC III, an instruction generally specifies three things:

1. *The operation to be done.*
2. *The address of the data to be operated on.*
3. *The arithmetic register with respect to which the operation is to be done.*

These three elements of an instruction are specified, respectively, in the OP, m and AR portions of the instruction.

To see how coding might appear, consider the function of adding two quantities together and storing the resulting sum. The Processor would perform this function in three operations.

1. *Select one quantity.*
2. *Select the second quantity and add it to the first.*
3. *Store the sum.*

For the Processor to do these three operations in the order indicated, it must have a program of instructions. The program for this addition problem would consist of three instructions, one for each of the operations. These particular instructions might have the following mnemonic codes.

- LA – Select the quantity from the storage location specified and transfer it to the AR specified.

- DA – Select the quantity from the storage location specified and add it to the quantity in the AR specified, the sum to be returned to that same AR.

- SA – Store the quantity in the AR specified into the storage location specified.

A corresponding problem might be stated as follows:

1. *Assume that the two quantities to be added are stored in storage locations 800 and 801.*
2. *Store the sum of these quantities in storage location 802.*

The coding needed to execute the problem might look like the following.

LOCATION OF INSTRUCTION	INSTRUCTION		
	OP	AR	m
0	LA	8	800
1	DA	8	801
2	SA	8	802

The first line of coding brings the quantity stored in location 800 into AR8. The second line adds the quantity stored in location 801 to the quantity now in AR8 and stores the resulting sum in AR8. The last instruction stores in storage location 802 the quantity now in AR8.

The storage location for storage of the first instruction (storage location 0) was chosen arbitrarily. However, the second and third instructions were stored, respectively, in storage locations 1 and 2 to assure that the control unit of the Processor would cause the instructions to be executed in the order written. Only in this order of execution would the instructions effect the desired result.

Of course, in the computer's store, the op codes would appear as six bit codes (LA happens to be 001010, DA 010000, and SA 001000), the AR designation would appear as a four bit positional representation (AR8 is 1000), and the m portions would appear as 10 bit binary addresses (800 is 1100100000, 801 is 1100100001, and 802 1100100010). Therefore, to be effective in the computer, the above instructions would have to be stored in storage locations 0-2 in the following form:

IA	X	OP	AR	m
0	0000	001010	1000	1100100000
0	0000	010000	1000	1100100001
0	0000	001000	1000	1100100010

Instructions would be hard to write in such form, and would be even harder to read after having been written. As a consequence, instructions are not written in this *object code* form, but are instead written in another *source code* form. This source code form is the UTMOST language. A program written in UTMOST language is recorded on tape and fed into a computer program called the UTMOST Assembler. The Assembler has the function of writing out on another tape object code instructions corresponding to the source code instructions fed into the Assembler as input. The coding on this object code tape can then be loaded into the computer to do the operations described by the programmer in source code. All examples in this manual will be coded in the UTMOST language.

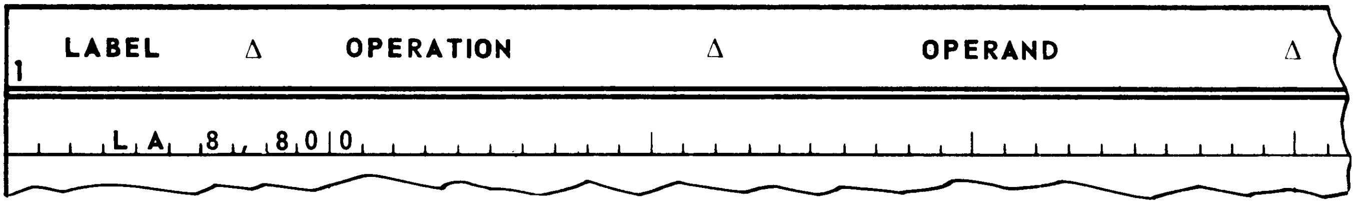
UTMOST Coding is written on UTMOST coding paper, an example of which is shown in figure 2-19. The coding paper is essentially nothing more than a series of lines each marked off into 80/90 character segments. Writing of UTMOST code is restricted to the first 72 of these positions. The coding paper is used in the following way.

In general, one instruction is written per line. The address of the storage location in which an instruction is to be stored must be justified left in the line on which the instruction is written. (In actuality, UTMOST uses "labels" rather than storage addresses for this purpose, but introduction of the concept of labels is delayed until later in this manual.) The address must be followed by one or more spaces. The number of spaces used is an option of the programmer. The mnemonic op code for the instruction is then written. The op code must also be followed by one or more spaces. The AR and m portions of the instruction are then written. They must be separated by a comma. Following the m portion of an instruction, provided that at least one space separates the m portion and the comment, the programmer may write any comments about this instruction that he wishes. As indicated, op codes are written in mnemonic form, AR's are indicated by the decimal equivalent of their pure binary code (8, 4, 2, and 1), and the addresses in the m portion are written in decimal.

LOAD AR – LA

Transfer the contents of the storage location specified to the arithmetic register specified.

Example:



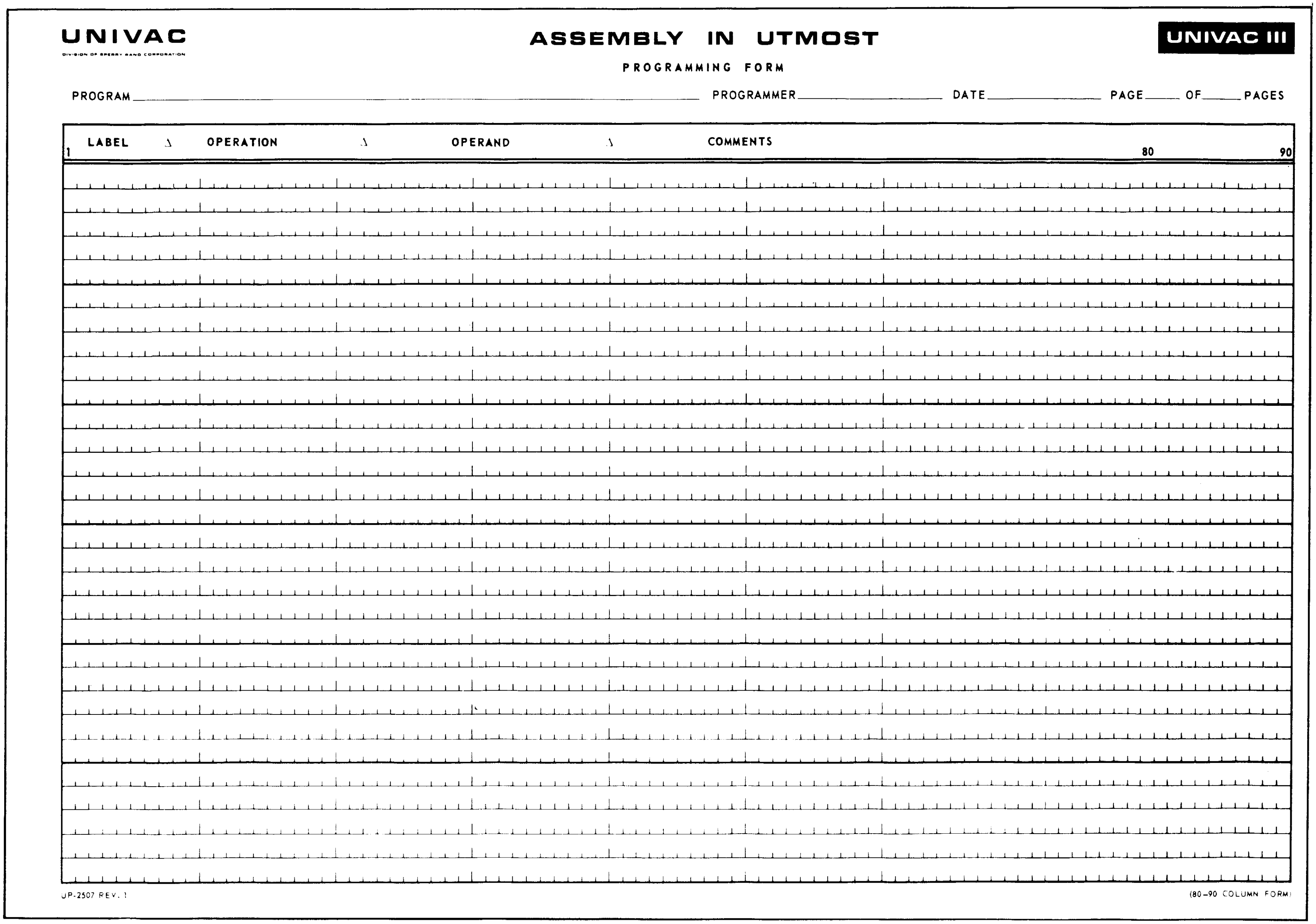
1 LABEL Δ OPERATION Δ OPERAND Δ

```
   LA 8, 800
```

(AR8) i = – 012345 (AR8) f = + 987654

(800) i = + 987654 (800) f = + 987654

The notation used in this example is as follows. Parentheses stand for "the contents of". Thus (800) means "the contents of storage location 800". Parentheses followed by an "i" stand for "the contents of_____ before instruction execution". Thus, (800) i means "the contents of storage location 800 before instruction execution". Parentheses followed by an "f" stand for "the contents of_____ after instruction execution". Thus, (800)f means "the contents of storage location 800 after instruction execution. Mnemonically, "i" stands for "initial", "f" for "final".



UNIVAC
DIVISION OF SPERRY RAND CORPORATION

ASSEMBLY IN UTMOST

PROGRAMMING FORM

UNIVAC III

PROGRAM________ PROGRAMMER________ DATE________ PAGE____ OF____ PAGES

1 LABEL	Δ OPERATION	Δ OPERAND	Δ COMMENTS	80	90

UP-2507 REV. 1

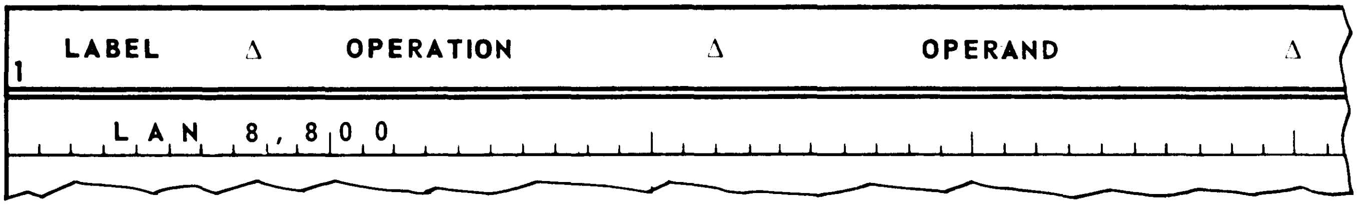
(80–90 COLUMN FORM)

Figure 2-19. Assembly in UTMOST Coding Form

LOAD AR NEGATIVE – LAN

Transfer the contents of the storage location specified to the arithmetic register specified and change the sign.

Example:



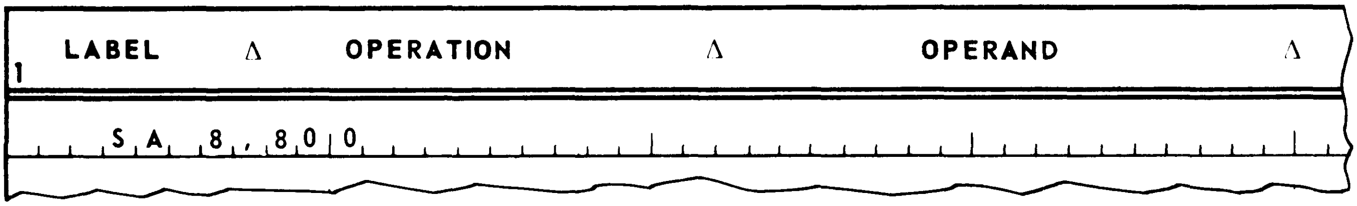
```
1  LABEL    Δ    OPERATION         Δ         OPERAND         Δ
      LAN  8,800
```

(AR8) i = – 012345	(AR8) f = –987654
(800) i = + 987654	(800) f = +987654

STORE AR – SA

Transfer the contents of the arithmetic register specified to the storage location specified.

Example:



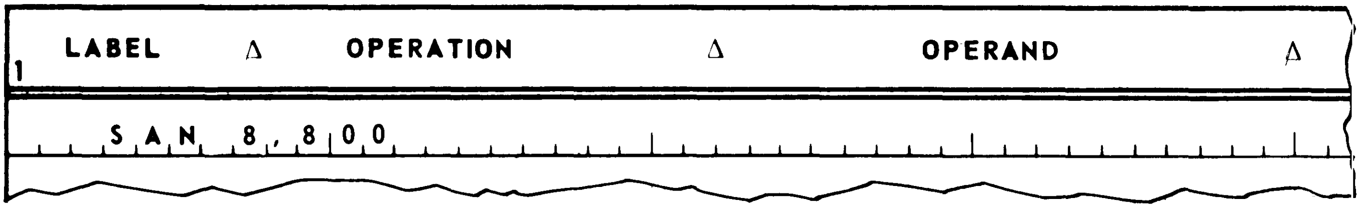
```
1  LABEL    Δ    OPERATION         Δ         OPERAND         Δ
      SA  8,800
```

(AR8) i = – 012345	(AR8) f = –012345
(800) i = + 987654	(800) f = _ 012345

STORE A NEGATIVE – SAN

Transfer the contents of the arithmetic register specified to the storage location specified and change the sign.

Example:



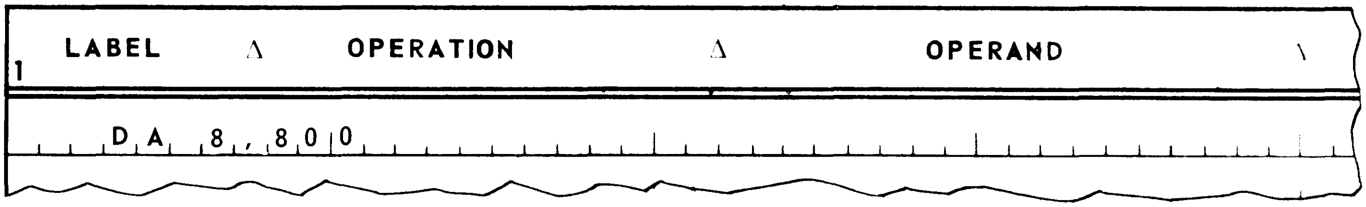
```
1  LABEL    Δ    OPERATION         Δ         OPERAND         Δ
      SAN  8,800
```

(AR8) i = – 012345	(AR8) f = – 012345
(800) i = + 987654	(800) f = + 012345

DECIMAL ADD – DA

Add the contents of the storage location specified to the contents of the arithmetic register specified and store the sum in that arithmetic register. The Processor assumes that the operands are in decimal format. In the operands, a decimal digit with a bit combination of 0000 (decimal “digit” “space”) will be treated as a bit combination of 0011 (decimal digit “zero”)

Example:



```
1  LABEL   Δ   OPERATION   Δ   OPERAND   \
      DA  8, 800
```

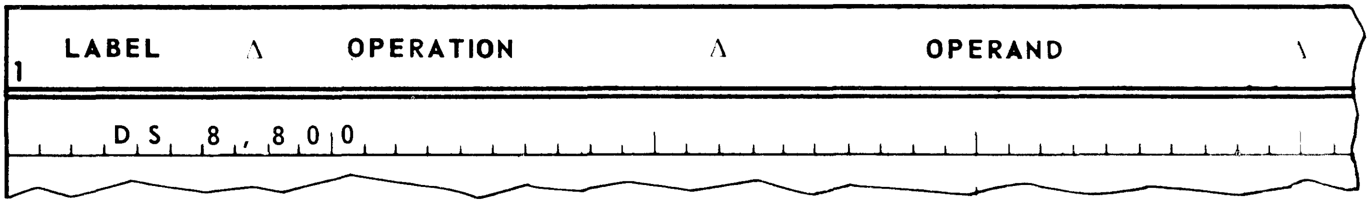
(AR8) i = −012345 (AR8) f = + 975309

(800) i = +987654 (800) f = + 987654

DECIMAL SUBTRACT – DS

Subtract the contents of the storage location specified from the contents of the arithmetic register specified and store the difference in that arithmetic register. The Processor assumes that the operands are in decimal format. In the operands a decimal digit with a bit combination of 0000 will be treated as a decimal zero.

Example:



```
1  LABEL   Δ   OPERATION   Δ   OPERAND   \
      DS  8, 800
```

(AR8) i = – 012345 (AR8) f = + 999999

(800) i = + 987654 (800) f = + 987654

BINARY ADD – BA

Add the contents of the storage location specified to the contents of the arithmetic register specified and store the sum in that arithmetic register. The Processor assumes that the operands are in binary format.

Example:

```
1 LABEL     Δ     OPERATION          Δ          OPERAND          Δ
     BA  8,800
```

(AR8) i = – 15053170_8 (AR8) f = + 45651417_8

(800) i = + 62724607_8 (800) f = + 62724607_8

Notice that the binary numbers in this example are expressed in octal notation.

BINARY SUBTRACT – BS

Subtract the contents of the storage location specified from the contents of the arithmetic register specified and store the difference in that arithmetic register. The Processor assumes that the operands are in binary format.

Example:

```
1 LABEL     Δ     OPERATION          Δ          OPERAND          Δ
     BS  8,800
```

(AR8) i = – 15053170_8 (AR8) f = – 77777777_8

(800) i = + 62724607_8 (800) f = + 62724607_8

1. Example using Preceding Instructions

The onhand quantity of a commodity is stored in location 800, the onorder quantity in location 801, and the expected requirements for the next 60 days in 802. All quantities are in decimal format. Store the sum of the onhand and onorder quantities reduced by the expected requirements in location 803.

a. Logical Analysis

1. Add the onorder quantity to the onhand quantity.

2. Reduce the sum by the expected requirements.

3. Store the difference.

b. Coding

```
1  LABEL    Δ    OPERATION              Δ              OPERAND              Δ
0               LA 8, 800      ADD ONORDER AND ONHAND
1               DA 8, 801
2               DS 8, 802      SUBTRACT EXPECTED REQUIREMENTS
3               SA 8, 803      STORE DIFFERENCE
```

To produce this coding, the programmer might have approached the problem in the following manner. As indicated in the logical analysis, the first data-processing step is to add the onorder quantity to the onhand quantity. Since the quantities are in decimal format, to do an addition the Processor must be given a DA instruction. This instruction requires that one of the quantities to be added must be in an arithmetic register. The other quantity must be selected from the storage location specified. Since both quantities are presently in storage locations, one of them must be transferred to an arithmetic register before they can be added together. The choice of the arithmetic register is arbitrary. Suppose AR8 is chosen. To place the onhand quantity in AR8, the Processor must execute an instruction of the form LA 8, 800.

Choice of storage location 0 for storage of the LA instruction is arbitrary. However, it does require that the next instruction in the program be stored in storage location 1. Following execution of the LA 8, 800 instruction, the onhand quantity is stored in AR8. To add the onorder quantity to the contents of AR8, the Processor should have as its next instruction, DA 8, 801. This instruction must be stored in storage location 1. After executing the DA instruction, the Processor has the sum of the onhand and onorder quantities stored in AR8.

The logical analysis indicates that the next operation to be done is the subtraction of the required quantity from this sum. This step calls for a DS instruction. To execute this instruction, the desired minuend must be in an arithmetic register and the subtrahend in a storage location. Since both of these conditions are satisfied, a DS 8, 802 instruction is stored in storage location 2 to subtract the required quantity from the sum of the onhand and onorder quantities in AR8.

The final step is to store the difference in storage location 803. This operation can be done by the execution of a SA 8, 803 instruction stored in storage location 3.

2. Student Exercises

(1) A quantity is stored in storage location 800. Store the quantity in storage locations 801, 802 and 803.

(2) Two quantities are stored in locations 800 and 801. Interchange the quantities.

(3) Three quantities are stored in locations 800, 801 and 802 in decimal format. Store the sum of the quantities in location 803.

(4) Quantities A, B, C and D are stored in locations 800 – 803, respectively, in decimal format. If

$$R = 2A - B + 3(C + D)$$

calculate R and store it in location 804.

D. MULTIWORD OPERANDS

Most instructions may specify multiword operands which may be two, three or four words in length. The number of words in an operand is determined by the number of arithmetic registers specified in the AR portion of the instruction. For example, suppose it is desired to load a two word operand in AR's 8 and 4. The positional notation for AR8 is 1000, for AR4 0100. Therefore, the positional notation for AR's 8 and 4 would be 1100. The decimal equivalent of the pure binary number 1100 is 12, which happens to be the sum of eight and four. Thus, to load a two word operand into AR's 8 and 4, an LA instruction with 12 in its AR portion would be specified. This convention holds true in all cases. Thus, if it is desired to load a three word operand into AR's 4, 2 and 1, an LA instruction with 7 in its AR portion would be specified. (Seven is the sum of four, two and one.)

The arithmetic registers can be conceived of as being arranged in a line, as shown in Figure 2-20.

AR8	AR4	AR2	AR1

Figure 2-20. Arrangement of Arithmetic Registers

Thus, arranged from "most significant" register to "least significant" register the arithmetic registers are listed as AR8, AR4, AR2 and AR1. When using arithmetic registers in a multiword operation, the least significant word of the multiword operand is found in the least significant register indicated, the next least significant word in the next least significant register, and so on, until the most significant word of the multiword operand is found in the most significant register.

When the AR portion of an instruction calls for a multiword operand, the m portion of the instruction specifies the location of the least significant word of the multiword operand. Words of increasing significance in the multiword operand are found in continguous storage locations moving "backword" through the store. Thus, if an instruction specifying a three word operand addresses location 802 in the m portion, the three word operand is found in locations 800, 801 and 802. The most significant word of the three word operand is stored in location 800, the next most significant word in location 801, and the least significant word in location 802. For example, the instruction

1 LABEL	Δ	OPERATION	Δ	OPERAND	Δ
LA 14,802					

would load the contents of location 802 into AR2, the contents of 801 into AR4, and the contents of 800 into AR8.

Figure 2-20 is misleading in the sense that the arithmetic registers are not physically connected in any way. Thus, it is not necessary to load a two word operand in AR's 8 and 4, 4 and 2, or 2 and 1. Any two arithmetic register may be used. For example, the instruction

1 LABEL	Δ	OPERATION	Δ	OPERAND	Δ
LA 13,802					

would load the contents of location 802 into AR1, the contents of 801 into AR4, and the contents of 800 into AR8.

Multiword operands may be used with any of the instructions previously defined in this manual.

1. Example

A quantity is stored in locations 800, 801 and 802. Store the quantity in locations 803, 804 and 805. Do not destroy the contents of AR4.

a. Coding

1	LABEL	Δ	OPERATION	Δ	OPERAND	Δ
0			LA 11,802			
1			SA 11,805			

2. Multiword Arithmetic Instructions

In an arithmetic operation the sign of a multiword operand is determined by the sign of the least significant word of the operand. Therefore, in an arithmetic operation, if the contents of storage location m-1 are –XXXXXX and the contents of m are + XXXXXX, any multiword instruction addressing m as the least significant portion of an operand will involve a positive quantity regardless of the signs of the most significant words. After an arithmetic operation, the correct sign will appear in every word of the result.

Example:

1	LABEL	Δ	OPERATION	Δ	OPERAND	Δ
			DA 12,801			

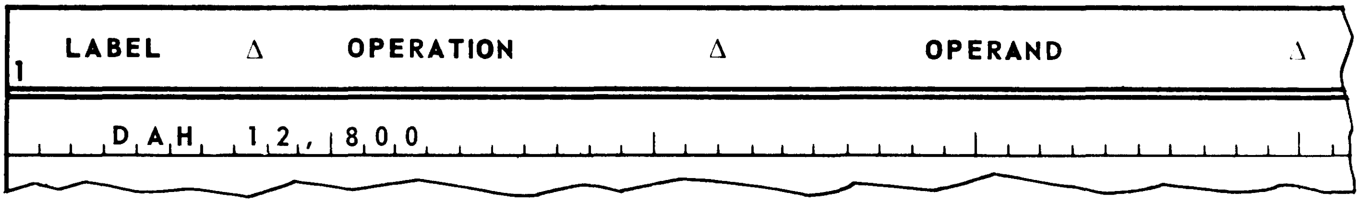
(AR8) i = + 222222	(AR8) f = + 888889
(AR4) i = + 333333	(AR4) f = + 177777
(800) i = – 666666	(800) f = – 666666
(801) i = + 844444	(801) f = + 844444

DECIMAL ADD HIGHER - DAH

The decimal add higher instruction may be used with one word or two word operands. If a one word operand is used, two AR's are specified. If a two word operand is used, all four AR's are specified.

1. If two AR's are specified, add the contents of the storage location specified to the contents of the more significant register specified and store the sum in the less significant register specified.

   Example:

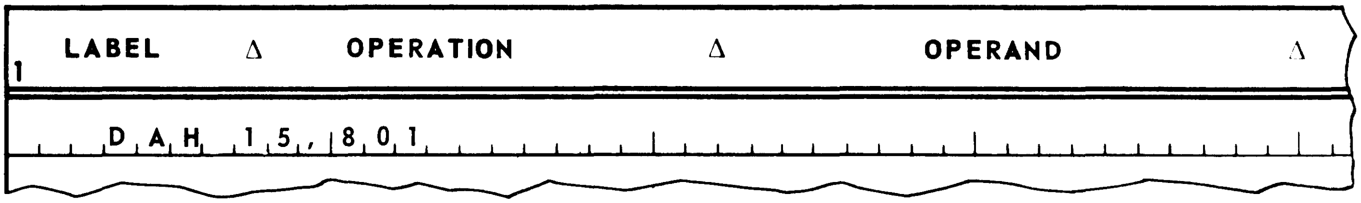


```
1  LABEL     Δ     OPERATION          Δ          OPERAND          Δ
     DAH  12, 800
```

(AR8) i = – 012345	(AR8) f = – 012345
(AR4) i = + 789012	(AR4) f = + 975309
(800) i = + 987654	(800) f = + 987654

2. If four AR's are specified, add the storage operand specified by m (the contents of m-1 and m) to the contents of AR's 8 and 4 and store the sum in AR's 2 and 1.

   Example:



```
1  LABEL     Δ     OPERATION          Δ          OPERAND          Δ
     DAH  15, 801
```

(AR8) i = + 333333	(AR8) f = + 333333
(AR4) i = + 999999	(AR4) f = + 999999
(AR2) i = + 444444	(AR2) f = + 555556
(AR1) i = + 111111	(AR1) f = + 000005
(800) i = + 222222	(800) f = + 222222
(801) i = + 000006	(801) f = + 000006

In both cases (one or two word operands) the Processor assumes that the operands are in decimal format. In the operands, a decimal digit with a bit combination of 0000 will be treated as a decimal zero.

DECIMAL SUBTRACT HIGHER – DSH

If two AR's are specified, subtract the contents of the storage location specified from the contents of the more significant register specified and store the sum in the less significant register specified. If four AR's are specified, subtract the storage operand specified by m from the contents of AR's 8 and 4 and store the sum in AR'S 2 and 1. The Processor assumes that the operands are in decimal format. In the operands, a decimal digit with a bit combination of 0000 will be treated as a decimal zero.

BINARY ADD HIGHER – BAH

If two AR's are specified, add the contents of the storagelocation specified to the contents of the more significant register specified and store the sum in the less significant register specified. If four AR's are specified, add the storage operand specified by m to the contents of AR's 8 and 4 and store the sum in AR's 2 and 1. The Processor assumes that the operands are in binary format.

BINARY SUBTRACT HIGHER – BSH

If two AR's are specified, subtract the contents of the storage location specified from the contents of the more significant register specified and store the difference in the less significant register specified. If four AR's are specified, subtract the storage operand specified by m from the contents of AR's 8 and 4 and store the sum in AR's 2 and 1. The Processor assumes that the operands are in binary format.

3. Student Exercises

   (1) Two quantities are stored in locations 800 and 801. Interchange the quantities.

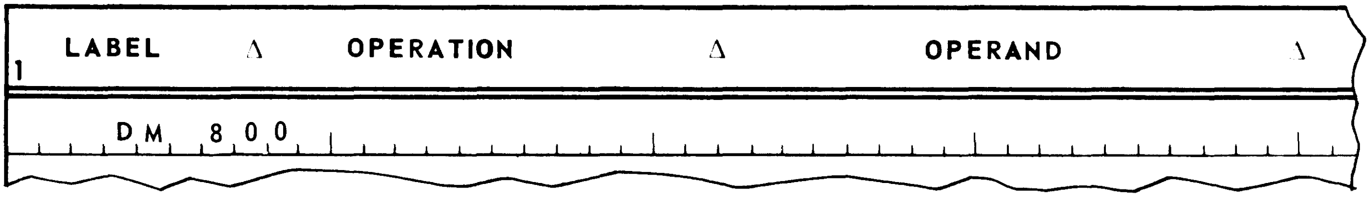
   (2) Quantity A is stored in locations 800 and 801, quantity B in locations 802 and 803, and quantity C in 804 and 805. All quantities are in decimal format. Compute A + B and store the sum in 806 and 807. Compute A + C and store the sum in 808 and 809.

E. MULTIPLICATION AND DIVISION

DECIMAL MULTIPLY - DM

Multiply the contents of the storage location specified by the contents of arithmetic register 8 to produce a 12 digit product. Store the six most significant digits of the product in arithmetic register 4 and the six least significant digits in arithmetic register 2. Store the sign of the product in the sign position of both arithmetic register 4 and arithmetic register 2. The Processor assumes that the operands are in decimal format. In the operands, a decimal digit with a bit combination of 0000 will be treated as such. It will not be treated as a decimal zero. The programmer has no choice as to which arithmetic registers to use. Arithmetic register 8 is always used to hold the operand, and arithmetic registers 4 and 2 to receive the product. As a consequence, in the UTMOST language, no AR portion need be specified in the instruction.

Example:



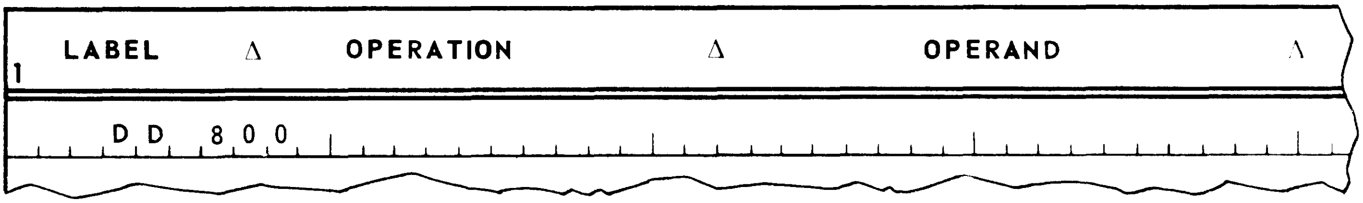
1 LABEL Δ	OPERATION Δ	OPERAND Δ
	DM 800	

(AR8) i = + 000600	(AR8) f = + 000600
(AR4) i = – 123456	(AR4) f = + 000002
(AR2) i = + 987654	(AR2) f = + 400000
(800) i = + 004000	(800) f = + 004000

DECIMAL DIVIDE - DD

Divide a 12 digit dividend in arithmetic registers 8 and 4 by the contents of the storage location specified to produce a six digit quotient in arithmetic register 4 and a remainder in arithmetic register 8. The sign of the remainder will be the same as the sign of the contents of arithmetic register 4 before instruction execution. The Processor assumes that the operands are in decimal format. In the operands, a decimal digit with a bit combination of 0000 will be treated as such. The programmer has no choice as to which arithmetic registers to use. As a consequence, in the UTMOST language, no AR portion need be specified in the instruction.

Example:



1 LABEL Δ	OPERATION Δ	OPERAND Δ
	DD 800	

(AR8) i = + 060000	(AR8) f = – 100000
(AR4) i = – 010000	(AR4) f = – 300000
(800) i = + 200000	(800) f = + 200000

F. THE DECIMAL POINT

The Processor has been so designed that, in all arithmetic operations, the decimal point of each operand is considered to be immediately to the left of the most significant digit of the operand. Consequently, so far as the Processor is concerned, the value of all operands lies between plus one and minus one, and are, consequently, fractional.

To represent quantities of greater or lesser magnitude than recognized by the Processor, the programmer must mentally assign the decimal point to a position other than that fixed by the Processor. This assumed decimal point is called the *program* decimal point, and in this manual, is indicated by a carat. Thus, the programmer may mentally assign a decimal point to a one word operand in decimal format as follows.

0060▲00

To the programmer this operand is the quantity 60.00. However, to the Processor it is the quantity .006000, and the Processor will treat it as such. Since the Processor ignores the program decimal point, a record of program decimal points must be maintained by the programmer. The position of the program decimal point in the results of an arithmetic operation can be determined by means of the following rules:

1. Rule for Addition and Subtraction

In adding or subtracting quantities in the Processor, the program decimal points must be lined up in both operands. The program decimal point in the result will be in the same position as in the operands entering the addition or subtraction.

Thus, the rule for handling the program decimal point in addition and subtraction is the same as the rule for handling the decimal point in pencil and paper addition and subtraction.

Example:

```
          PENCIL AND PAPER          UNIVAC III
             $3600.05                3600▲05
               156.23                0156▲23
             --------               --------
Sum          $3756.28                3756▲28
```

2. Rule for Multiplication

The Processor multiplies one 6 digit operand by another 6 digit operand to produce a 12 digit product. As with addition and subtraction, position of the program decimal point in the product is determined the same way placing the decimal point in pencil and paper multiplication is effected. The number of decimal places in the product is the sum of the decimal places in the multiplier and the multiplicand.

Example:

```
     PENCIL AND PAPER          UNIVAC III
           2.46                  0002▲46
           3.29                  0003▲29
        -------            ---------------
         8.0934             00000008▲0934
```

3. Rule for Division

Let M be the number of digit positions that the program decimal point is to the right or left of the Processor decimal point in the dividend. If the program decimal point is to the right of the Processor decimal point, M is positive; if to the left, M is negative. Let N be the number of digit positions that the program decimal point is to the right or left of the Processor decimal point in the divisor. If the program decimal point is to the right of the Processor decimal point N, is positive; if to the left N is negative. Then M-N is the number of digit positions that the program decimal point is to the right or left of the Processor decimal point in the quotient. If the result of M-N is positive, the program decimal point is to the right of the Processor decimal point in the quotient. If the result of M-N is negative the program decimal point is to the left of the Processor decimal point in the quotient.

Example: Divide 00063‸2497100 by 02‸0000

To the Processor this problem appears as follows: Divide .000632497100 by .020000. Thus, the Processor will come up with a quotient of .031624. In this case M is 5 and N is 2. Therefore, M-N is 3 and the quotient with the program decimal point is 031‸624.

To determine the program decimal point of the remainder it is necessary to consider it as being twelve digits, by adding 6 zeros to the left of the most significant digit position. Then the program decimal point would be located in the same position as it was in the twelve digit dividend.

In the case of the above example, the Processor would come up with a remainder of .017100 whereas according to the above, the program decimal point would be 00000‸0017100. As an example of how this may be used, assume that it is desired to verify the division, similar to the paper and pencil method, then:

Quotient	031‸624
Divisor	02‸0000
Partial Dividend	00063‸2480000
Remainder	00000‸0017100
Dividend	00063‸2497100

G. NONZERO DIGITS

It is sometimes important for the programmer to determine the maximum number of nonzero digits that may appear in the result of an arithmetic operation. The following rules are designed to make this determination.

1. Addition or Subtraction

If two operands are added, or if one operand is subtracted from another, the maximum number of nonzero digits in the sum or difference is one more than the number of nonzero digits in the operand having the greater number of nonzero digits. In the following example, nonzero digits are indicated by X's.

```
 0XX‸XX0
 00X‸XX0
 -------
 XXX‸XX0
```

2. Multiplication

The number of zeros before the first nonzero digit in the product is equal to the sum of the number of zeros before the first nonzero digit in the multiplier, added to the number of zeros before the first nonzero digit in the multiplicand. The maximum number of nonzero digits in the product is equal to the sum of the number of nonzero digits in the multiplier, added to the number of nonzero digits in the multiplicand. The remaining digits in the product are zeros.

Example:

$$C = 0XXX_{\blacktriangle}XX$$
$$D = 0X_{\blacktriangle}XX00$$
$$C \times D = 00XXXX_{\blacktriangle}XXXX00$$

3. Division

If, in the values just prior to dividing, u is the number of zeros before the first nonzero digit of the dividend and v is the number of zeros before the first nonzero digit of the divisor, then there will be a minimum of u minus v minus 1 zeros before the first nonzero digit of the quotient. All the other digits in the quotient may be nonzero. For example, given E and F.

$$E = 00XX_{\blacktriangle}XX$$
$$F = 0XX_{\blacktriangle}X00$$
$$E/F = X_{\blacktriangle}XXXXX$$

since u = 2 and v = 1

then u minus v minus 1 = 0

In this case, the remainder would have the same format as the divisor F:

$$0XX_{\blacktriangle}X00$$

If $u - v - 1 = -1$ this indicates that the most significant X will be a zero.

H. CONSTANTS

A constant is any UNIVAC III word (or words) which is neither executed as an instruction nor is a part of the data.

In solving a problem it is often necessary to use values that are not introduced with the data but are essential to the successful execution of a program. These values are established and written by the programmer at the time the program is written and, therefore, will be included with the instructions. Then at the time that the instructions are read into the Processor the constants necessary for the successful completion are also introduced.

Example:

Company X desires to give every employee a $100.00 bonus. The salary of each employee may be read into the Processor as a part of the data or may have been computed during a payroll run. Now the programmer desires to add $100.00 to this pay. He has at his disposal an ADD instruction to perform the addition but he does not have the value $100.00 as a part of the data. This value may be established as a constant by writing it on the coding paper and assigning it to a storage location that is not being used for the data or instructions. Since the constant is to be placed with the instructions but may not be executed as an instruction the following consideration must be given. UNIVAC III is a sequential processor and when control for execution has been given to the Processor it will continue to execute each instruction in sequence until something occurs to break this sequence. It is obvious that a constant may not be placed in direct line with this execution. Therefore, constants may be listed after a break in sequence with no fear of the control unit accessing them as instructions. Consideration must also be given by the programmer to the format of the constant.

In UTMOST language, constants are written in the following way. As is the case with instructions, the address of the storage location in which the constant is to be stored is left justified on the line on which the constant is to be written. This address is followed by one or more spaces. The next thing to appear on the line is a ''plus sign'' or a ''minus sign''. If the constant is to be positive, the ''plus sign'' is used; if negative, the ''minus sign''. The sign may or may not be followed by spaces at the programmer's option. There then follows the absolute value of the constant.

The programmer has the need to write three formats for constants: alphabetic, decimal and binary. If it is desired to write a constant word in six bit aphabetic format, the constant is written with characters and numbers and is surrounded by ''apostrophes''. In the Processor, one word holds four six bit characters. However, in UTMOST language it is not necessary to write any more characters than is desired. The UTMOST Assembler will take the characters enclosed in the ''apostrophes'', right justify them in the word into which they are to be stored, and fill the rest of the word with six bit space symbols (binary code 000000). The following are some examples of the operation of the UTMOST Assembler on alphabetic constants.

LOCATION OF CONSTANT	UTMOST LANGUAGE	BINARY CODE STORED	ALPHABETIC REPRESENTATION
0	+ 'ABCD'	0010100010101010110010111	+ ABCD
1	+ 'ABC '	0000000010100010101010110	+ \ABC
2	– 'A1'	1000000000000010100000100	– \ \A1

If it is desired to write a constant in four bit decimal format, the constant is written with decimal numbers and is preceded by a ''colon''. It is not necessary to write any more numbers than is desired. The UTMOST assembler will take the numbers between the colon and the first following space, right justify them, and fill the rest of the word with four bit space symbols (binary code 0000). The following are some examples of the operation of the Assembler on decimal constants.

LOCATION OF CONSTANT	UTMOST LANGUAGE	BINARY CODE STORED	DECIMAL REPRESENTATION
0	+ : 123456	0010001010110011110001001	+ 123456
1	+ : 123	0000000000000010001010110	+ \\\123
2	– : 14	1000000000000000001000111	– \\\\14

If it is desired to write a constant in binary format, the programmer may write the number in decimal or in octal. If written in decimal, it is written with decimal numbers alone. The most significant number may not be a zero. If written in octal, it is written with octal numbers preceded by a "zero". In both cases it is not necessary to write any more numbers than is desired. The binary equivalent of the numbers written will be right justified and preceded by binary zeros. The following are some examples of the operation of the Assembler on binary constants.

LOCATION OF CONSTANT	UTMOST LANGUAGE	BINARY CODE
0	+ 017	0000000000000000000001111
1	+ 07007	0000000000000111000000111
2	– 9	1000000000000000000001001
3	+ 1024	0000000000000010000000000

1. Example:

A dollar amount is stored in location 800 in format 0XX‸XX0. Add $25.74 to the amount.

a. Coding

LABEL Δ	OPERATION Δ	OPERAND Δ
0	LA 8,800	
1	DA 8,1023	
2	SA 8,800	
.		
.		
.		
1023	+ : 25740	

2. Student Exercises

(1) If A has the form ‸00XXXX, and B the form ‸0XXXXX, what is the form of AB?

(2) If A has the form 0XXX‸XX, and B the form XXX‸XX0, what is the form of AB?

(3) If A has the form 000XXXXX‸XXXX, and B the form 00XX‸XX, what is the form of A ÷ B?

(4) Three quantities of form + QQQQQQ are stored in locations 800, 801 and 802. Store the 18 digit product of the three quantities in locations 803, 804 and 805.

(5) Given the following:

DATA	FORM	LOCATION
Quantity A	▲0XX000	800
Quantity B	▲0XX000	801
Quantity C	▲0XX000	802
Quantity D	▲0XX000	803

$$E = AB$$

$$F = \frac{AB}{.9C}$$

$$G = \frac{AB}{.9C} - D$$

Quantity C has a value of .011 or greater. Store quantities E, F and G, respectively, in locations 804, 805 and 806.

(6)

DATA	FORM	LOCATION
Income	GGGGGG▲GG0000	800, 801,
Number of Dependents	PP▲0000	802
Deductions other than for Dependents	00AAAA▲AA0000	803, 804

A deduction of $600 is allowed for each dependent. The tax is 20% of the taxable income. Store the unrounded tax in form 0000TTTTTT▲TT in locations 805 and 806.

I. Branching

In certain operations, the next instruction to be executed is dependent of the nature of the data being processed. If, for example, a customer is to receive a discount only on orders of $10,000 or more, the billing procedure must consist of two different paths. One path bills the customer with a discount, the other bills him without a discount. Decision of which path to take for a particular customer depends on the amount of his order. The separation of the flow path of the sequence of instruction execution is called *branching*. Choice of which branch of instructions to take is determined by a *logical decision*. In this case, the logical decision is embodied in the question: Is the customer's order amount $10,000 or more? In the Processor, logical decisions are made on the basis of comparisons.

1. Comparisons

In the Processor, comparison is made between two operands. The results of a comparision is reflected in the resulting condition of *indicators*. An indicator has two states: on and off. There are three comparison indicators: high, low and equal. The first step in the execution of a comparison instruction is to set all three indicators to off. The comparison between one operand (in the arithmetic registers) and the other (in storage) is then made. If the two operands are equal, the equal indicator is turned on. If the operand in the arithmetic unit is larger than the operand in storage, the high indicator isturned on. If the operand in the arithmetic unit is smaller, the low indicator is turned on. Once a comparison has been made, the indicators remain in the state resulting from the comparison until one of the following occurs:

- Another comparison is made.

- An addition or subtraction is made. (A zero sum or difference turns the equal indicator on. A nonzero sum or difference turns the equal indicator off.)

2. The Collation Sequence of Characters

There is no question about the meaning of the equal indicator being turned on as the result of a comparison. Nor is the result ambiguous when the high or low indicator is turned on as the result of comparing binary or decimal operands. However, some question may arise as to what a high or low indicator may mean as a result of a comparison of alphabetic operands.

There is an arithmetic relation of relative magnitude with respect to numbers. Thus, two is larger than one, three is larger than two, and so on. This relation is called the *collation sequence* of numbers.

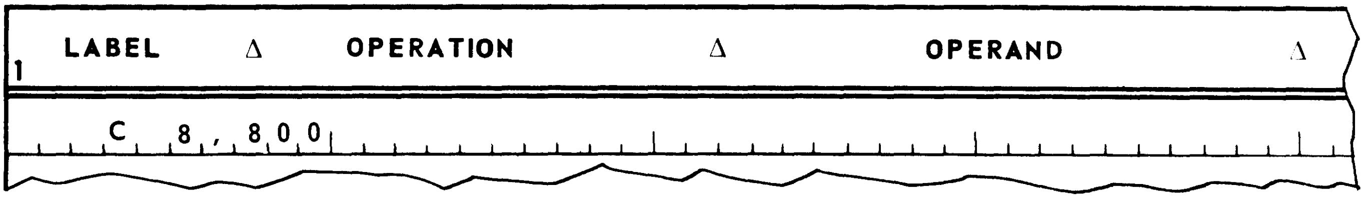
For purposes of comparing two alphabetic operands for relative magnitude, the Processor recognizes a collation sequence of characters. This collation sequence is as follows. If the characters are read off of Figure 2-18 by reading down the first column, then down the second, then down the third, and finally down the fourth, the characters are being read from smallest in magnitude to largest in magnitude. Thus, "A" is larger than "3". "Q" is larger than "K", "U" is larger than " P", and so on.

3. Comparison Instructions

COMPARE – C

Compare the operand specified by AR with the operand specified by m and turn the appropriate indicator on. This comparison takes into consideration the signs of the operands and is, consequently, an algebraic comparison. This instruction allows the use of multiword operands.

Example:



```
1  LABEL      Δ    OPERATION          Δ         OPERAND          Δ
      C   8, 800
```

If: (AR8) = + AAAAAA

(800) = + 666666

The high indicator is turned on.

If: (AR8) = + AAAAAA

(800) = – 666666

The high indicator is turned on.

If: (AR8) = – AAAAAA

(800) = – 666666

The low indicator is turned on.

COMPARE MAGNITUDE - CM

Compare the operand specified by AR with the operand specified by m and turn the appropriate indicator on. The signs of the operands are not taken into consideration, and consequently, this is a comparison of absolute values. This instruction allows the use of multiword operands.

Examples:

1 LABEL	Δ	OPERATION	Δ	OPERAND	Δ
		C		8,800	

If: (AR8) = + AAAAAA

(800) = + 666666

The high indicator is turned on.

If: (AR8) = + AAAAAA

(800) = – 666666

The high indicator is turned on.

If: (AR8) = – AAAAAA

(800) = – 666666

The high indicator is turned on.

COMPARE PRODUCT WITH A - CPA

If the operand specified by m has a one in every bit position where the operand specified by AR has a one, the equal indicator is turned on. Otherwise, the high indicator is turned on. This instruction allows the use of multiword operands.

Examples:

1 LABEL	Δ	OPERATION	Δ	OPERAND	Δ
		CPA		8,800	

If: (AR8) = 0100100000000000000000000

(800) = 0101100000000000000000000

The equal indicator is turned on.

If: (AR8) = 0100100000000000000000000

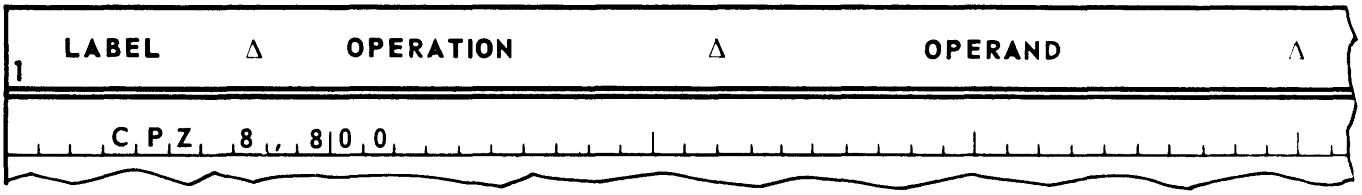
(800) = 01110000000000000000000000

The high indicator is turned on.

COMPARE PRODUCT WITH ZERO - CPZ

If the operand specified by m has a zero in every bit position where the operand specified by AR has a one, the equal indicator is turned on. Otherwise, the high indicator is turned on. This instruction allows the use of multiword operands.

Examples:



```
1  LABEL    Δ    OPERATION         Δ          OPERAND        Δ
      CPZ  8,800
```

If: (AR8) = 110100000000000000000000

(800) = 000010000000000000000000

The equal indicator is turned on.

If: (AR8) = 110100000000000000000000

(800) = 110010000000000000000000

The high indicator is turned on.

4. Transfer of Control

The Processor normally executes instructions sequentially. That is, after the instruction in storage location c is executed, the processor normally executes the instruction in storage location c + *1*. To effect branching, this normal sequence must be broken. The sequence is broken by means of a *transfer of control* instruction. For example, if the instruction in storage location c is being executed, and this instruction is an unconditional transfer of control instruction, the next instruction to be executed is found, not in storage location c + *1*, but in the storage location specified in the m portion of the transfer of control instruction.

For brevity, transfer of control instructions are called *jump* instructions, since they "jump" the Processor out of the normal sequence of instruction execution to a new sequence. Once the jump has been effected, normal sequential execution of instructions resumes.

JUMP – J

The next instruction to be executed is to be found in the storage location specified by m.

Example:

LOCATION OF INSTRUCTION	INSTRUCTION
4	J 10

Normally, the next instruction would be found in storage location 5. Execution of the jump instruction causes the next instruction to be found in storage location 10.

Notice that the jump instruction has no entry in the AR portion. Consequently, in the UTMOST language no AR portion is written.

The jump instruction is an *unconditional* transfer of control. Control is transferred regardless of any conditions present. As a result, it cannot be used for branching. The following instructions are *conditional* transfers of control. They jump only if some specified condition is met. Consequently, they, together with the comparison instructions, provide the means to effect branching.

JUMP EQUAL – JE

If the equal indicator is turned on, jump to m. That is, if the equal indicator is on, the next instruction to be executed is to be found in the storage location specified by m. If the equal indicator is off, normal sequential execution of instructions continues. In writing the instruction in UTMOST language, no AR portion is specified.

JUMP GREATER – JG

If the high indicator is on, jump to m. In writing the instruction, no AR portion is specified.

JUMP LESS – JL

If the low indicator is on, jump to m. No AR portion is specified.

JUMP POSITIVE – JP

If the contents of the arithmetic register specified are positive, jump to m.

NO OPERATION – NOP

This instruction does nothing. The Processor just goes to the next storage location in sequence to select the next instruction. Although, the NOP instruction involves neither AR or m, in the UTMOST language it must have an m portion. For example:

```
NOP 0
```

Uses of the NOP instruction will become clear later in this manual.

5. Example:

DATA	FORM	LOCATION
Account Number	0AAAAA	800
Delinquent Account Number	0DDDDD	801

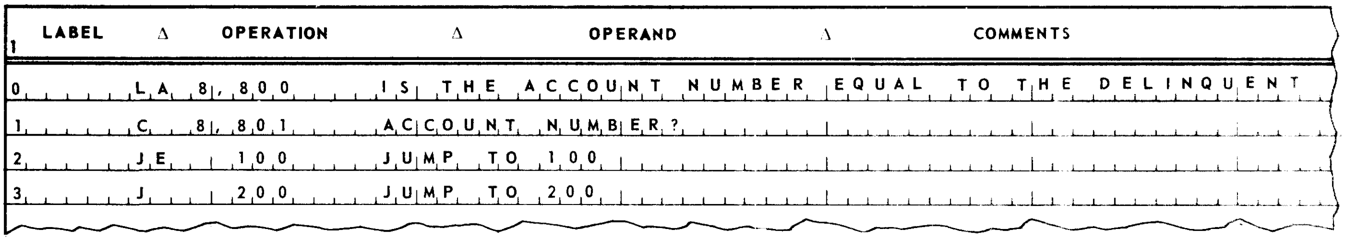
If the account number is equal to the delinquent account number, jump to storage location 100. If not, jump to location 200.

a. Logical Analysis

1. Is the account number equal to the delinquent account number?

1a. No	1b. Yes.
2. Jump to 200.	2. Jump to 100.

b. Coding



```
LABEL   Δ   OPERATION         Δ       OPERAND        Δ            COMMENTS
            LA  8, 800            IS THE ACCOUNT NUMBER EQUAL TO THE DELINQUENT
            C   8, 801            ACCOUNT NUMBER?
            JE     100            JUMP TO 100
            J      200            JUMP TO 200
```

6. Student Exercises

(1) If the absolute value of the contents of storage location 800 are less than the absolute value of the contents of location 801, add the contents of 802 to the contents of 803 and jump to 100. Otherwise, subtract the contents of 804 from the contents of 803 and jump to 200.

(2) If bit positions 12, 14 and 16 of the contents of storage location 800 are ones and bit positions 8, 9 and 10 are zeros, add the contents of location 801 to the contents of 802, store the sum in 803, and jump to 100. Otherwise, subtract the contents of 802 from the contents of 801, store the difference in 804, and jump to 200.

(3)

DATA	FORM	LOCATION
Pay	PPPP‸PP	800
Deduction	00DD‸DD	801

If the deduction will not reduce the pay below $15.00, make the deduction. Otherwise, store the deduction in storage location 802. In any case, store the pay to be received by the employee in location 803. When finished, jump to location 100.

2C. INTRODUCTION TO FLOWCHARTING

The subject of flowcharting may be best presented by means of example.

Example:

DATA	FORM	LOCATION
Days of Medical Absence	AA▲0000	800
Remaining Days of Medical Leave	LL▲0000	801
Hourly Rate of Pay	R▲RR000	802

Update the medical leave and store the medical pay in form PPPP▲PP in storage location 803. Then jump to location 100.

The first step in the solution of the above programming exercise is to make a logical analysis of the problem. The logical analysis might take the following form.

```
1.  Is medical absence equal to zero?
-------------------------------------------------------------------------------
1a. No.                                                          | 1b. Yes.
-------------------------------------------------------------------------------
2.  Is medical leave equal to zero?                              |
-----------------------------------------------------------------|
2a. No.                                             | 2b. Yes.   |
-----------------------------------------------------------------
3.  Is medical leave greater than medical absence?  |
----------------------------------------------------|
3a. No.                        | 3b. Yes.           |
4.  Store medical leave in     | 4.  Store medical  |
    storage.                   |     absence in     |
                               |     storage.       |
5.  Store zero in medical      | 5.  Reduce medical |
    leave.                     |     leave by       |
                               |     medical        |
                               |     absence.       |
6.  Multiply storage by eight. |                    |
7.  Multiply product by rate.  |                    |
8.  Store product in pay.      |                    | 8.  Store zero in pay.
9.  Jump to 100.               |                    |
```

While correct, the above analysis is bulky and unwieldy. Consequently, when developing a logical analysis, the programmer uses a different form of notation called *flowcharting*, and the form which his logical analysis takes in this notation is known as a *flowchart*.

Flowcharts differ from logical analyses in several respects. For one thing, the steps in a flowchart are typically shown in boxes, and arrows are used to indicate the sequence of steps. For example, the above logical analysis would be modified to look like the flowchart in Figure 2-21.

Notice that on those boxes in Figure 2-21 which have more than one arrow emerging from the box, the condition under which each path is taken is indicated on the arrow symbolizing the path. For example, the second box in Figure 2-21 represents a logical decision and has two paths emerging from it, one to be taken if the condition being tested for is met, and the other to be taken when the condition is not met. The two paths are labelled appropriately.

Programmers further reduce the bulkiness of their flowcharts by using symbols to represent fields, operations and conditions. Thus, the medical absence field might be represented by an “A”, the medical leave field by an “L”, the pay rate by an “R”, storage by an “S”, and the pay by a “P”. Many algebraic symbols are borrowed from mathematics to represent operations and conditions. Thus. “+” represents addition, “ – ” subtraction, “ x ” multiplication, “ ÷ ” division, “ = ” equal to, “ ≠ ” not equal to, “ > ” greater than, “ < ” less than, “ ≥ ” greater than or equal to, and “≤” less than or equal to. The operation of storing one field in another field (such as storing the medical absence in storage) is represented by an arrow. For example, the operation “store medical absence in storage” would be represented as:

$$A \rightarrow S$$

An arrow is also used to fill out an arithmetic operation. For example, the operation “reduce medical leave by medical absence” would be represented as:

$$L - A \rightarrow L$$

Where the arrow indicates that the new “L” is constituted by the difference between “A” subtracted from the old “L”.

The operation of comparing one field with another is represented by a colon. For example, the operation “is medical leave greater than medical absence” is a comparison of medical leave and medical absence. Consequently, it would be represented as:

$$L : A$$

By convention, logical decisions are shown on flowcharts in the form of diamonds rather than rectangles. Such a “logical decision box” always has at least two arrows emerging from it, each arrow being marked by the condition that must hold for the path to be taken.

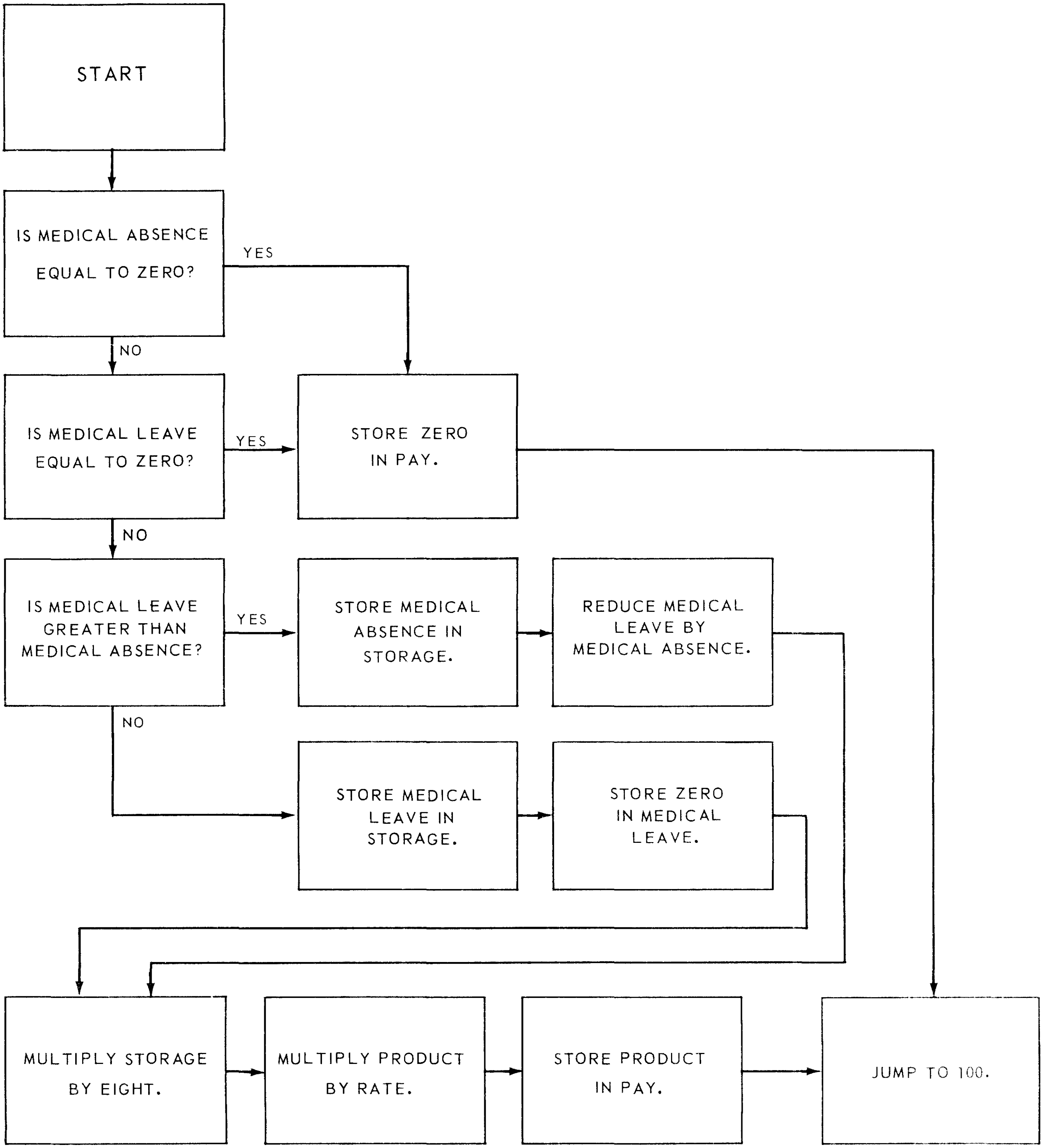

START
IS MEDICAL ABSENCE EQUAL TO ZERO?
YES
NO
IS MEDICAL LEAVE EQUAL TO ZERO?
YES
STORE ZERO IN PAY.
NO
IS MEDICAL LEAVE GREATER THAN MEDICAL ABSENCE?
YES
STORE MEDICAL ABSENCE IN STORAGE.
REDUCE MEDICAL LEAVE BY MEDICAL ABSENCE.
NO
STORE MEDICAL LEAVE IN STORAGE.
STORE ZERO IN MEDICAL LEAVE.
MULTIPLY STORAGE BY EIGHT.
MULTIPLY PRODUCT BY RATE.
STORE PRODUCT IN PAY.
JUMP TO 100.

Figure 2-21. Flowchart Incorporating Boxes and Arrows.

Adopting the conventions described above, the flowchart shown in Figure 2-21 would be modified to look like the flowchart in Figure 2-22.

Notice that the latter flowchart contains a legend which defines the arbitrary symbols used in the flowchart. Such a legend is always necessary in order to make a flow chart incorporating symbols legible.

To make their flowcharts even more compact, programmers make use of a special symbol, called a *connector*, to eliminate the long arrows that otherwise crisscross the flowchart to show the logical line of flow. A connector is a numbered circle. When, in a flowchart, an arrow leads to a connector, the next operation in the logical line of flow follows the arrow leading out of the connector containing the same number. Using connectors, the flowchart shown in Figure 2-22 would be modified to look like that in Figure 2-23.

Notice that, to distinguish between different connectors, different numbers are used. Notice also that a connector containing a given number and having an arrow leading into it can appear in a flowchart as many times as is necessary, but to avoid ambiguity, only one connector containing the number and having an arrow leading out of it can appear.

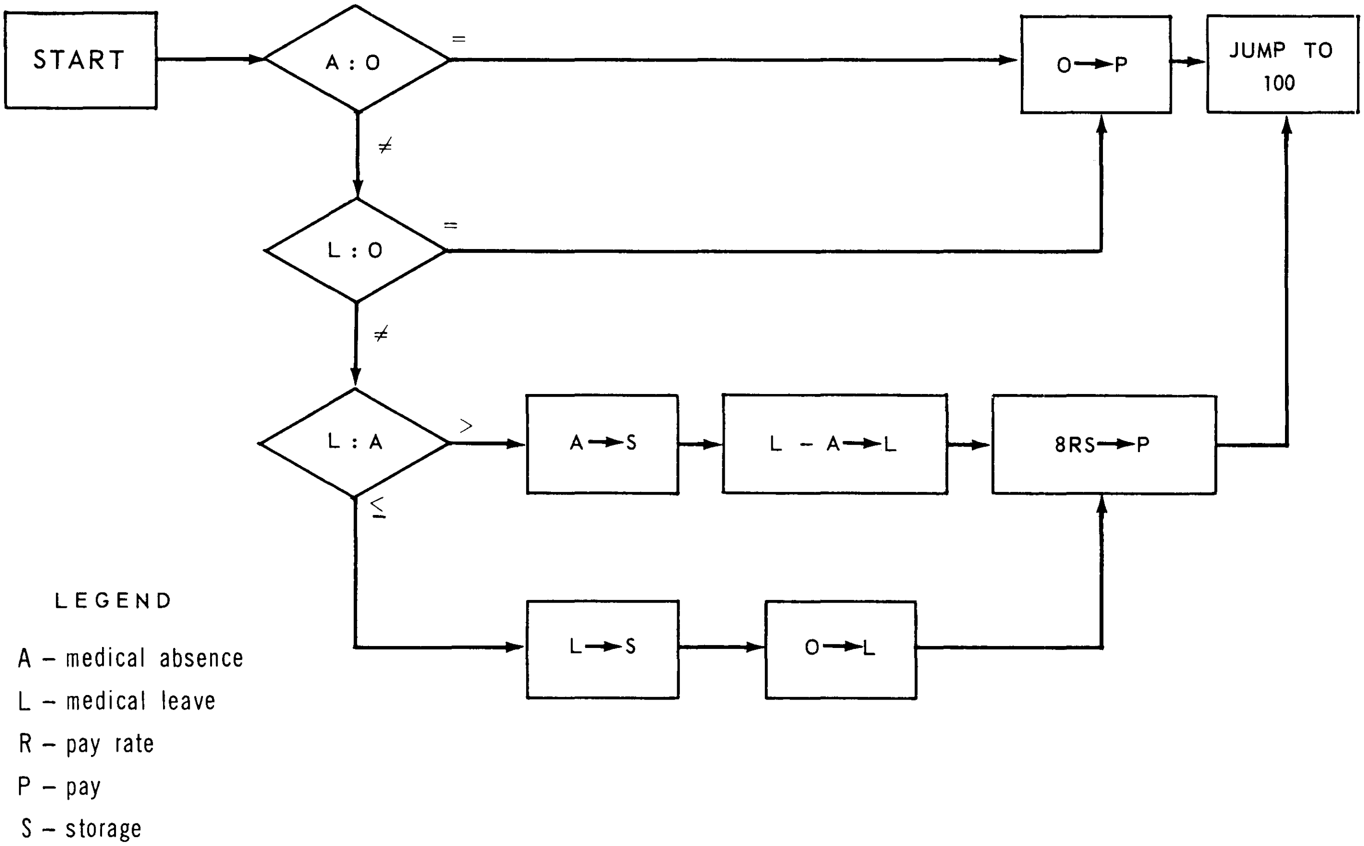


Figure 2-22. Flowchart Incorporating Symbols

In the flowchart in Figure 2-23, the "START" legend is shown in an oval. This is conventional.

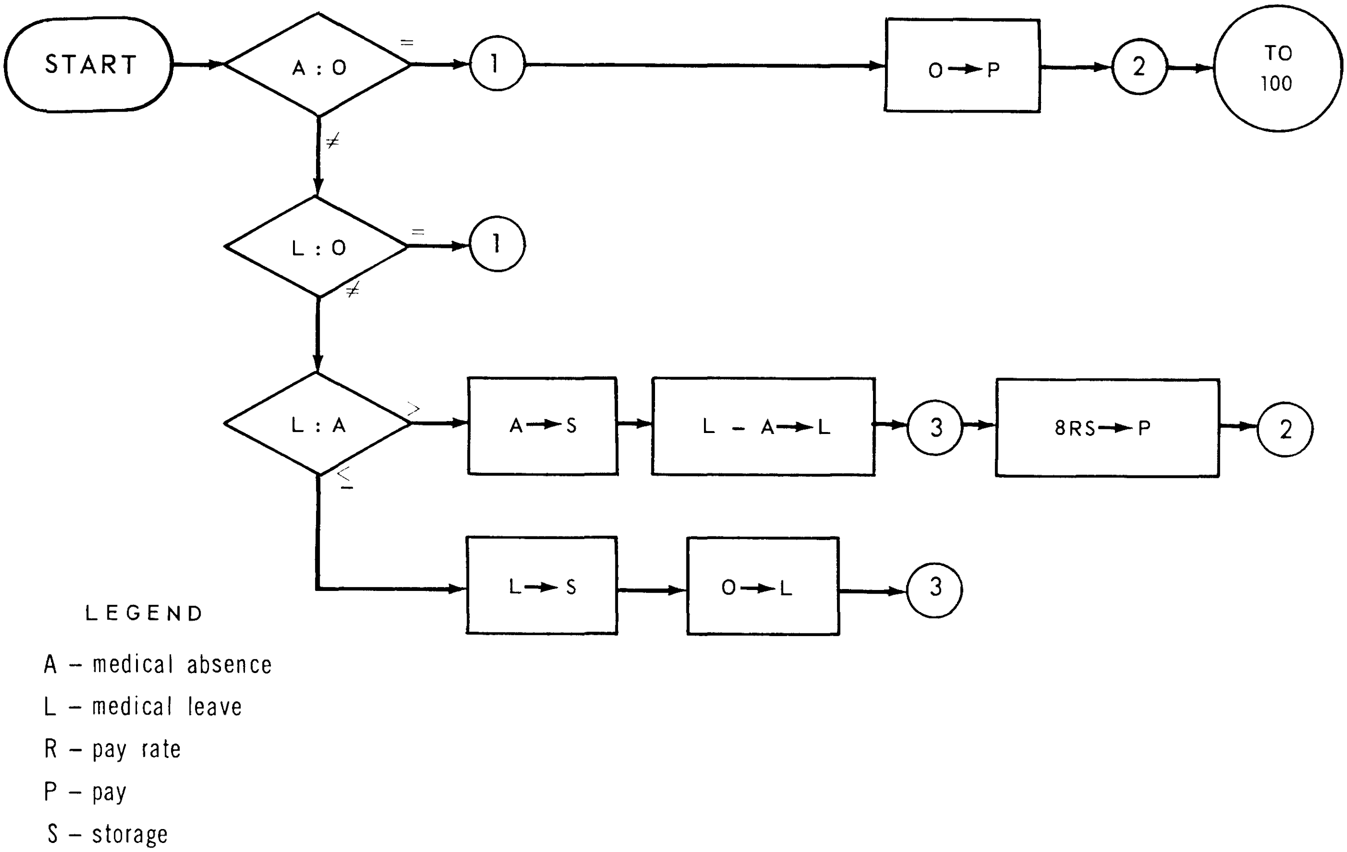


Figure 2-23. Flowchart Incorporating Connectors

Figure 2-23 shows the flowchart as a programmer might have originally produced it. Programmers and installations vary as to the style of flowchart produced. Any cross between the flowchart shown in Figure 2-23 and the one shown in Figure 2-21 is possible.

The coding for the above exercise is shown on the following page.

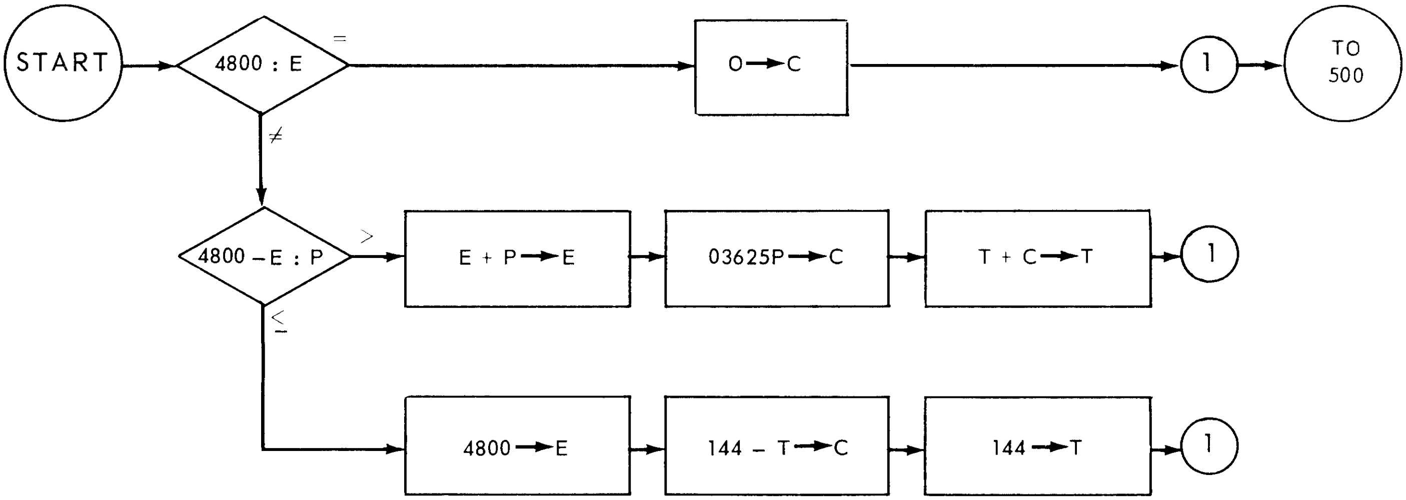
```
1  LABEL     Δ   OPERATION           Δ        OPERAND           Δ

0          LA   8,800       A:0
1          C    8,1023
2          JE     19
3          LA   8,801       L:0
4          C    8,1023
5          JE     19
6          C    8,800       L:A
7          JG     16
8          LA   4,1023      0---L
9          SA   4,801
10         DM     802       8RS---P
11         SA   4,1022
12         LA   8,1022
13         DM     1021
14         SA   4,803
15         J      100       TO 100
16         DSH 12,800       L-A---L
17         LA   8,800       A---S
18         J      9
19         LA   8,1023      0---P
20         ST   8,803
21         J      100       TO 100
.
.
.
1021       +:800000
1022       +0
1023       +:000000
```

A. EXAMPLE

DATA	FORM	LOCATION
YTD FICA Earnings	EEEE▲EE	880
YTD FICA Tax	OTTT▲TT	881
Current Pay	PPPP▲PP	882

Update the year to date FICA earnings and tax, and store the current FICA tax in form OOCC▲CC in location 883, Then jump to 500.

B. FLOWCHART



LEGEND

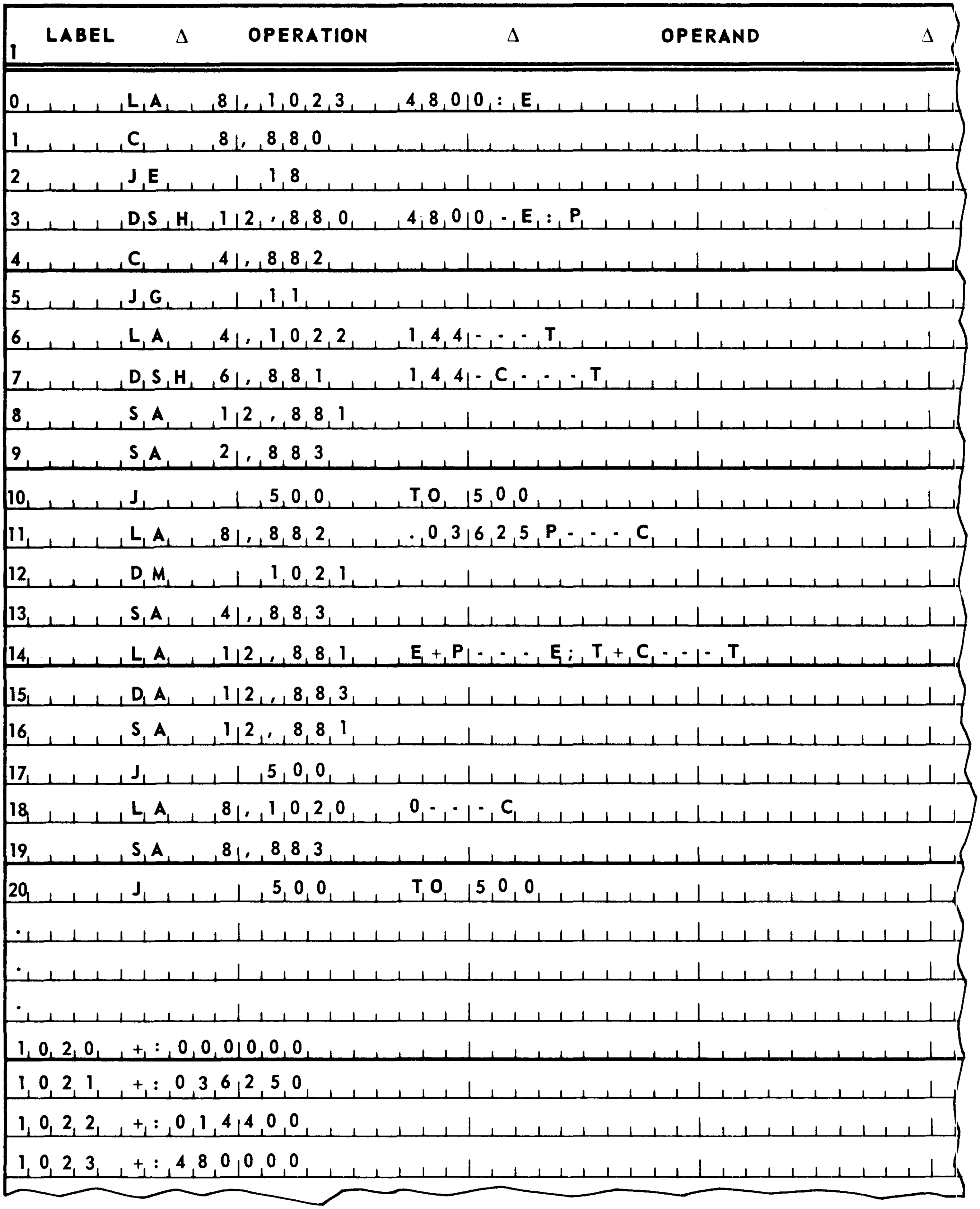
E – YTD FICA earnings

T – YTD FICA tax

P – current pay

C – current tax

C. CODING



```
LABEL     OPERATION                          OPERAND
1
0         LA     8, 1023       4800:E
1         C      8, 880
2         JE        18
3         DSH   12, 880        4800-E:P
4         C      4, 882
5         JG        11
6         LA     4, 1022       144---T
7         DSH    6, 881        144-C---T
8         SA    12, 881
9         SA     2, 883
10        J        500         TO 500
11        LA     8, 882        .03625P---C
12        DM      1021
13        SA     4, 883
14        LA    12, 881        E+P---E;T+C---T
15        DA    12, 883
16        SA    12, 881
17        J        500
18        LA     8, 1020       0---C
19        SA     8, 883
20        J        500         TO 500
.
.
.
1020   +:000000
1021   +:036250
1022   +:014400
1023   +:480000
```

D. STUDENT EXERCISES

1.

DATA	FORM	LOCATION
Quantity A	± AAAAAA▲	880
Quantity B	± BBBBBB▲	881
Quantity C	± CCCCCC▲	882

Store the smallest of the three quantities in storage location 883. Then jump to location 500.

2.

DATA	FORM	LOCATION
Badge Number	NNNNNN	880
Bond Deduction	OODD▲DD	881
Cumulative Bond Deduction	OCCC▲CC	882
Bond Price	OPPP▲PP	883

Update the cumulative bond deduction, and if a bond can be purchased, store the badge number in storage location 844 and the bond price in location 885. Then jump to 500

3. EDITING

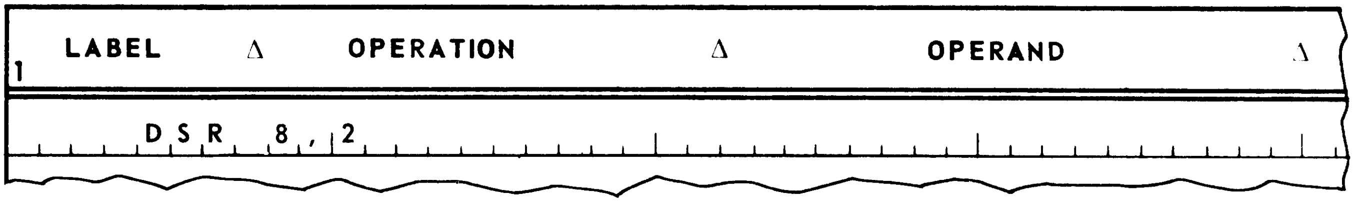
Fields of data fed into the Processor through the input units and put out by the Processor through output units may vary widely in form and content. A word may contain more than one field of information. To operate on one of these fields it may be necessary to isolate it from the other fields in a word. Two fields to be added together may not have their program decimal points lined up. To add the two fields it is then necessary to shift one or both of the fields to line up the decimal points prior to addition. Such field manipulation is accomplished through the use of editing instructions.

A. SHIFT INSTRUCTIONS

DECIMAL SHIFT RIGHT – DSR

Shift right the contents of the arithmetic register(s) specified the number of digit positions specified in m. Signs are not shifted. Digits shifted outside the register(s)' capacity are dropped. Decimal zeros are inserted in the vacated digit positions. The contents of one or two arithmetic registers may be shifted.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

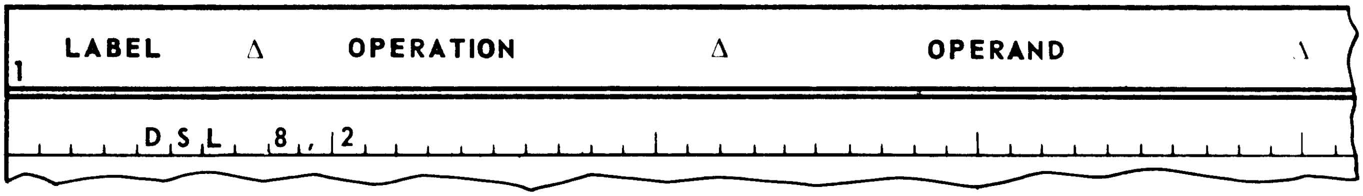
DSR 8,2

(AR8) i = + 123456 (AR8) f = + 001234

DECIMAL SHIFT LEFT – DSL

Shift left the contents of the arithmetic register(s) specified the number of digit positions specified in m. Signs are not shifted. Digits shifted outside the register(s)' capacity are dropped. Decimal zeros are inserted in the vacated digit positions. The contents of one or two arithmetic registers may be shifted.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

DSL 8,2

(AR8) i = + 123456 (AR8) f = + 345600

The decimal shift instructions treat operands in four bit groups. In four bits, sixteen different codes can be represented, as follows.

CODE	DECIMAL DIGIT
0000	
0001	
0010	
0011	0
0100	1
0101	2
0110	3
0111	4
1000	5
1001	6
1010	7
1011	8
1100	9
1101	
1110	
1111	

As indicated above, ten of these codes represent the ten decimal coefficients in excess three code. These ten codes retain their integrity during a decimal shift. The other six, however, change their nature as follows.

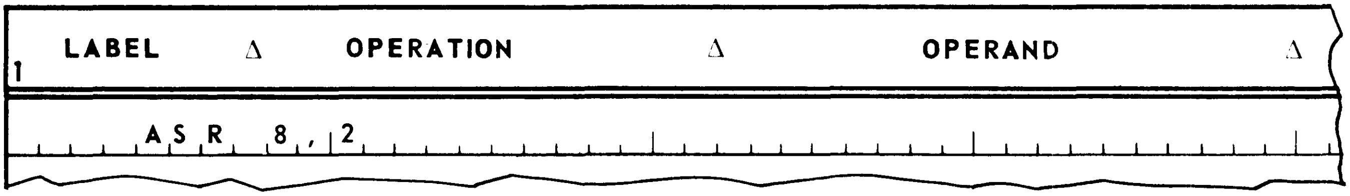
BEFORE	AFTER
0000	0011
0001	0011
0010	0011
1101	0011 *
1110	0100 *
1111	0101 *

Also, those three codes with an asterisk produce an arithmetic carry into the digit position to the left when the shift is performed.

ALPHANUMERIC SHIFT RIGHT – ASR

Shift right the contents of the arithmetic register(s) specified the number of characters specified in m. Signs are not shifted. Characters shifted outside the register(s)' capacity are dropped. "Space" characters are inserted in the vacated digit positions. The contents of one or two arithmetic registers may be shifted.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

ASR 8,2

(AR8) i = + ABCD (AR8) f = + ΔΔAB

ALPHANUMERIC SHIFT LEFT - ASL

Shift left the contents of the arithmetic register(s) specified the number of characters specified in m Signs are not shifted. Characters shifted outside the register(s)' capacity are dropped. "Space" characters are inserted in the vacated digit positions. The contents of one or two arithmetic registers may be shifted.

Example:

```
1  LABEL    Δ    OPERATION        Δ        OPERAND        Δ
       ASL  8, 2
```

BINARY ROTATE RIGHT - BRR

Shift right the contents of the arithmetic register specified the number of bits specified in m. The sign is shifted. Bits shifted beyond the right band of the registers capacity "circulate" and are reinserted at the left. The contents of a maximum of one arithmetic register may be shifted.

Example:

```
1  LABEL    Δ    OPERATION        Δ        OPERAND        Δ
       BRR  8, 6
```

(AR8) i = 1111111111000000000000000

(AR8) f = 0000001111111111000000000

In the case of any shift instruction, if the number of positions to be shifted, which is specified in m, exceeds the number of positions in the operand, the result of the shift is unpredictable and useless. An example of such a useless instruction would be ASR 12, 9 instruction.

B. LOGICAL OPERATION INSTRUCTIONS

AND

For every bit position containing a zero in the operand specified by m place a zero in the corresponding bit position of the operand specified by AR. Multiword operands may be used. All 25 bit positions are examined.

Example:

LABEL	OPERATION	OPERAND
	AND	8,800

(AR8) i = 0111111111111110000000000

(800) i = 0000000011111110000011111

(AR8) f = 0000000011111110000000000

(800) f = 0000000011111110000011111

OR

For every bit position containing a one in the operand specified by m place a one in the corresponding bit position of the operand specified by AR. Multiword operands may be used. All 25 bit positions are examined.

Example:

LABEL	OPERATION	OPERAND
	OR	8,800

(AR8) i = 0111111111111110000000000

(800) i = 0000000011111110000011111

(AR8) f = 0111111111111110000011111

(800) f = 0000000011111110000011111

C. INDIRECT ADDRESSING

With the exception of the NOP instruction, all instructions introduced thus far in this manual have a meaningful m portion. In all these latter cases m has been defined as being used to specify either the location of an operand, as in the case of the LA instruction, or a shift count, as in the DSR instruction. This holds true as long as the contents of bit 25 of the instruction (the indirect address bit) contains a zero. If bit position 25 of the instruction contains a one instead of a zero, then m specifies, not an operand location or a shift count, but an address at which the operand location or shift count can be found. Thus, the instruction addresses the operand not directly, but indirectly. In this case the contents of the storage location specified by m is called the *indirect address control word.*

The format of the indirect address control word is as follows.

1. Bits 1-15, the l portion, contain a 15 bit address which corresponds to the m portion of an instruction without indirect addressing. Thus, it becomes the operand address or shift count. Notice that, since the l portion of the indirect address control word consists of 15 bits, indirect addressing provides one means of addressing any location in the store. A more general method of such addressing will be described later in this manual.

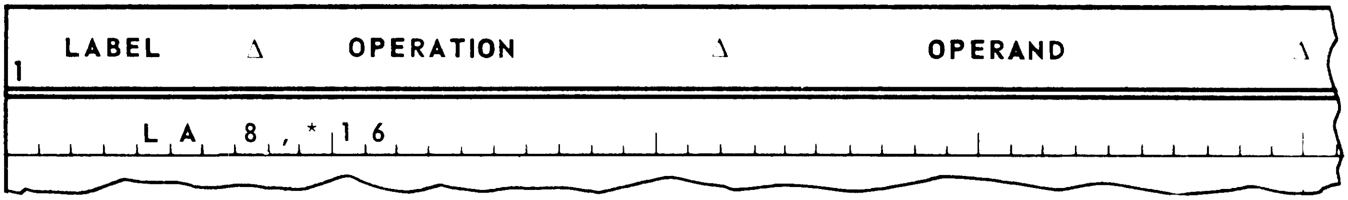
2. Bits 16 - 20 always contain binary zeros.

3. Bits 21 - 24 perform a function similar to that which these same bits perform in an instruction word. This function will be discussed later in this manual.

4. Bit 25 is the indirect address bit just as it is in an instruction word. Thus, if bit 25 is a zero, then l is an operand address or shift count. If bit 25 is one, then l also is an address at which the operand address or shift count can be found. This "cascading" of indirect addresses can be carried as far as the programmer desires.

In UTMOST language, an instruction that is to use indirect addressing is indicated by placing an asterisk immediately before the m portion of the instruction. An indirect address control word is written like a constant, that is, a plus or minus sign followed by the l portion of the indirect address control word. If l is preceded by a plus, then l is the operand address or shift count. If l is preceded by a minus then l also is an indirect address, and cascading results.

Example:



1
LABEL
Δ
OPERATION
Δ
OPERAND
Δ

```
    LA  8,*16
```

(16) = + 800

(AR8) i = + 123456   (AR8) f = + 987654

(800) i = + 987654   (800) f = + 987654

Indirect addressing may be used with any instruction in which the m portion of the instruction is significant.

D. FIELD SELECTION

The mechanism used to achieve indirect addressing is also used to achieve the operation of *field selection.* That is, if field selection is to occur during instruction execution, then bit 25 of the instruction word is to be a one. In UTMOST language, this means that the m portion of the instruction is to be preceded by an asterisk. Whether field selection or indirect addressing is to result is determined by the word found in the m address. If bits 16 - 20 of this word are binary zeros, then indirect addressing is to result. If any of these bits is a one, then field selection is to result. In this latter case, the word is referred to as a *field select control word.*

Like an indirect address control word, a field select control word contains the address of the operand. However, a field select control word also specifies certain bits within the operand specified by l. The function of field selection is to allow the Processor to operate on only those bits of the operand specified by the field select control word.

A field select control word has the following format.

1. Bits 1 - 10, the l portion, contain a 10 bit address which corresponds to the m portion of an instruction without indirect addressing. Thus, it becomes the operand address.

2. Bits 11 - 15 specify in XS3 code the rightmost bit within the operand that the instruction is to operate on.

3. Bits 16 - 20 specify in XS3 code the leftmost bit within the operand that the instruction is to operate on.

4. Bits 21 - 24 perform a function similar to that which these same bits perform in an instruction word or indirect address control word. This function will be discussed later in this manual.

5. Bit 25 must contain a zero.

In UTMOST language, a field select control word is written in the following way:

$$+ e_1, e_2, e_3$$

where:

- e_1 is the leftmost bit to be operated on and is written as a decimal number representing the number of the bit desired.

- e_2 is the rightmost bit to be operated on, and is written as a decimal number representing the number of the bit desired.

- e_3 is the l operand address, and is written in the usual way. In writing a field select control word, the programmer may optionally insert one or more spaces after the plus sign and each of the commas.

Bit positions outside those bits specified in the field select control word are considered as containing binary zeros. Moreover, sign bits cannot be included in a field select operation. Thus, all operands field selected become positive when operated on.

Example:

```
1  LABEL    Δ    OPERATION          Δ          OPERAND        \
         LA   8,  *16
```

(16) = + 18, 7, 800

(AR8) i = – ABCD (AR8) f = + ΔXYΔ

(800) i = – WXYZ (800) f = – WXYZ

Example.

```
1  LABEL    Δ    OPERATION          Δ          OPERAND        \
         LAN   8,  *16
```

(AR8) i = – ABCD (AR8) f = – ΔXYΔ

(800) i = WXYZ (16) = + 18, 7, 800

Note in the second example that the field selected operand comes from the store with a positive sign. The operation of the LAN instruction then changes this sign to minus. LAN with FS always produces a negative quantity.

Field selection has meaning when an operand is to be selected from the store to be operated on. Thus, field selection is not pertinent with respect to shift instructions, jump instructions, and store arithmetic register instructions. Also, it is not possible to use field selection with the multiply or divide instruction. Operation of field selection with respect to the load arithmetic register instructions is exemplified in the above illustrations.

When field selection is used with addition or subtraction instructions, the field selection occurs on the operand as it comes from the store. No field selection occurs on the operand coming from the arithmetic register(s).

Example:

```
1  LABEL    Δ    OPERATION          Δ          OPERAND        \
         DA   8,  *16
```

(16) = + 20, 5, 800

(AR8) i = + 123456 (AR8) f = + 223446

(800) i = – 999999 (800) f = – 999999

With respect to comparison instructions, field selection operates on both the operand from the store and the operand from the arithmetic registers. Also, with the C instruction, the sign of the AR operand also enters the comparison. Execution of all other comparison instructions with field selection ignores the sign bit.

```
1  LABEL     Δ     OPERATION            Δ             OPERAND             Δ
        C  8,*16
```

(16) = + 18, 7, 800

(AR8) = + ABCD

(800) = - YBCZ

After instruction execution, the equal indicator is turned on.

Example:

```
1  LABEL     Δ     OPERATION            Δ             OPERAND             Δ
        C  8,*16
```

(16) = + 18, 7, 800

(AR8) = - ABCD

(800) = - YBCZ

After instruction execution, the low indicator is turned on.

The logical instructions, AND and OR, use field selection in a similar way in that only the portion of the AR operand specified by the field selection is affected by instruction execution.

Example:

1 LABEL	Δ OPERATION	Δ OPERAND	\
	AND	8,*16	

(16) = + 21, 11, 800

(AR8) i = 0111111111111110001100000

(800) i = 0000000011111110000011111

(AR8) f = 0111000011111110001100000

(800) f = 0000000011111110000011111

When field selection is used with multiword operands, the rightmost bit of the field selected is to be found in the least significant word of the operand, the leftmost bit in the most significant word of the operand.

Example:

1 LABEL	Δ OPERATION	Δ OPERAND	\
	LA	12,*16	

(16) = + 18, 7, 801

(AR8) i = - ABCD (AR8) f = + ΔTUV

(AR4) i = - EFGH (AR4) f = + WXYΔ

(800) i = - STUV (800) f = - STUV

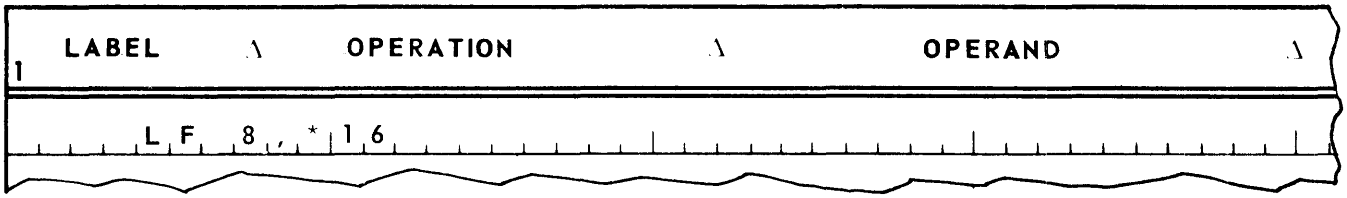
(801) i = - WXYZ (801) f = - WXYZ

E. THE LOAD FIELD INSTRUCTION

LOAD FIELD – LF

The LF instruction is similar to the LA instruction in that it causes the operand specified by m to be loaded into the arithmetic register(s) specified. Multiword operands and indirect addressing may be used with the LF instruction. The LF instruction differs from the LA instruction in the way it operates when field selection is specified. With the LA instruction, arithmetic register bit positions outside the field specified are set to zero. With the LF instruction, the contents of the arithmetic register bit positions outside of the field specified are undisturbed.

Example:



1
LABEL ∆ OPERATION ∆ OPERAND ∆

```
    LF  8,*16
```

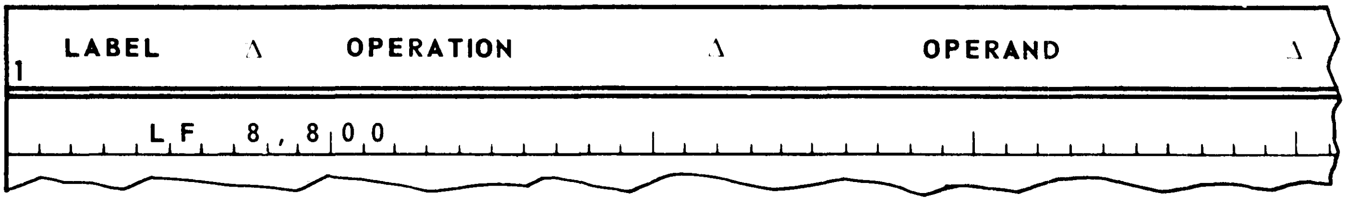
(16) = + 18, 7, 800

(AR8) i = - ABCD (AR8) f = - AXYD

(800) i = + WXYZ (800) f = + WXYZ

If the LF instruction is used without field selection, it operates in the same manner as the LA instruction except that the sign of the arithmetic register(s) is undisturbed.

Example:



1
LABEL ∆ OPERATION ∆ OPERAND ∆

```
    LF  8,800
```

(AR8) i = - ABCD (AR8) f = - WXYZ

(800) i = + WXYZ (800) f = + WXYZ

F. EXAMPLE

Given:

LOCATION	FORM
880	NNNNNN
881	0LLLLL▲
882	LLMMMM
883	M▲MMVVV
884	VV▲VVPP
885	PPPP▲PP

where:

N	is a job number
L	is the cost of labor for the job
M	is the material cost for the job
V	is the overhead cost
P	is the price the job is contracted for

Create the following

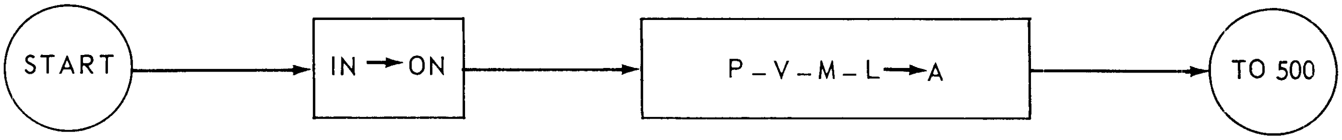
LOCATION	FORM
886	NNNNNN
887	AAAAAA▲
888	AA0000

where:

N	is the job number
A	is the profit for the job

When done, jump to storage location 500.

G. FLOWCHART



LEGEND

IN	–	the input job number
ON	–	the output job number
P	–	the contract price
V	–	the overhead cost
M	–	the material cost
L	–	the labor cost
A	–	the profit

H. CODING

LABEL	OPERATION	OPERAND	
0	LA	8, 880	IN---ON
1	SA	8, 886	
2	LA	12, *1023	P-V-M-L---A
3	DSL	12, 2	
4	DS	12, *1022	
5	DSL	12, 1	
6	DS	12, *1021	
7	DSL	12, 1	
8	DS	12, *1020	
9	SA	12 888	
10	J	500	
.			
.			
.			
1020	+	20,17,882	
1021	+	16,13,883	
1022	+	12,9,884	
1023	+	8,1,885	

4. INDEX REGISTERS

The user has the option of obtaining his Processor with either 9 or 15 index registers. Index registers are identified by number. Thus, there is X1, X2, X3, and so on, up through X15, where ''X'' is a commonly used abbreviation for ''index register''.

An index register has the capacity to store 16 bits, although for most purposes only the least significant 15 bits have meaning. Bit positions in an index register are numbered from right to left as 1 through 16. An index register has no sign bit position. The contents of an index register are always considered to be a positive binary number.

An instruction, indirect address control word, and field select control word address index registers by means of bit positions 21-24. The number of the index register to be addressed is represented in pure binary in these bit positions.

In UTMOST language an index register is addressed by placing a comma after the m portion of an instruction (or l portion of an indirect address control word or field select control word) and following the comma by the number of the index register to be addressed. This number is written in decimal. The programmer may optionally leave one or more spaces between the comma and the index register specification.

Index registers have the following function. When an instruction is to be executed, the contents of the index register specified are added to the m portion of the instruction. The sum of this addition is the address of the operand (or shift count), and is commonly referred to as m′ (m prime).

Example:

```
1  LABEL     Δ     OPERATION          Δ              OPERAND              Δ
         LA  8,6,15
```

	(X15) = 10000*
(AR8) i = + ABCD	(AR8) f = - WXYZ
(10006) i = - WXYZ	(10006) i = - WXYZ

If binary zeros are placed in the index register portion of an instruction, no indexing results. In UTMOST language, binary zeros in the index register portion of an instruction is indicated by not specifying an index register in the instruction. Thus, all instructions shown thus far in this manual (with the exception of the one in the last example) do not call for indexing.

Manipulation of the contents of index registers is achieved by means of the following instructions, which specify the index register whose contents is to be manipulated in the AR portion of the instruction.

* *Although written here in decimal for ease of presentation, the contents of index register 15 would actually be the binary equivalent of a decimal 10,000.*

LOAD INDEX REGISTER – LX

Load the contents of bits one through 15 of the storage location specified by m′ in the index register specified in AR. Multiword operands are meaningless with this instruction. Indirect addressing may be used. However, field selection is not allowed.

Example:

```
1  LABEL      Λ     OPERATION            Λ              OPERAND              Λ
         LX   15,800
```

(X15) i = 15000*   (X15) f = 20000*

(800) i = 20000*   (800) f = 20000*

STORE INDEX REGISTER – SX

Store the contents of the index register specified by AR in bits one through 16 of the storage location specified by m′. Store binary zeros in the other bit positions of location m′. Actually, bit position 16 of the index register will always contain a zero. Consequently, this instruction could be defined as follows.

> Store the contents of bits one through 15 of the index register specified by AR in bits one through 15 of the storage location specified by m′. Store binary zeros in the other bit positions of storage location m′.

Multiword operands and field selection are meaningless with this instruction. Indirect addressing may be used.

Example:

```
1  LABEL      Λ     OPERATION            Λ              OPERAND              Λ
         SX   15,800
```

(X15) i = 15000*   (X15) f = 15000*

(800) i = 20000*   (800) f = 15000*

If a Processor is equipped with nine index registers and index register 10 through 15 is specified in the AR portion of a SX instruction, binary ones will be stored in bit positions 1-16 of the storage location specified by m′. Binary zeros are stored in the other bit positions of storage location m′.

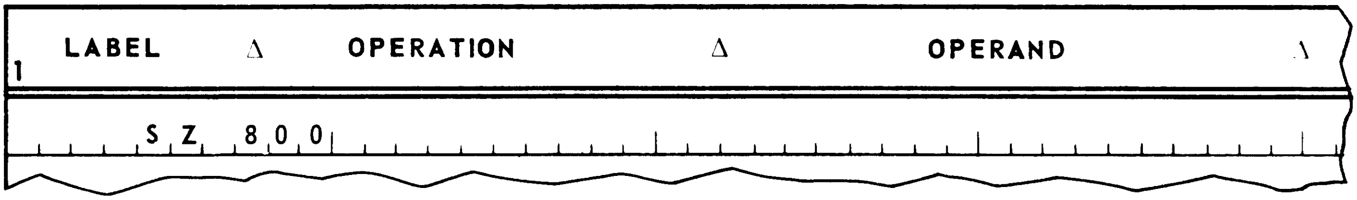
If binary zeros are placed in the AR portion of a SX instruction, binary zeros are stored in the storage location specified by m′. UTMOST language provides a special instruction for this operation.

* *Actually binary numbers. Moreover, only the 15 least significant bits of the contents of storage location 800 are shown.*

STORE ZEROS – SZ

Store binary zeros in the storage location specified by m′.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

SZ 800

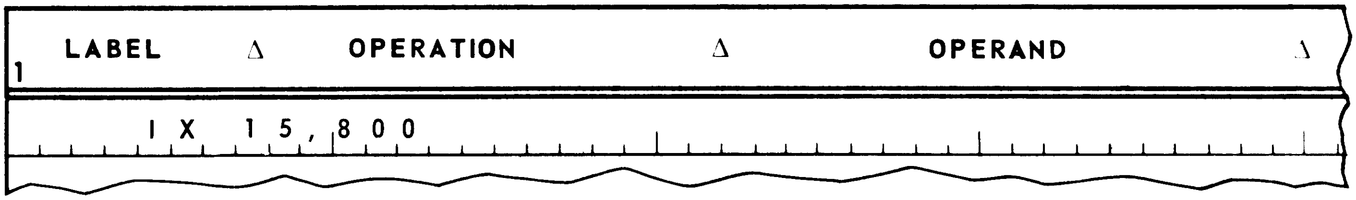
(800) i = – ABCD (800) f = + ΔΔΔΔ

Since the binary format of a NOP instruction is all binary zeros, the SZ instruction can be used to create a NOP instruction in storage location m′.

INCREMENT INDEX REGISTER – IX

Add, in binary, add the contents of bit positions one through nine of the storage location specified by m′ to the contents of the index register specified by AR. An algebraic addition which attends to the contents of the sign bit of storage location m′ is performed. Thus, if the sign of the contents of location m′ is positive, the contents of the index register specified are increased, or *incremented*. If negative, the contents of the index register are decreased, or *decremented*. If as a result of this addition, a carry is propogated from bit position 15 of the index register, this carry is dropped. Thus, the contents of bit position 16 of the index register always remains zero. Multiword operands are meaningless with this instruction. Indirect addressing may be used. However, field selection is not allowed.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

IX 15,800

(X15) i = 15000* (X15) f = 15010*

(800) i = + 10 (800) f = + 10

INCREMENT INDEX REGISTER AND COMPARE – IXC

Add, in binary, add the contents of bit positions one through nine of the storage location specified by m' to the contents of the index register specified by AR. The addition takes into consideration the contents of the sign bit of storage location m'. Carry from bit position 15 of the index register is inhibited. After the index register has been incremented, the contents of bits one through 15 of the index register are compared with the contents of bits 10 through 24 of location m'. If the two are equal, the equal indicator is turned on. If the contents of bits one through 15 of the index register are greater than the contents of bits 10 through 14 of m', the high indicator is turned on. Otherwise, the low indicator is turned on. Multiword operands and field selection are meaningless with this instruction. Indirect addressing may be used. If binary zeros are placed in the AR portion of an IXC instruction, no incrementation occurs, and for purposes of comparison the index register is considered to contain binary zeros. If a Processor is equipped with nine index registers and index register 10 through 15 is specified in the AR portion of an IXC instruction, no incrementation occurs, and for purposes of comparison the index register is considered to contain all binary ones.

The contents of the storage location specified by m' of an IXC instruction is called an *increment and compare word*. In UTMOST language, an increment and compare word is written as follows.

ICW e_1, e_2

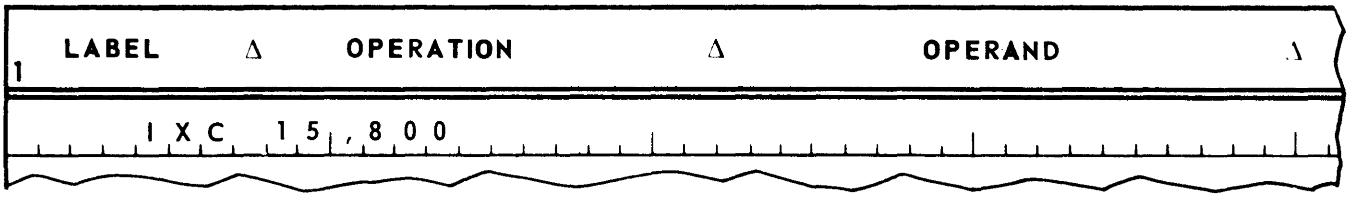
where:

1. e_1 is the comparison amount (bits 10 - 24) usually written in decimal.

2. e_2 is the increment amount (bits 1 - 9) usually written in decimal.

At least one space must be left between ICW and e_1. The programmer may optionally leave one or more spaces between the comma and e_2. If it is desired to decrement, e_2 is preceded by a minus sign.

Example:



1 LABEL	Δ	OPERATION	Δ	OPERAND	Δ
		IXC		15,800	

(X15) i = 15000* (X15) f = 15002*

(800) i = ICW 16000, 2 (800) f = ICW 16000, 2

At the end of instruction execution, the low indicator is turned on.

A. EXAMPLE

There are 100 delinquent account numbers stored in storage locations 400 through 499. If the new account number stored in location 500 is delinquent, jump to 300. Otherwise, jump to 350.

B. CODING

This problem could be coded as follows.

LABEL	OPERATION	OPERAND	COMMENTS
0	LA	8, 500	IS THE NEW ACCOUNT NUMBER EQUAL
1	C	8, 400	TO THE FIRST DELINQUENT ACCOUNT
2	JE	300	NUMBER?
3	C	8, 401	IS IT EQUAL TO THE SECOND
4	JE	300	DELINQUENT ACCOUNT NUMBER?
5	C	8, 402	IS IT EQUAL TO THE THIRD ONE?
6	JE	300	
.	.	. .	
.	.	. .	
.	.	. .	
199	C	8, 499	IS IT EQUAL TO THE LAST ONE?
200	JE	300	
201	J	350	JUMP TO 350.

This solution requires 202 lines of coding. Such an approach is referred to as *straight line* coding. Study of this coding may allow reduction of this number of lines.

Notice that the body of this coding consists of a repetition of two lines of the following form.

LABEL	OPERATION	OPERAND
	C	8, y
	JE	300

In each set of two lines, y is one more than it was last time. Thus, in the first set it is 400, in the second set 401, in the third 402, and so on. This observation leads to the conclusion that this is a natural situation for the use of index registers. The following coding employs index registers.

LABEL	OPERATION	OPERAND	COMMENTS
0	LX	15, 1023	TAKE THE FIRST DELINQUENT AC
1	LA	8, 500	
2	C	8, 400, 15	DOES THE NEW ACCOUNT NUMBER
3	JE	300	MATCH THIS DELINQUENT ACCOUNT
4	IX	15, 1022	TAKE THE NEXT DELINQUENT ACC
5	NOP	0	
6	J	2	LOOP.
.			
.			
.			
1022	+	1	
1023	+	0	

In this coding the instruction in line 0 loads X15 with binary zeros. Line 1 loads the new account number into AR8. The m portion of the instruction in line 2 is 400. This instruction specifies modification of m by the contents of X15. Since the contents of X15 are zero, m′ is 400. Thus, the new account number is compared against the first delinquent account number.

Line 3 tests for equality. If the two are not equal, line 4 increases the contents of X15 by one. Thus, X15 now contains one. Line 5 is a do nothing line, and line 6 returns control to line 2. Now m′ is 401, since the contents of X15 are one. Thus, the new account number is compared with the second delinquent account number. If these are not equal, the contents of X15 are again increased by one. On return of control to line 2, m′ is now 402, since the contents of X15 are now two. Thus, the new account number is compared with the third delinquent account number. And so on.

The above coding is known as *iterative* coding. The distinguishing characteristic of interative coding is that it processes many items with the same set of coding, which it modifies and loops through once for each item .

The above coding incorporates such an iterative loop. However, it is a "closed" loop. If the new account number is unequal to all 100 delinquent account numbers, this coding provides no way to exit from the loop after all 100 delinquent account numbers have been tested.

When all 100 delinquent account numbers have been tested, X15 will contain 99. If an IXC instruction is substituted for the IX in line 4, and if the comparison "element" of the associated ICW is set at 100, the equal indicator will be set after all 100 delinquent account numbers have been tested. The following coding incorporates this change.

```
1   LABEL   Δ   OPERATION        Δ        OPERAND        Δ        COMMENTS
0       LX      15,   1023                TAKE THE FIRST DELINQUENT AC
1       LA      8,    500
2       C       8,    400,15              DOES THE NEW ACCOUNT NUMBER
3       JE            300                 MATCH THIS DELINQUENT ACCOUNT
4       IXC     15,   1022                TAKE THE NEXT DELINQUENT ACC
5       JE            350                 JUMP TO 350.
6       J               2                 LOOP.
.
.
.
1022    ICW     100,1
1023    +       0
```

A slightly more efficient use of index registers to set up the iterative loop is shown in the following coding.

```
1   LABEL   Δ   OPERATION        Δ        OPERAND        Δ        COMMENTS
0       LX      15,   1023                TAKE THE FIRST DELINQUENT AC
1       LA      8,    500
2       C       8,    400,15              DOES THE NEW ACCOUNT NUMBER
3       JE            300                 MATCH THIS DELINQUENT ACCOUNT
4       IXC     15,   1022                TAKE THE NEXT DELINQUENT ACC
5       JL              2                 LOOP.
6       J             350                 JUMP TO 350.
.
.
.
1022    ICW     100,1
1023    +       0
```

C. FLOWCHART

An English language flowchart of the previous coding might look like that shown in Figure 4-1.

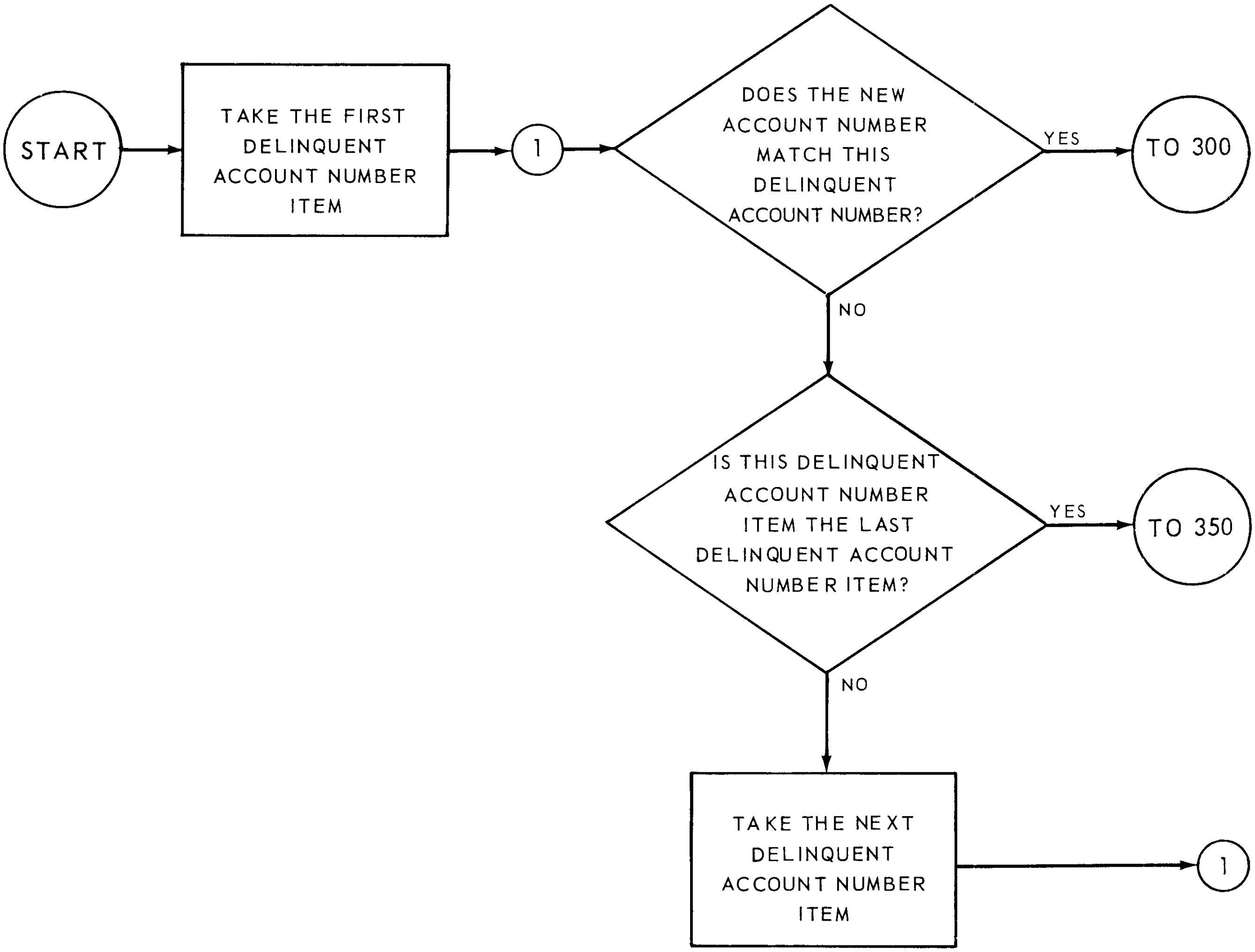

START
TAKE THE FIRST DELINQUENT ACCOUNT NUMBER ITEM
1
DOES THE NEW ACCOUNT NUMBER MATCH THIS DELINQUENT ACCOUNT NUMBER?
YES
TO 300
NO
IS THIS DELINQUENT ACCOUNT NUMBER ITEM THE LAST DELINQUENT ACCOUNT NUMBER ITEM?
YES
TO 350
NO
TAKE THE NEXT DELINQUENT ACCOUNT NUMBER ITEM
1

Figure 4-1. English Language Flowchart of Iteration

Symbols commonly used to show an iterative process in a flowchart are as follows:

1. A capital letter is used to symbolize a set of data. For example, "D" might be used to symbolize the set of 100 delinquent account numbers.

1. Numeric subscripts to the set symbol are used to distinguish between units in the set. For example, D, would stand for the first delinquent account number item in the set of delinquent account number items D, D_2 stands for the second delinquent account number item in the set D, D_3 stands for the third delinquent account number item, and so on, until D_{100} stands for the 100th delinquent account number.

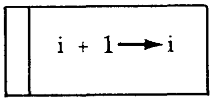
3. The general unit of the set is shown by means of an alphabetic subscript to the set symbol. For example, D_i would symbolize the ith delinquent account number item in the set D. The ith item is only one item, but it is not any particular one. The ith item is the general item. For example, the previous coding is designed to process only one delinquent account number item. Which one it happens to process depends on the contents of X15. The coding is designed to process the general delinquent account number item, D_i. The coding is particularized to process a certain delinquent account number item by establishing the contents of X15.

4. Initially, in the previous program, the contents of X15 are set so the general processing processes the first delinquent account number. Symbolically, this condition is shown as:

$$i = 1$$

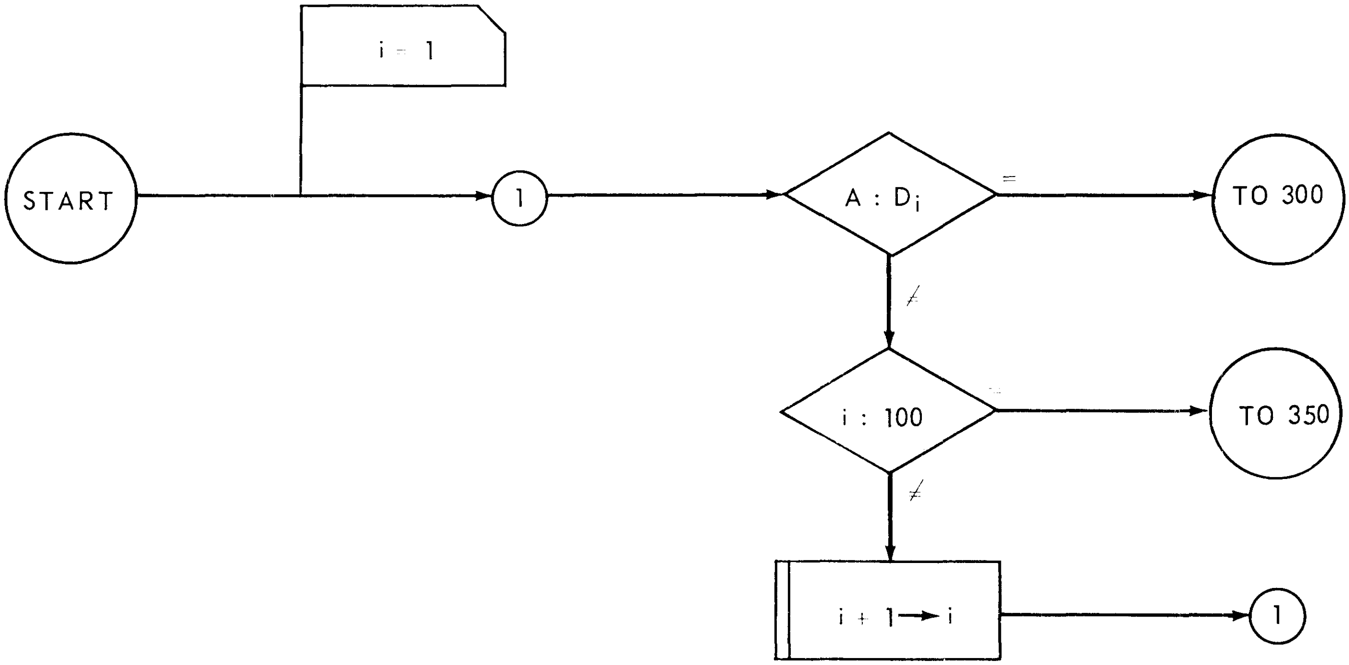
Thus, D_i becomes D_1.

5. After one delinquent account number has been processed, the contents of X15 are increased by one so the general coding which is looped through will process the next delinquent account number. Symbolically, this operation is shown as:



Thus, if D_6 (D_i with i equal to 6) has just been processed, then D_7 (D_i with i equal to 7) is the next item to be processed. As shown above, such operations are customarily shown in an "operation box" with a double line on the left.

Using the above symbols, the flowchart in Figure 4-1 might appear as shown in Figure 4-2. Notice in this flowchart that initial conditions (in this case, the fact that i is initially equal to one) are shown in an *assertion flag* set on the line of flow at the point at which the assertion shown in the flag holds true.



LEGEND

A – the new account number

D – a set of delinquent account number items

D_i – the ith item in D, i = 1, 2, 3, . . . , 100

Figure 4-2. Symbolic Flowchart of Iteration

D. STUDENT EXERCISE

There are 100 delinquent account numbers in ascending order in storage locations 200 - 299. There are 10 account numbers in random order in locations 300 - 309. Store the account numbers that are delinquent in sequential locations beginning at 310. When finished, jump to 100.

E. ITERATIVE versus STRAIGHTLINE CODING

Section B of this chapter shows both a straightline and an iterative solution to the same coding problem. These solutions exemplify the principle characteristics of these two approaches.

1. Straightline coding requires many storage locations but involves few instructions to be executed per item processed.

2. Iterative coding requires fewer storage locations but involves more instructions to be executed per item.

F. MODULAR ADDRESSING

The operation of the index register specified during instruction execution is to have its contents added to the m portion of the instruction to yield m′, the operand address. The result of this addition is a 15 bit operand address. Thus, by means of index registers, any storage location in the Processor may be addressed. The following example shows how the index registers are used to achieve this addressing.

1. Example

There are 100 delinquent account numbers stored in storage locations 5120 - 5219. If the new account number stored in location 6144 is delinquent, jump to 2000. If not, jump to 1500. Start your coding in 1026. Presume that index register 3 contains 1024. (Index register 3 is known as a *cover* index register, since it "covers" the coding. That is, it is the index register whose contents allow the instructions to address each other.)

2. Coding

LABEL Δ	OPERATION Δ	OPERAND Δ	COMMENTS
0			THIS IS LOCATION 1024
1			THIS IS LOCATION 1025
2	LX	15, 1023,3	THIS IS LOCATION 1026
3	LX	14, 1022,3	THIS IS LOCATION 1027
4	LA	8, 0,14	THIS IS LOCATION 1028
5	C	8, 0,15	THIS IS LOCATION 1029
6	JE	976,3	THIS IS LOCATION 1030
7	IXC	15, 1021,3	THIS IS LOCATION 1031
8	JL	5,3	THIS IS LOCATION 1032
9	J	476,3	THIS IS LOCATION 1033
.			.
.			.
.			.
476			THIS IS LOCATION 1500
.			.
.			.
.			.
976			THIS IS LOCATION 2000
.			.
.			.
.			.
1021	ICW	5220,1	THIS IS LOCATION 2045
1022	+	6144	THIS IS LOCATION 2046
1023	+	5120	THIS IS LOCATION 2047

Notice in this coding that the first entry in a line is no longer the address of the storage location in which the instruction or constant is to be stored, but is now the *address relative* to the contents of the cover index register (index register 3, whose contents is 1024). Thus, storage location 1024 is assigned the relative address 0 (1024 + 0 = 1024). location 1030 is assigned the relative address 6 (1024 + 6 = 1030), 2045 is assigned the relative address 1021 (1024 + 1021 = 2045), and so on. These relative addresses are the ones used in the m portion of the instructions specifying that this m portion should be modified by index register 3 to develop m′. From now on this convention will be followed in this manual until the concept of labels is introduced.

In the above exercise the contents of the cover register was given. Suppose that such is not the case. It is then the programmer's responsibility to load his cover index registers. To demonstrate how this is done, it is necessary to have a finer understanding of how the control unit of the Processor works and to have the definition of one more instruction. These are given below.

3. Control Unit Operation

The control unit contains a fifteen bit register called the *control counter*, customarily abbreviated as cc. In general, the control counter contains the address of the instruction currently being executed.

The control unithas an operating cycle depicted in flowchart form in Figure 4-3. The operating cycle begins at connector one.

The flowchart in Figure 4-3 demonstrates how the control unit effects the sequential instruction execution characteristic of the Processor, and also how the jump instructions interrupt this sequence. It also indicates that, during the execution of any instruction other than jump instructions, the control counter contains the address of the storage location from which the instruction was selected for execution.

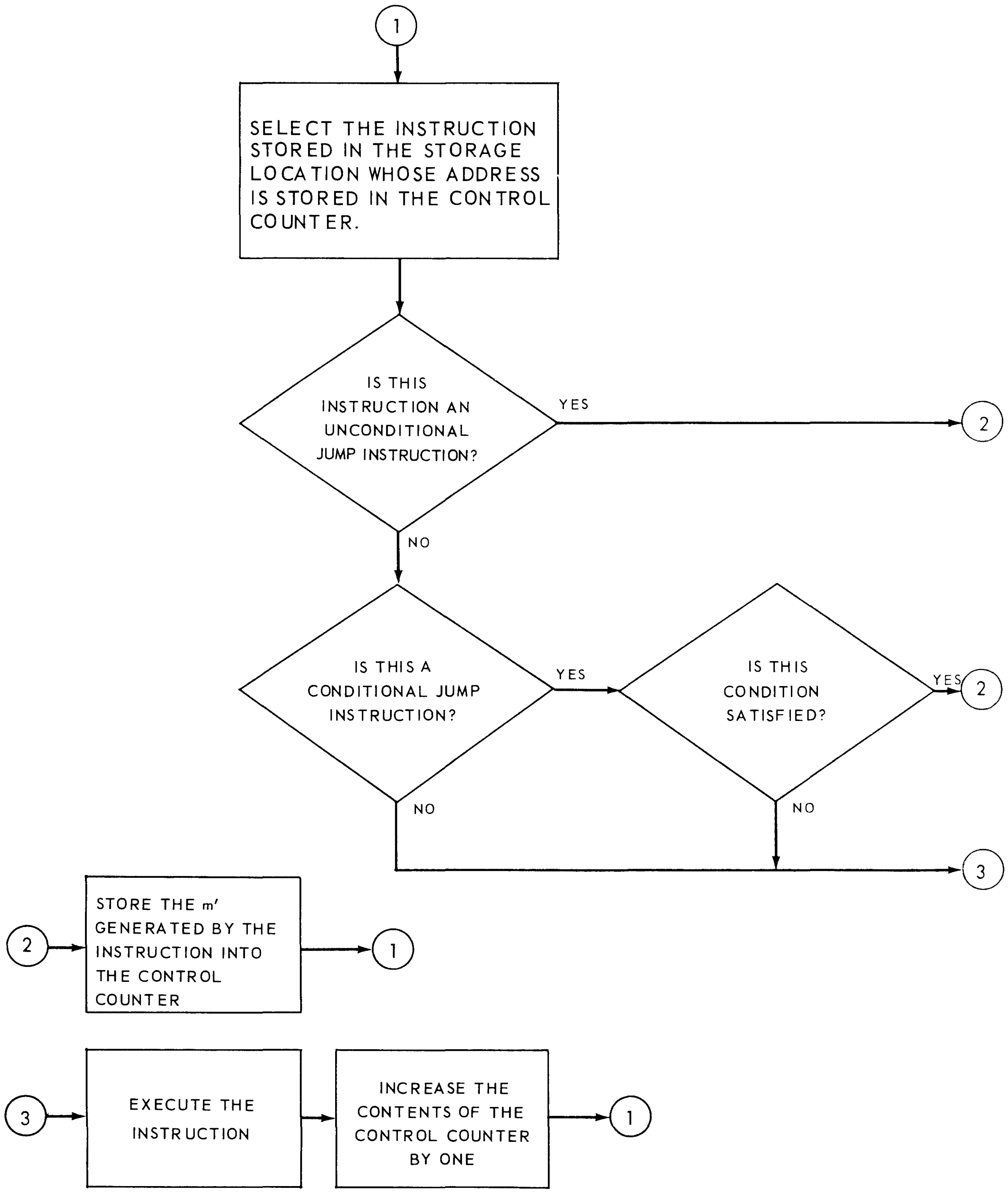

1
SELECT THE INSTRUCTION STORED IN THE STORAGE LOCATION WHOSE ADDRESS IS STORED IN THE CONTROL COUNTER.
IS THIS INSTRUCTION AN UNCONDITIONAL JUMP INSTRUCTION?
YES
2
NO
IS THIS A CONDITIONAL JUMP INSTRUCTION?
YES
IS THIS CONDITION SATISFIED?
YES
2
NO
NO
3
2
STORE THE m′ GENERATED BY THE INSTRUCTION INTO THE CONTROL COUNTER
1
3
EXECUTE THE INSTRUCTION
INCREASE THE CONTENTS OF THE CONTROL COUNTER BY ONE
1

Figure 4-3. Control Unit Operating Cycle

4. The Store Location Instruction

STORE LOCATION – SL

Store the contents of the control counter in bits one through 15 of the storage location specified by m′. Store binary zeros in the other bit positions of m′. Multiword operands and field selection are meaningless with this instruction. Indirect addressing may be used.

Example:

```
1  LABEL      Δ      OPERATION          Δ          OPERAND          Δ
           S L  1
```

(cc) = 1024*

(1) i = + 012345   (1) f = 1024*

5. Loading Cover Index Registers

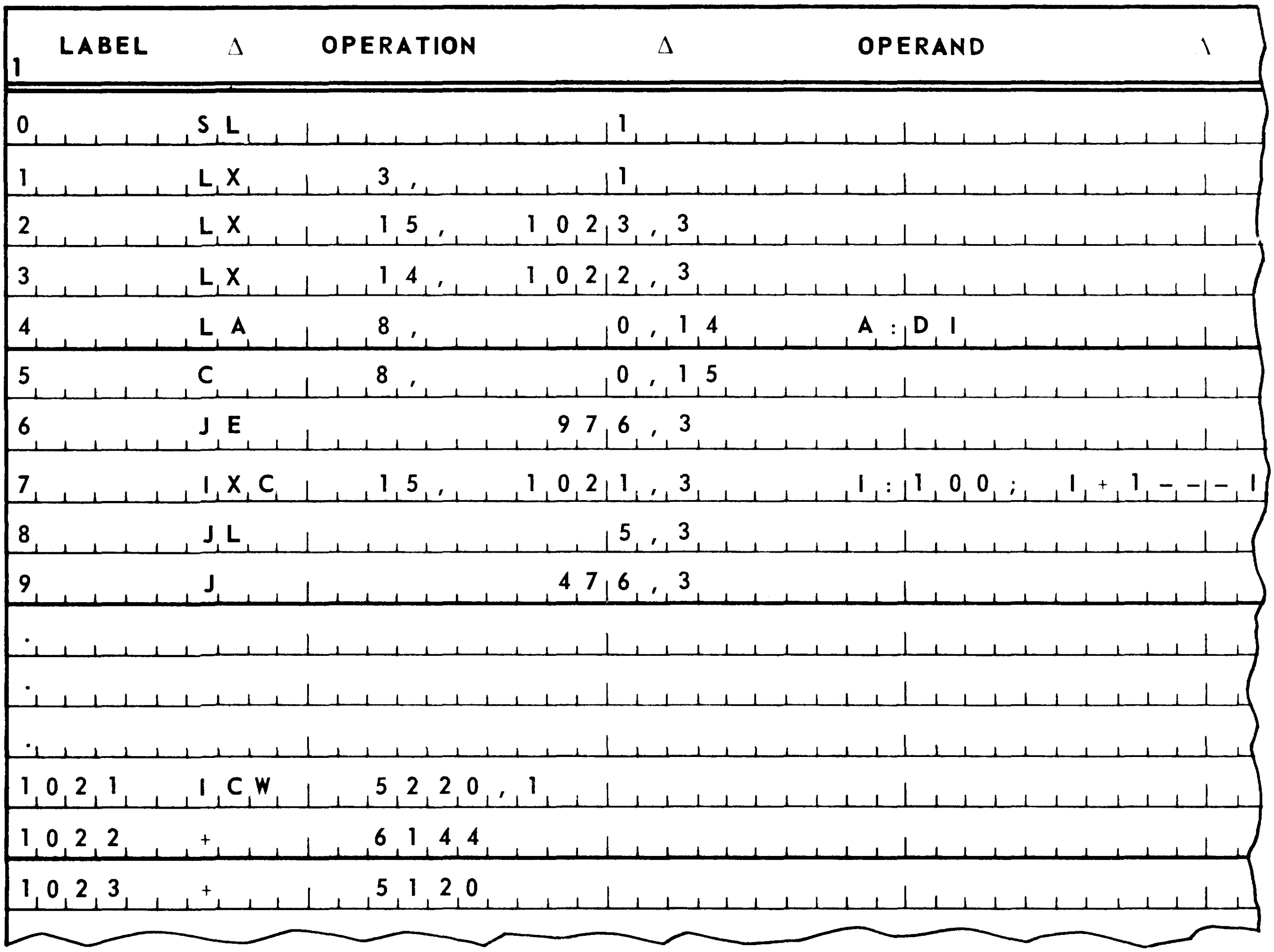
Before any cover index registers are loaded, which is the case at the beginning of a program, the only storage locations that an instruction may address are locations 0 – 1023. Since the executive routine preempts these locations, they are not available to the programmer. However, because of the need to use at least one of these locations until a cover index register is loaded, the executive routine makes available for programmer use storage locations 0 and 1. The following example shows how one of these locations might be used to load a cover index register

a. Example

There are 100 delinquent account numbers stored in storage locations 5120 - 5219. If the new account number stored in location 6144 is delinquent, jump to 2000. If not, jump to 1500. Start your coding in 1024.

* *Actually binary numbers. Moreover, only the contents of the 15 least significant bits of the final contents of storage location 1 are shown.*

b. Coding



```
1 LABEL      Δ      OPERATION          Δ          OPERAND          \
0           SL                       1
1           LX       3,              1
2           LX       15,       1023, 3
3           LX       14,       1022, 3
4           LA       8,              0, 14         A:DI
5           C        8,              0, 15
6           JE                  976, 3
7           IXC      15,       1021, 3             I:100;  I+1---I
8           JL                       5, 3
9           J                   476, 3
.
.
.
1021        ICW      5220, 1
1022        +        6144
1023        +        5120
```

c. Student Exercise

There are 100 delinquent account numbers stored in ascending order in storage locations 5120 - 5219. There are 10 account numbers stored in random order in locations 6144 - 6153. Store the account numbers that are delinquent in sequential locations beginning at 7168. When finished, jump to location 1500. Start your coding at 1024.

d. Test

There are 100 consumption items stored in storage locations 5120 - 5319 in the form:

WORD	DATA
0	NNNNNN
1	00AAAA▲

where:

N is a meter number

A is a consumption amount

Compute the body of the following table.

CONSUMPTION RANGE	TOTAL CONSUMPTION AMOUNT	TOTAL NUMBER OF METERS
0 – 100		
101 – 500		
501 – 1000		
1000 and over		

Store your results in decimal form in eight storage locations. When finished, jump to location 1500. Start your coding in 1024.

5. SUBROUTINES

The general format of a program is shown in Figure 5-1. The processing that is done on input items to produce output items is unique to the program. However, the coding required to advance input and output items is usually quite standard from one program to the next. This input and output item advance coding is, consequently, generally written as units and used by whatever program needs them. These units are called *subroutines*. Thus, there might be an input item advance subroutine and an output item advance subroutine.

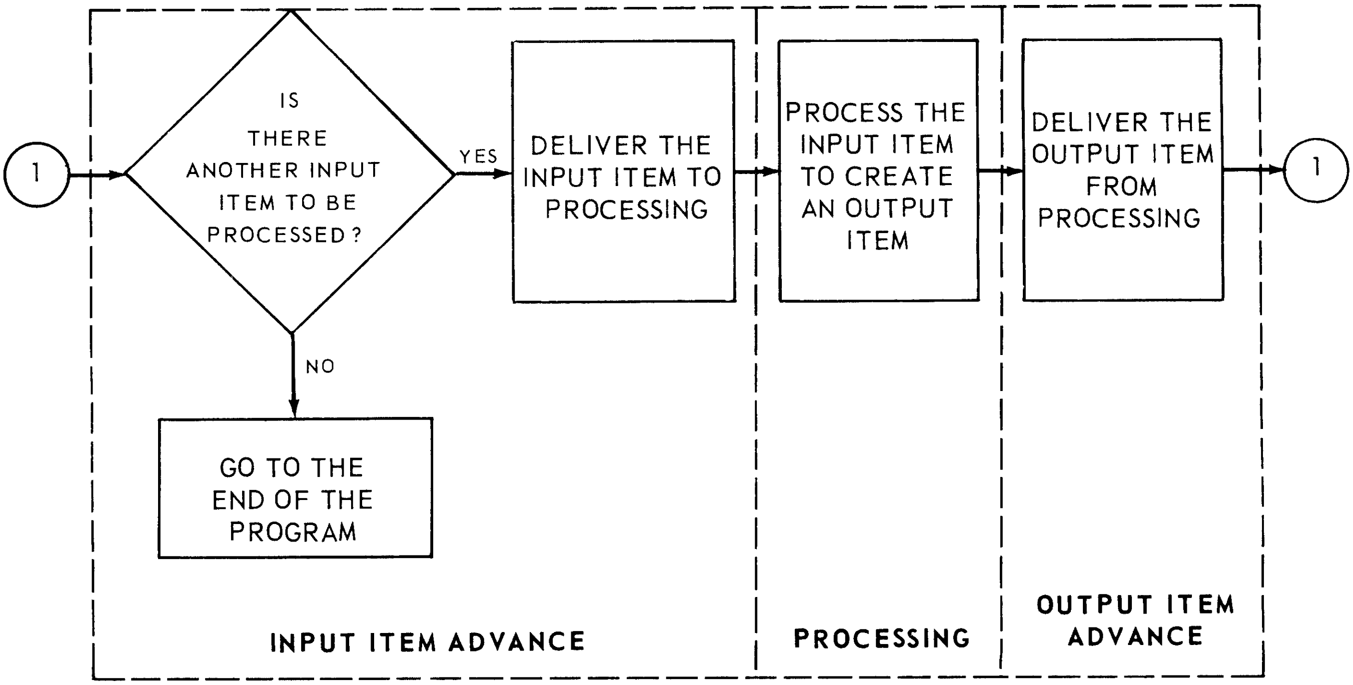

1
IS THERE ANOTHER INPUT ITEM TO BE PROCESSED ?
YES
NO
DELIVER THE INPUT ITEM TO PROCESSING
GO TO THE END OF THE PROGRAM
PROCESS THE INPUT ITEM TO CREATE AN OUTPUT ITEM
DELIVER THE OUTPUT ITEM FROM PROCESSING
1
INPUT ITEM ADVANCE
PROCESSING
OUTPUT ITEM ADVANCE

Figure 5-1. General Program Format

Subroutines are not restricted to input and output item advance handling. Other examples of subroutines that are generally useful are rounding subroutines, double precision arithmetic subroutines, floating point subroutines, and so on.

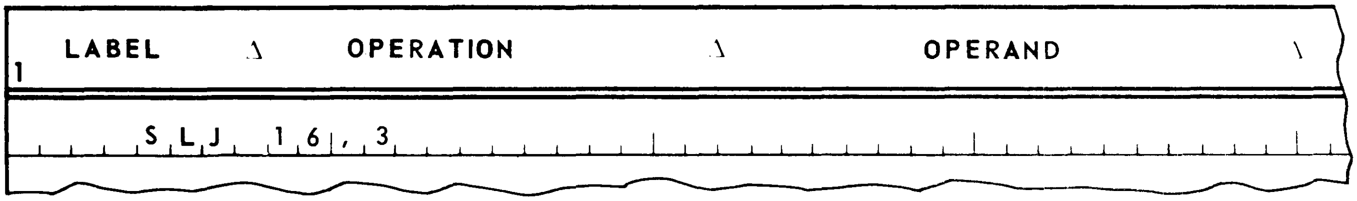
Nor is the use of the subroutine concept restricted to functions that are useful across programs. For example, a program may need to do a standard type of editing at several points along the chain of processing. Rather than code the instructions needed to perform this editing at each point at which it is required in the processing chain, the editing can be coded once as a subroutine. Then, whenever in the chain of processing it is required to do the editing, the editing subroutine can be "executed". Such a subroutine is called a *common subroutine*, because it is "common" to more than one point in the program.

Subroutines are characteristically executed by means of the store location and jump instruction.

STORE LOCATION AND JUMP – SLJ

Add one to the contents of the control counter and store the sum in bit positions one through 15 of the storage location specified by m'. Set the other bit positions of the location to zero. Jump to m' + 1. Multiword operands and field selection are meaningless with this instruction. Indirect addressing may be used.

Example:



LABEL Δ	OPERATION Δ	OPERAND Δ
	SLJ	16,3

```
                    (X3)     = 1024*
                    (cc)     = 1027*
(1040) i = + ΔΔΔΔ   (1040)f  = 1028*
```

The next instruction to be executed is found in storage location 1041.

The general form of a subroutine which is n + 1 words in length and which begins in storage location *m* is as follows:

```
m        NOP      0
m + 1                  )
.                      )  coding to perform
.                      )  the function of
.                      )  the subroutine.
m + n    J        *m
```

Line m of the subroutine is called the *return line*, line m + 1 the *entrance line*, and line m + n the *exit line*.

* *Actually in binary. Moreover, only the final contents of the 15 least significant bits of storage location 1040 are shown.*

To exemplify the way in which a subroutine is executed, suppose an input item advance subroutine has the following characteristics.

1. *The return line is found in location 1040.*
2. *The entrance line is found in 1041.*
3. *Lines 1041 - 1050 contain the coding required to perform the input item advance.*
4. *The exit line is found in 1051 and has the form J * 16,3.*

Suppose further that the sequence of coding of the main chain of the program is such that the programmer is now ready to write an instruction that will ultimately be stored in storage location 1027. The programmer is using index register 3 as a cover index register, and index register 3 contains 1024. He now wants to execute an input item advance. Therefore, he should write the instruction SLJ 16,3.

As a result of the execution of this instruction, the contents of the control counter (1027) plus one will be stored in location 1040 (1024 + 16). Thus, 1040 will now contain 1028. Control is transferred to location 1041 (m′ + 1), where the instructions stored in locations 1041 through 1050 will be executed to perform the input item advance. Control then goes to the instruction stored in location 1051. This instruction (J * 16,3) will transfer control to the address stored in location 1040 (1024 + 16) by means of indirect addressing. Thus, control returns to the instruction stored in location 1028, the next instruction in sequence in the programmer's main chain. A schematic of this sequence of control is shown in Figure 5-2.

A. SUBROUTINE FLOWCHART SYMBOLS

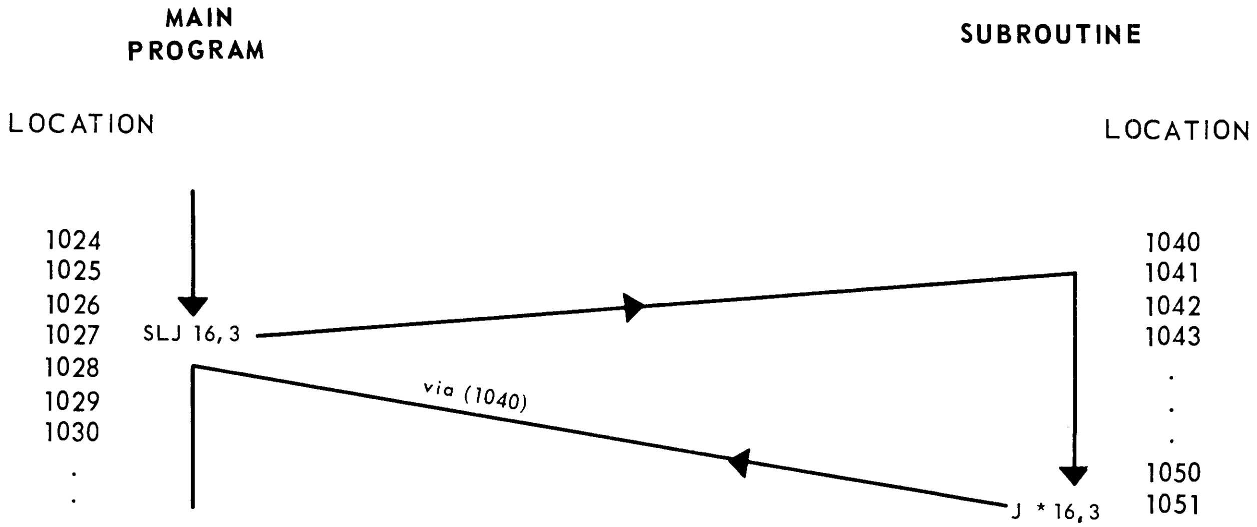
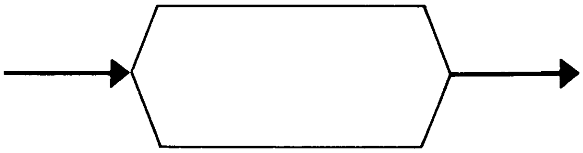


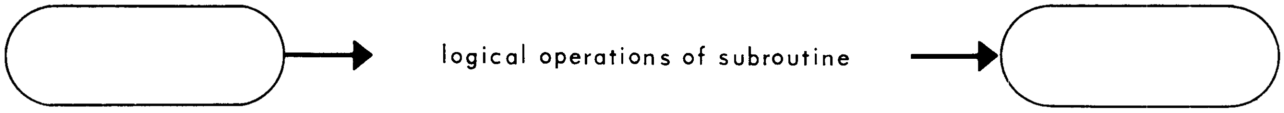
Figure 5-2. Schematic of Control Sequence in Subroutine Execution

When on the logical line of flow, the programmer wants to execute a subroutine, he shows the following symbol.



The name of the subroutine to be executed is written inside the subroutine symbol.

If the programmer wants to flowchart a subroutine itself, the logical operations constituting the subroutine are enclosed within the following symbols.

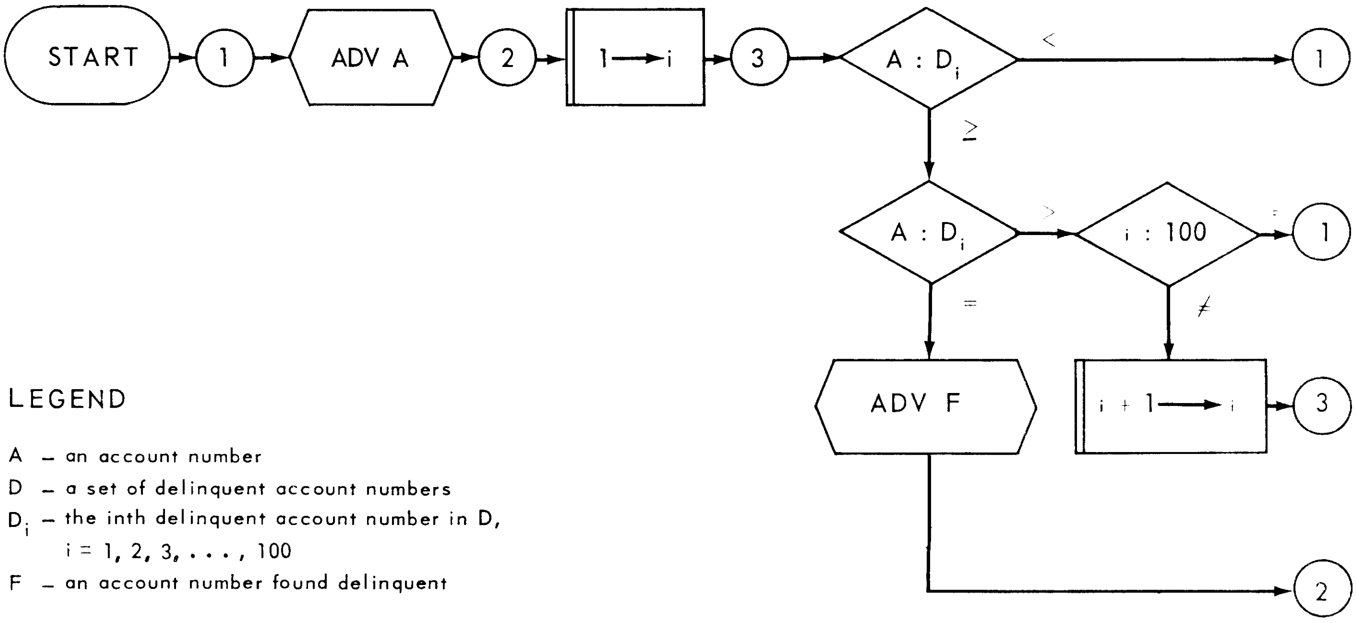


The symbol on the left indicates the beginning of the subroutine, the symbol on the right the end of the subroutine. The name of the subroutine should appear in both the beginning and ending symbol.

B. EXAMPLE

There are 100 delinquent account numbers stored in ascending sequence in storage locations 5120 - 5219. Executing a input item advance subroutine with an SLJ to location 2048 will deliver the address of an account number in 2100. If the account number is delinquent, store it in an output area. Executing an input/output item advance subroutine with an SLJ to location 2548 will store in an output area the account number whose address is in 2100 and will store the address of the next account number in 2100. Start your coding in 1024.

C. FLOWCHART



LEGEND

A – an account number

D – a set of delinquent account numbers

D_i – the inth delinquent account number in D,
i = 1, 2, 3, . . . , 100

F – an account number found delinquent

D. CODING

```
LABEL  Δ  OPERATION         Δ      OPERAND        Δ      COMMENTS
0         SL                 1
1         LX      3,         1
2         SLJ           *1023,3   ADV  A
3         LX      15,   *1022,3
4         LX      14,    1021,3   1 - - - I
5         LA      8          0,15 A  :  DI
6         C       8          0,14
7         JL                 2,3
8         JG                11,3  A  :  DI
9         SLJ           *1020,3   ADV  F
10        J                  3,3
11        IXC     14,    1019,3   I  :  100  ;  I  +  1  - - -  I
12        JL                 5,3
13        J                  2,3
.
.
1019      ICW     5220,1
1020      +       2548
1021      +       5120
1022      +       2100
1023      +       2048
```

F. FLOWCHART FIELD NOTATION

An item is shown on a flowchart by means of a capital letter, either subscripted or not, depending on whether the item is an element of a set of items. For example, the letter J might be used to represent a job item.

An item may consist of several fields. Fields are also symbolized by capital letters, but are written as superscripts to their item symbol. Thus, J^N might be used to represent the job number field of a job item, J^C to represent the contract price field of the job item, J^L the labor cost field of the job item, J^M the material cost of the job item, and J^V the overhead cost of the job.

1. Example

Given a job item of form:

WORD	DATA
0	NNNNNN
1	OCCC▲CC
2	OLLL▲LL
3	OMMM▲MM
4	OVVV▲VV

where:

N	is the job number
C	is the contract price
L	is the labor cost
M	is the material cost
V	is the overhead cost

Produce a profit item of form:

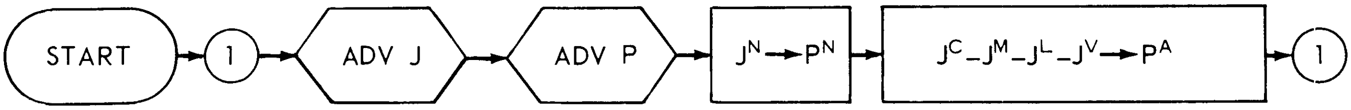
WORD	DATA
0	NNNNNN
1	OAAA▲AA

where:

N	is the job number
A	is the profit

Executing an input item advance subroutine with an SLJ to storage location 2048 will deliver the address of the zero word of a job item in location 2100. Executing an output item advance subroutine with an SLJ to 2548 will deliver in 2600 the address of the zero word of an output area for a profit item. Start your coding in 1024.

2. Flowchart



LEGEND

- J – a job item
- J^N – the job number of J
- J^C – the contract price of J
- J^M – the material cost of J
- J^L – the labor of J
- J^V – the overhead cost of J
- P – a profit item
- P^N – the job number of P
- P^A – the profit of P

3. Coding

```
1 LABEL   Δ   OPERATION                 Δ        OPERAND               Δ           COMMENTS
0         SL                   1
1         LX          3,       1
2         SLJ             *1023,3   ADV J
3         LX          15,*1022,3
4         SLJ             *1021,3   ADV P
5         LX          14,*1020,3
6         LA          12,      1,15 JN ---  PN
7         DS          4,       2,15 JC - JM - JL - JN --- PA
8         DS          4,       3,15
9         DS          4,       4,15
10        SA          12,      1,14
11        J                    2,3
.
.
.
1020      +           2600
1021      +           2548
1022      +           2100
1023      +           2048
```

4. Student Exercise

Given an inventory item of form:

WORD	DATA
0	NNNNNN
1	OHHHHH▲
2	XXXXXX
3	XXXXXX

where:

N is a stock number

H is the onhand quantity

X is other data

Also given a sales item of form:

WORD	DATA
0	NNNNNN
1	OQQQQQ

where:

N is a stock number

Q is the sales quantity

Executing an input item advance subroutine with an SLJ to storage location 2048 will deliver the address of the zero word of an inventory item in location 2100. Executing an input item advance subroutine with an SLJ to 2548 will deliver the address of the zero word of a sales item in location 2600. Executing an input/output item advance subroutine with an SLJ to 2848 will store in an output area the inventory item whose address is in 2100 and will store the address of the zero word of the next inventory item in 2100. The first inventory item and the first sales item have the same stock number, the second inventory item and second sales item have the same stock number, the third inventory item and sales item have the same stock number, and so on. Update the inventory. Start your coding in 1024.

6. VARIABLE CONNECTORS

A programming technique based on instruction modification is the use of a *variable connector*. This operation is a variation of branching, in which a decision is made to branch between two or more alternative lines of processing. In a branch the decision and the branch are made at the same point in the program. When a variable connector is used, the decision is made at one point and the branch is made at a later point. The result of the decision is stored in the form of the setting of a switch, or variable connector, which indicates the branch to be taken at a later point in the program.

A set of instructions that is sometimes used to implement variable connectors is the set of instructions which manipulate the *sense indicators*. A sense indicator is similar to the high, low and equal indicators. It is a two state device that can be turned on or off and whose setting can be tested by means of a jump sense instruction. The Processor provides eight sense indicators, numbered one through eight The instructions that operate on the sense indicators are as follows.

SET SENSE – SS

Turn the sense indicator specified in the AR portion of the instruction on. The number of the sense indicator to be turned on is specified in excess seven binary code in the AR portion. In UTMOST language this number is written as a decimal number (8-15). Multiword operands, indirect addressing, and field selection are not applicable to this instruction.

Example:

LABEL Δ	OPERATION Δ	OPERAND Δ
SS 8		

At the end of instruction execution, sense indicator one is turned on.

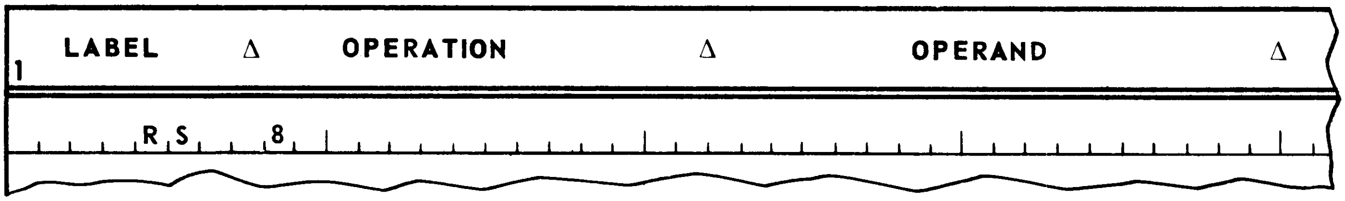
UTMOST generates this line of coding into

I/A/FS	x	OP	AR	m
0	0	62	1 0 0 0	0——————0

RESET SENSE – RS

Turn the sense indicator specified in the AR portion off. The number of the sense indicator is specified in excess seven binary code in the AR portion. In UTMOST language this number is written as a decimal number (8-15). Multiword operands, indirect addressing, and field selection are not applicable.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

RS 8

At the end of instruction execution, sense indicator one is turned off.

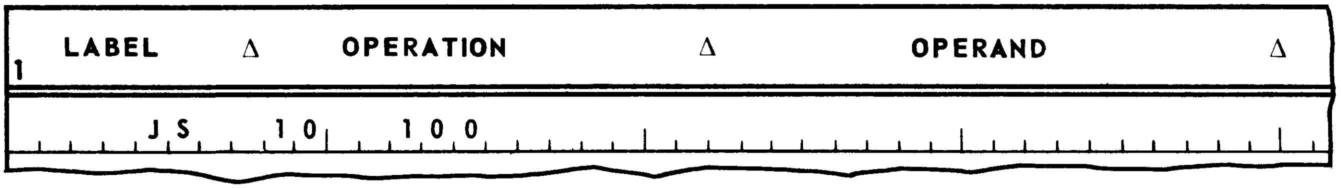
UTMOST generates this line of coding into

IA/FS	x	OP	AR	m
0	0	61	1 0 0 1	0——————————0

JUMP SENSE – JS

If the sense indicator specified in the AR portion is on, jump to m′. Otherwise, go to the next instruction. The number of the sense indicator is specified in excess seven binary code. In UTMOST language the number is written as a decimal number (8-15). Multiword operands and field selection are not applicable. Indirect addressing may be used.

Example:



1 LABEL Δ OPERATION Δ OPERAND Δ

JS 10 100

UTMOST generates this line of coding into

IA/FS	x	OP	AR	m
0	0	60	1 0 1 0	0——————0 1 0 0

Control will be transferred to location 100 if Sense Indicator 3 is set.

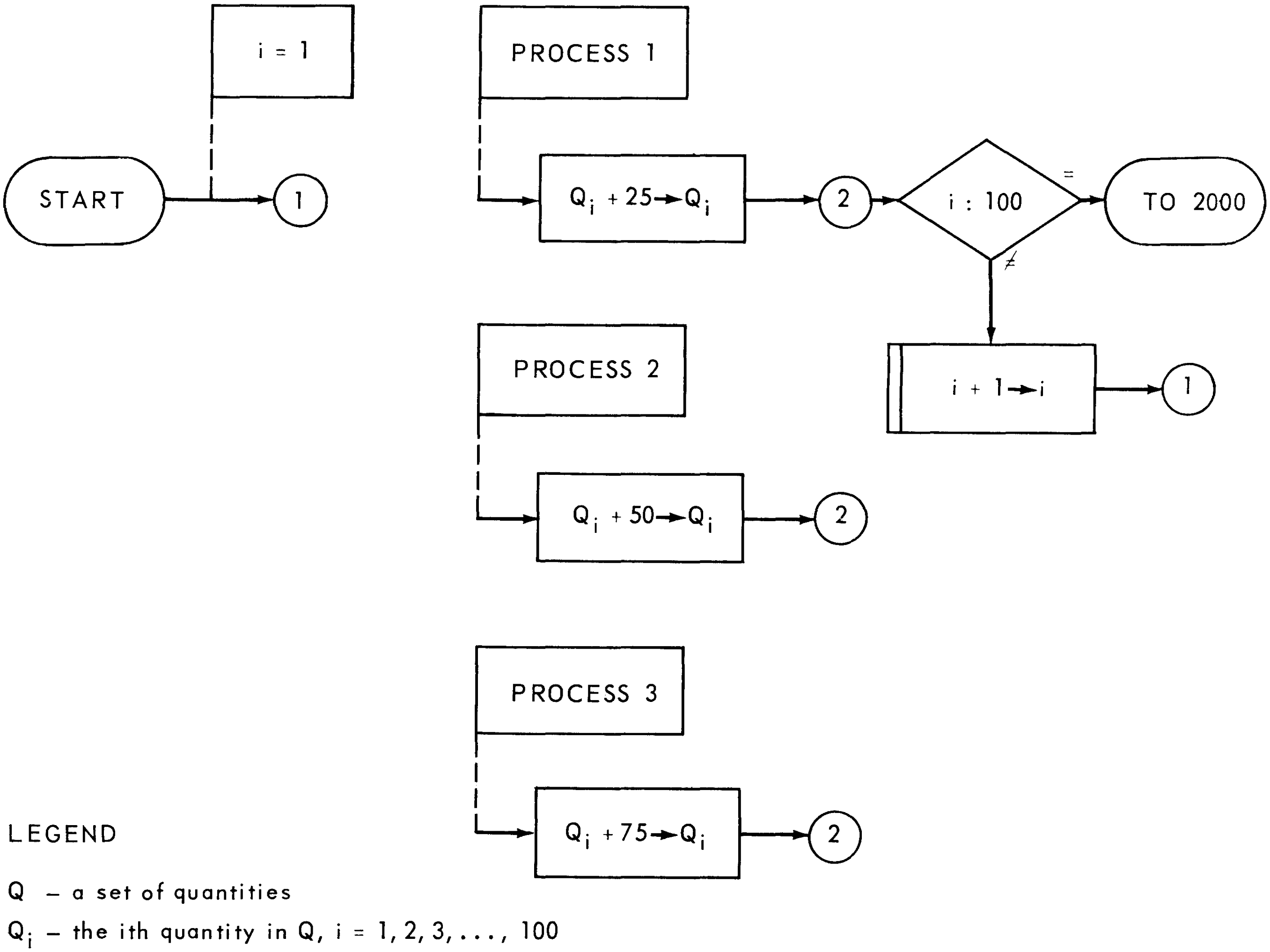
A. EXAMPLE

There are 100 quantities stored in storage locations 5120 - 5219 in the form OQQQQQ▲ Add 25 to the first, fourth, seventh, etc. quantities. Add 50 to the second, fifth, eighth, etc. quantities. Add 75 to the third, sixth, ninth, etc. quantities. Process the quantity stored in location 5120 first, the quantity in location 5121 second, the quantity in 5122 third, and so on. When finished jump to 2000. Start your coding in 1024.

B. FLOWCHART

A partial flowchart for this example is shown in Figure 6-1. This flowchart is indeterminate at connector one. The first time control reaches this connector, process one should be executed. The second time, process two should be executed. The third time process three should be executed. The fourth time, process one. And so on. Connector one must be variable. This is indicated by subscripting the one with a ''v''. The connector is then the terminal of a switch. The poles of the switch are also indicated by connectors, the connectors being identified by the same number but being subscripted with successive letters of the alphabet. Figure 6-2 shows the flowchart in Figure 6-1 with the variable connector included.

The flowchart in Figure 6-2 is still indeterminate in that it does not show which pole connector 1v is set to. Setting a variable connector is shown in an operation box. The pole to which the connector is to be set is written in the operation box and is preceded by a period (.), a customary abbreviation for "set" Figure 6-3 shows the flowchart in Figure 6-2 with the setting of the variable connector included.

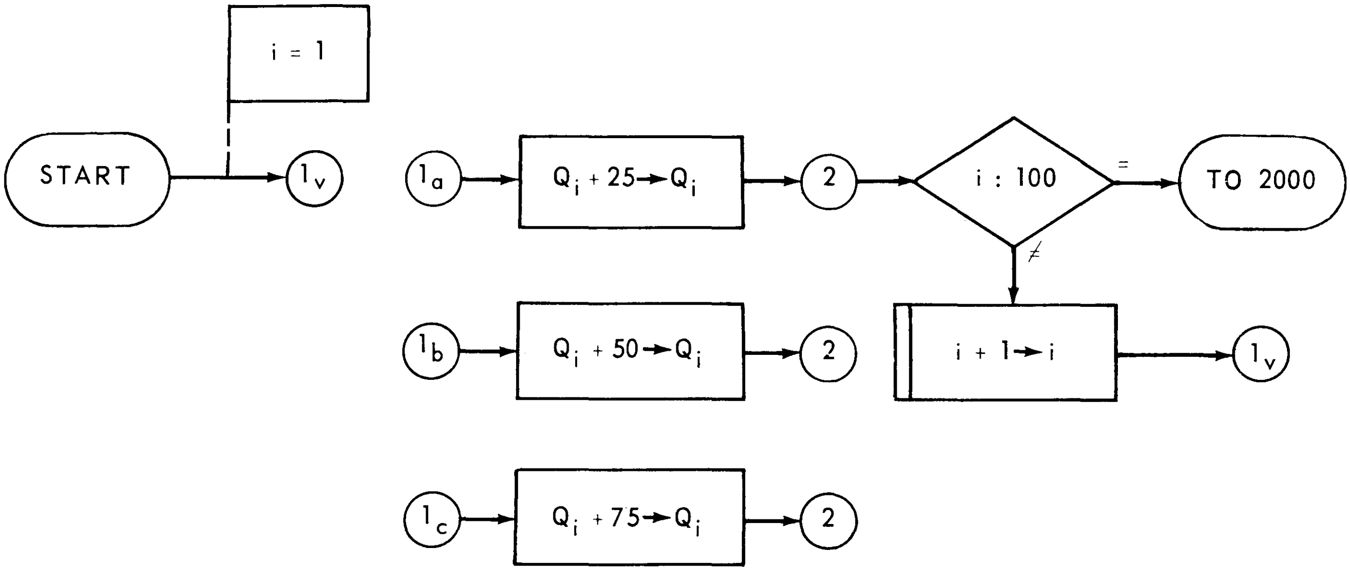


LEGEND

Q – a set of quantities

Q_i – the ith quantity in Q, i = 1, 2, 3, . . . , 100

Figure 6-1. Partial Flowchart

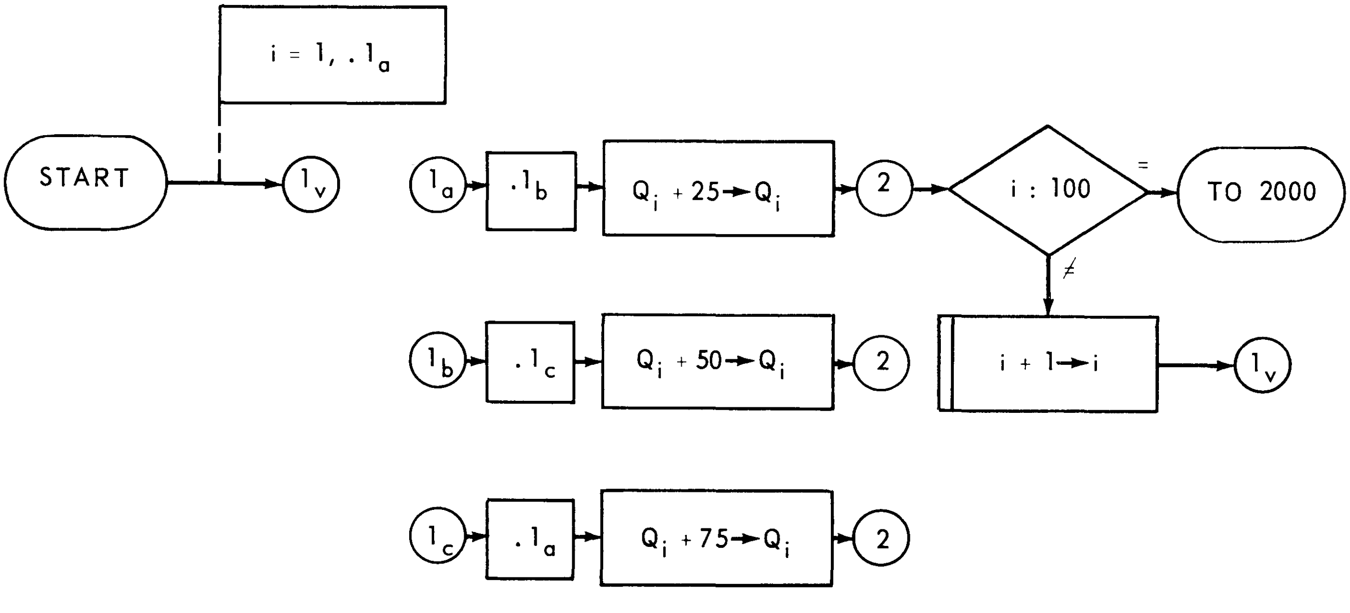

i = 1
START
1_v
1_a
$Q_i + 25 \rightarrow Q_i$
2
i : 100
=
TO 2000
≠
1_b
$Q_i + 50 \rightarrow Q_i$
2
$i + 1 \rightarrow i$
1_v
1_c
$Q_i + 75 \rightarrow Q_i$
2

LEGEND

Q – a set of quantities

Q_i – the ith quantity in Q, i = 1, 2, 3, . . . , 100

Figure 6-2. Flowchart with Variable Connector



LEGEND

Q – a set of quantities

Q_i – the ith quantity in Q, i = 1, 2, 3, . . . , 100

Figure 6-3. Flowchart with Variable Connector Settings

C. CODING

LABEL	OPERATION	OPERAND	COMME
0	SL	1	
1	LX	3,1	
2	LX	15,1023,3	I = 1
3	SS	8	.1A
4	LA	8,0,15	
5	JS	8,13,3	
6	JS	9,17,3	
7	SS	8	.1A
8	DA	8,1022,3	QI + 75 --- QI
9	SA	8,0,15	
10	IXC	15,1021,3	I : 100 ; I + 1 --- I
11	JL	4,3	
12	J	976,3	TO 2000
13	RS	8	
14	SS	9	
15	DA	8,1020,3	QI + 25 --- QI
16	J	9,3	
17	RS	9	
18	DA	8,1019,3	QI + 50 --- QI
19	J	9,3	
.			
.			
.			
1019	+	:0000050	
1020	+	:0000025	
1021	ICW	5220,1	
1022	+	:0000075	
1023	+	5120	

Another method of coding this example is as follows:

LABEL	OPERATION	OPERAND	COMMENTS
0	SL	1	
1	LX	3, 1	
2	LX	15, 1023,3	I = 1
3	LA	12, 1022,3	QI + 25 --- QI
4	DA	8, 0,15	
5	SA	8, 0,15	
6	SA	4, 3,3	.1B
7	IXC	15, 1016,3	I : 100 ; I + 1 --- I
8	JL	3,3	
9	J	976,3	
.			
.			
.			
1016	ICW	5220,1	
1017	+	:000075	
1018	LA	12, 1022,3	
1019	+	:000050	
1020	LA	12, 1018,3	
1021	+	:000025	
1022	LA	12, 1020,3	
1023	+	5120	

D. STUDENT EXERCISE

There are 166 six-word job items stored beginning at storage location 5120 in the form:

WORD	DATA
0	NNNNNN
1	SLLLLL▲
2	LLMMMM
3	M▲MMVVV
4	VV▲VVPP
5	PPPP▲PP

where:

N	is a job number
S	is a salesman's key and may be 1 for salesman one 2 for salesman two 3 for salesman three
L	is the labor cost
M	is the material cost
V	is the overhead cost
P	is the contract price

Compute for each salesman:

1. The gross sales amount.
2. The number of sales netting $1500 or more.

When finished, jump to location 2000. Start your coding in 1024.

7. TABLE LOOKUP

Many programs involve looking up information in a table stored in the memory. The technique used to do such a *table lookup* varies with the construction and sequence of the table involved. As an introduction to this subject an example table lookup illustration is given.

A. EXAMPLE

Storage locations 10,240 - 10,639 contain 400 six digit part numbers listed in ascending sequence. Listed in locations 10,640 - 11,039 are the unit costs for the parts whose part numbers are in the part number list. Each unit cost is in the form OOCC‸CC. The unit cost stored in 10,640 is the unit cost of the part whose part number is stored in 10,240. The unit cost stored in 10,641 is the unit cost of the part whose part number is in 10,241. The unit cost in 10,542 is the unit cost of the part whose part number is in 10,242. And so on. Given a sales item of the form:

WORD	DATA
0	KKKKKK
1	NNNNNN
2	QQQQ‸OO
3	OOOOOO

where:

K is a key

N is a part number

Q is the quantity sold

Compute the total cost for the sales item and store it in words 2 and 3 of the sales item in the form:

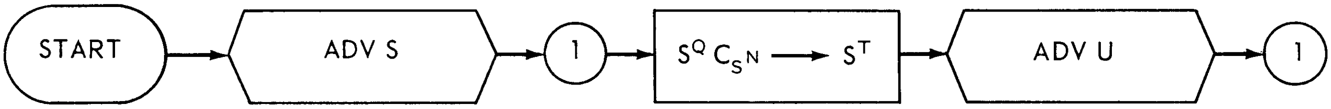
WORD	DATA
2	QQQQ‸TT
3	TTTT‸TT

where:

T is the total cost.

If the part number of a sales item cannot be found in the part number list, jump to 2000. Executing an input item advance subroutine with an SLJ to 2048 will deliver the address of the zero word of a sales item in 2100. Executing an input/output item advance subroutine with an SLJ to 2848 will store in an output area the sales item whose address is in 2100 and will store the address of the zero word of the next sales item in 2100. Start your coding in 1024.

B. FLOWCHART



LEGEND

S – a sales item

S^Q – the quantity of S

S^N – the part number of S

S^T – the total cost of S

C_{S^N}– the price of S^N

U – an updated sales item

C. CODING

```
LABEL          OPERATION              OPERAND
0              SL                        1
1              LX       3,               1
2              SLJ               *1023,3       ADV S
3              LX       15,      *1022,3
4              LA       8,               1,15
5              LA       6,        1021,3
6              SA       4,        1017,3
7              C        8,       *1017,3
8              JL                  976,3
9              SA       2,        1019,3       TABLE LOOKUP
10             BA       2,        1017,3
11             BRR      2,               1
12             OR       2,        1016,3
13             SA       2,        1018,3
14             C        8,       *1018,3
15             JL                    9,3
16             JE                   22,3
17             C        2,        1017,3
18             JE                  976,3
19             SA       2,        1017,3
20             BA       2,        1019,3
21             J                    11,3
22             BA       2,        1015,3       SQ  X  CSN  ¬--  ST
23             SA       2,        1018,3
24             LA       8,               2,15
25             DSR      8,               2
26             DM                *1018,3
27             LF       4        *1014,3
28             SA       6,               3,15
29             SLJ               *1013,3
30             J                     3,3
.
.
.
1013           +        2848
1014           +        24,9,2,15
1015           +        400
1016           +        07777777
1017           +        0                      BOTTOM
1018           +        0                      MIDDLE
1019           +        0                      TOP
1020           +        10240
1021           +        10640
1022           +        2100
1023           +        2048
```

The table lookup coding at *C.* is a specific example of a general technique known as log 2 lookup. This lookup employs the technique of comparing the middle table argument with the problem argument to determine in which half of the table the desired value lies. The selected half of the table is then divided to determine in which quarter of the table the desired value lies. This process continues until the choice is narrowed down to two values, at which point the desired value is chosen. The name of the technique derives from the fact that, if the log of the lowest power of two equal to or greater than the number of entries in the table is taken to the base 2, the result is the maximum number of comparisons required to find a specific entry.

The list of delinquent account numbers used in the delinquent account number example presented earlier in this manual can be considered a table. Lookup in this table was done by means of *sequential table lookup*. In sequential table lookup, the first entry in the table is interrogated to see if it is appropriate. If not, the second entry is interrogated. Then the third. And so on until the appropriate entry is found or the end of the table is reached.

An example of a third type of table lookup, *function table lookup,* will be given later in this manual.

8. UTMOST

Up to this point in this manual a restricted subset of the facilities available in the UTMOST language has been used in the coding of examples. The purpose of this chapter is to introduce the programmer to the full range of the UTMOST language. This presentation will be made in terms of an illustrative example, the statement of which follows.

A. EXAMPLE

Given a taxpayer item of the form:

WORD	DATA
0	NNNNNN
1	GGGGGG▲
2	GGOOOO
3	PP▲OOOO
4	OOAAAA▲
5	AAOOOO

where:

N	is a taxpayer identification
G	is the income
P	is the number of dependents
A	is the deductions other than for dependents

Produce a tax item of the form:

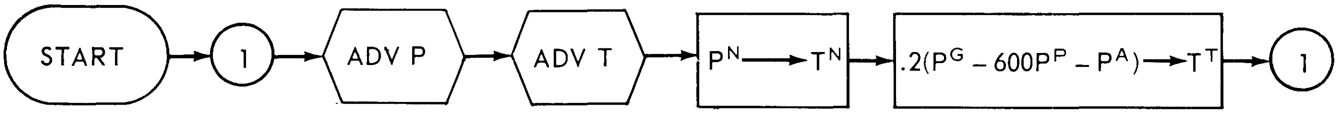
WORD	DATA
0	NNNNNN
1	OOOOTT
2	TTTT▲TT

where:

N	is the taxpayer identification
T	is the unrounded tax

A deduction of $600 is allowed for each dependent. The tax is 20% of the taxable income. Executing an input item advance subroutine with an SLJ to 2048 will deliver the address of the zero word of a taxpayer item in 2100. Executing an output item advance subroutine with an SLJ to 2548 will deliver in 2600 the address of the zero word of an output area for a tax item. Start your coding in 1024.

B. FLOWCHART



LEGEND

P – a taxpayer item

P^N – the identification of P

P^G – the income of P

P^P – the number of dependents of P

P^A – the other deductions of P

T – a tax item

T^N – the taxpayer identification of T

T^T – the amount of T

C. CODING

As has been done previously in this manual, this example might be coded as follows.

```
LABEL   Δ   OPERATION         Δ          OPERAND          Δ        COMMENTS
0       SL                 1
1       LX        3,       1
2       SLJ           *1023,3     ADV P
3       LX        15,*1022,3
4       SLJ           *1021,3     ADV T
5       LX        14,*1020,3
6       LA        8,       3,15   .2 (PG - 600PP - PA) --- TT
7       DM             1019,3
8       DS        6,       5,15
9       DA        6,       2,15
10      SA        6,   1018,3
11      LA        8,   1016,3
12      DM             1018,3
13      SA        4,   1015,3
14      DM             1017,3
15      DA        6,   1015,3
16      LA        8,       0,15   PN --- TN
17      SA        14,      2,14
18      J                  2,3
.
.
.
1014    +         :000000
1015    +         0
1016    +         :000020
1017    +         0
1018    +         0
1019    -         :060000
1020    +         2600
1021    +         2548
1022    +         2100
1023    +         2048
```

D. LABELS

UTMOST language relieves the programmer from keeping track of relative addresses. Instead a *label* is used. A label may be from one through sixteen characters long, through only the first eight(8) characters are considered by the assembler. The first character may be letters of the alphabet or decimal numbers. A label must begin in the first column of a line and must contain no spaces, but must be followed by a space. The example used in this chapter is restated below to make it appropriate for coding with labels.

1. Example

Given a taxpayer item of the form:

WORD	DATA
0	NNNNNN
1	GGGGGG▲
2	GGOOOO
3	PP▲OOOO
4	OOAAAA▲
5	AAOOOO

where:

N is a taxpayer identification

G is the income

P is the number of dependents

A is the deductions other than for dependents

Produce a tax item of the form:

WORD	DATA
0	NNNNNN
1	OOOOTT
2	TTTT▲TT

where:

N is the taxpayer identification

T is the unrounded tax

A deduction of $600 is allowed for each dependent. The tax is 20% of the taxable income. Executing an input item advance subroutine with an SLJ to the label FRD will deliver the the address of the zero word of a taxpayer item in the storage location labelled FILEP.

Executing an output item advance subroutine by means of an SLJ FWR will deliver in label FILET the address of the zero word of an output area for a tax item. Label the first instruction in your program to be executed with the label START.

2. Coding

```
LABEL   Δ  OPERATION      Δ     OPERAND      Δ     COMMENTS
START   SL          1
        LX    3,    1
C1      SLJ       *K1,3     ADV P
        LX    15,*K2,3
        SLJ       *K3,3     ADV T
        LX    14,*K4,3
        LA    8,  3,15      .2 (PG - 600PP - PA) --- TT
        DM         K5,3
        DS    6,  5,15
        DA    6,  2,15
        SA    6,  K7,3
        LA    8,  K8,3
        DM        K7,3
        SA    4,  K10,3
        DM        K6,3
        DA    6,  K10,3
        LA    8,  0,15      PN --- TN
        SA    14,2,14
        J         C1,3
K1      +     FRD
K2      +     FILEP
K3      +     FWR
K4      +     FILET
K5      -     :060000
K6      +     0
K7      +     0
K8      +     :000020
K9      +     :000000
K10     +     0
```

3. STUDENT EXERCISE

Given an input item of the form:

WORD	DATA
0	NNNNNN
1	▲0AA000
2	▲0BB000
3	▲0CC000
4	▲0DD000

where:

N is a key

A is a quantity

B is another quantity

C is a third quantity and has a minimum value of .011

D is a fourth quantity produce

produce an output item of the form

WORD	DATA
0	NNNNNN
1	▲00EEEE
2	▲FFFFFF
3	▲GGGGGG

where:

N is the key

$$E = AB$$

$$F = \frac{AB}{.9C}$$

$$G = \frac{AB}{.9C} - D$$

Executing SLJ FDR will deliver the address of the zero word of an input item to FILEI. Executing SLJ FWR will deliver in FILEO the address of the zero word of an output area for an output item. Label the first instruction in your program START.

E. DEFINITION OF TERMS

A line of UTMOST coding consists of three fields, a label, an *operation*, and an *operand*. Labels have been defined above.

The operation is the second field on a line. Examples of operations are the mnemonic op codes of instructions, the plus or minus sign of a constant, and the ICW of an increment and compare word. An operation can contain no spaces within it, must be preceded by at least one space and in general, must be followed by at least one space. The plus and minus operations are the sole exceptions to this last rule in that, if the programmer desires, the operand of a constant line can immediately follow the plus or minus operation with no intervening space.

The operand constitutes the rest of the UTMOST line. An operand is made up of one or more *expressions*, the expressions being separated by commas. An expression together with its following comma can contain no spaces within it. However, if the programmer so desires, spaces may be left between the comma of one expression and the beginning of the following expression. The last expression in an operand has no comma following it.

F. OPERATORS

All expressions written thus far in this manual have consisted of one *unit*. The following are examples of units taken from the above coding.

C1

: 060000

0

UTMOST allows an expression to consist of two or more units connected by *operators*. The operator describes to the UTMOST assembler how the units making up the expression in source code are to be combined to form the expression in object code. For example, "+" is an operator. It tells the assembler to form the object code expression out of the arithmetic sum of the source code units.

To clarify this explanation, consider the label K5 used as the address of the first DM instruction in the above coding. In transforming this source coding to object code, the UTMOST assembler is going to substitute for K5 in this DM instruction the relative address that it assigns to the constant – : 060000.

Now consider the expression K5 + 3. This tells the UTMOST assembler to arithmetically add together the relative address it assigns to the label K3 and the binary equivalent of the decimal number three. The result will be the relative address assigned to the label K8. Thus, in the above coding the expressions K5 + 3 and K8 are equivalent. Consequently, in this coding the instruction LA 8, K8, 3 could have been just as correctly written LA 8, K5 + 3, 3.

Other operators will be described later in this chapter.

G. THE USE DIRECTIVE

Instead of the programmer specifying the cover index register in instructions addressing the coding itself, the *USE directive* may be employed. A USE directive has the following form:

USE e_1 , e_2, e_3, . . .

where USE is the operation, and e_1, e_2, e_3, . . . the operand. The expressions e_1, e_2, e_3, . . . are index register numbers.

The UTMOST assembler treats the USE directive in the following way. It interprets the directive to mean that the next 1024 lines of coding immediately following the USE directive are to be covered by the first index register specified, that the next 1024 lines of coding are to be covered by the second index register specified, that the next 1024 are to be covered by the third index register specified and so on. On the basis of this assumption, the UTMOST assembler will insert the proper cover index register specification into the instructions addressing the coding itself.

In addition to causing UTMOST to insert cover index register specifications, the USE directive also causes the executive routine to properly load a program's cover index registers before turning over control to the program. Consequently, once a USE directive has been given, no further concern with cover index registers is necessary.

For efficient index register use, it is recommended that in writing a program, only one USE directive be used to specify cover index registers for coding.

The USE directive is an *assembler directive*. It is a communication between the programmer and the assembler. As a consequence, although it takes up a line of source code, it will not cause the generation of any lines of object coding. It is, instead, absorbed by the assembler.

The following is coding, incorporating the use directive, for the example being used in this chapter.

```
1 LABEL   Δ  OPERATION        Δ        OPERAND        Δ        COMMENTS

             USE              3
START        SLJ              *K1             ADV P
             LX          15,  *K2
             SLJ              *K3             ADV T
             LX          14,  *K4
             LA          8,   3,15            .2 ( PG - 600PP - PA ) --- TT
             DM               K5
             DS          6,   5,15
             DA          6,   2,15
             SA          6,   K7
             LA          8,   K8
             DM               K7
             SA          4,   K10
             DM               K6
             DA          6,   K10
             LA          8,   0,15            PN --- TN
             SA          14,  2,14
             J                START
K1           +           FRD
K2           +           FILEP
K3           +           FWR
K4           +           FILET
KT           -           :060000
K6           +           0
K7           +           0
K8           +           :000020
K9           +           :000000
K10          +           0
```

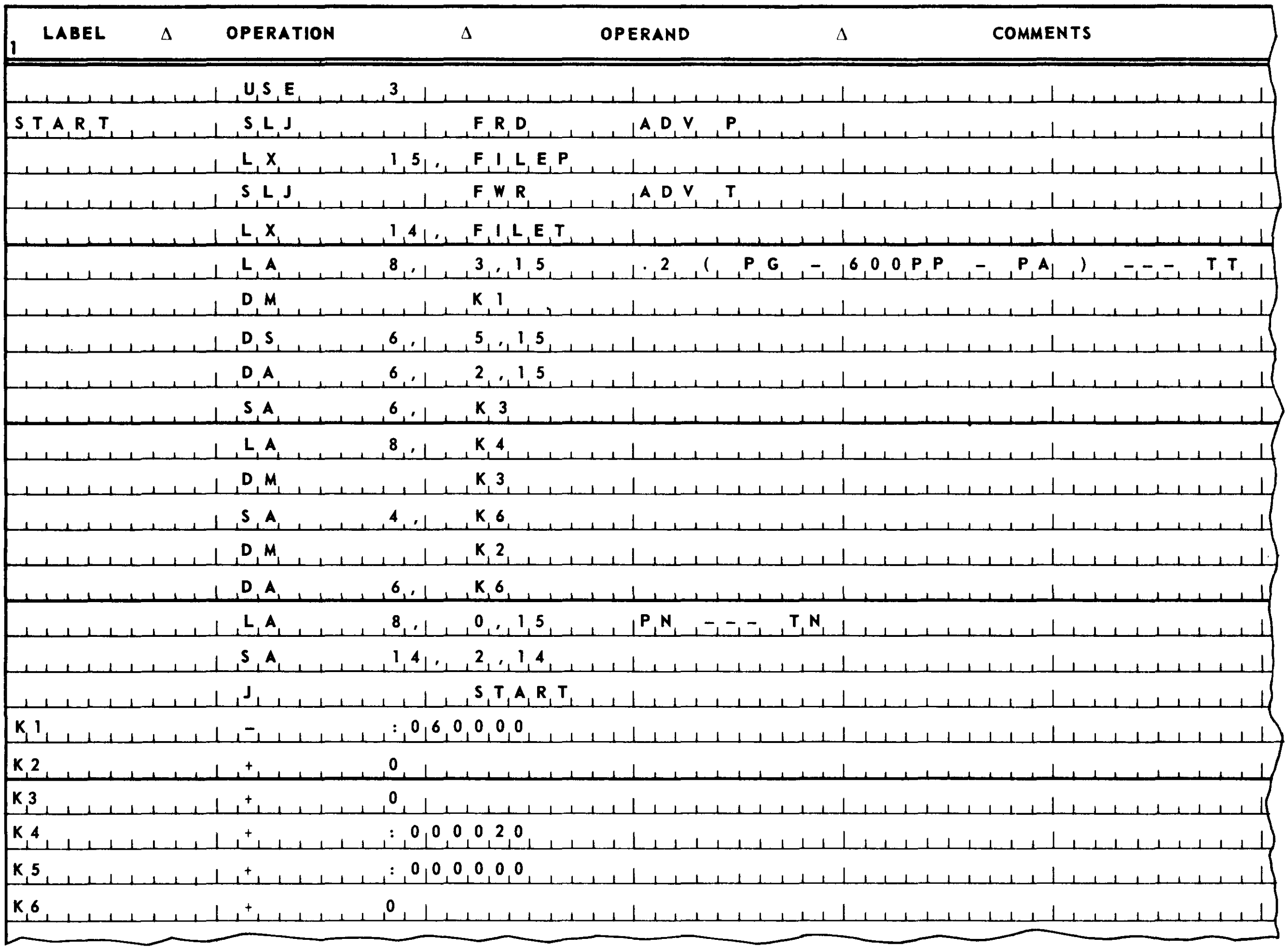
H. STUDENT EXERCISE

Code the previously stated student exercise using the USE directive.

I. INDIRECT ADDRESSING

If the UTMOST assembler encounters a source code instruction addressing a line of coding not covered by a USE directive, it will automatically modify this instruction to use indirect addressing and fabricate an indirect address control word to effect the proper addressing. Consequently, in writing a label as the operand address of an instruction addressing coding, there is never any need to worry about whether the label is covered or not.

The following is coding, taking advantage of this indirect addressing feature of the UTMOST assembler, for the example being used in this chapter.



```
1 LABEL  Δ  OPERATION         Δ           OPERAND          Δ          COMMENTS

          USE      3
START     SLJ           FRD       ADV  P
          LX       15,  FILEP
          SLJ           FWR       ADV  T
          LX       14,  FILET
          LA       8,   3,15      .2 ( PG - 600PP - PA ) --- TT
          DM            K1
          DS       6,   5,15
          DA       6,   2,15
          SA       6,   K3
          LA       8,   K4
          DM            K3
          SA       4,   K6
          DM            K2
          DA       6,   K6
          LA       8,   0,15      PN --- TN
          SA       14,  2,14
          J             START
K1        -        :060000
K2        +        0
K3        +        0
K4        +        :000020
K5        +        :000000
K6        +        0
```

J. STUDENT EXERCISE

Code the previously stated student exercise using the automatic indirect addressing feature of the UTMOST assembler.

K. LITERALS

Notice, for example, the following coding from the previous example.

```
   LABEL     Δ    OPERATION          Δ          OPERAND          Δ
1
             DM        K1

.
.
K1           -         :060000
```

Coding in this form requires the programmer to create a label for the m portion of an instruction, and then, in a separate section of coding, write the label and the desired constant. UTMOST language simplifies this process by means of *literals*. A literal allows the programmer to write the desired constant as the m address. This is done by enclosing the desired constant in parentheses. Thus the above coding would appear as follows.

```
             DM        (-:060000)
```

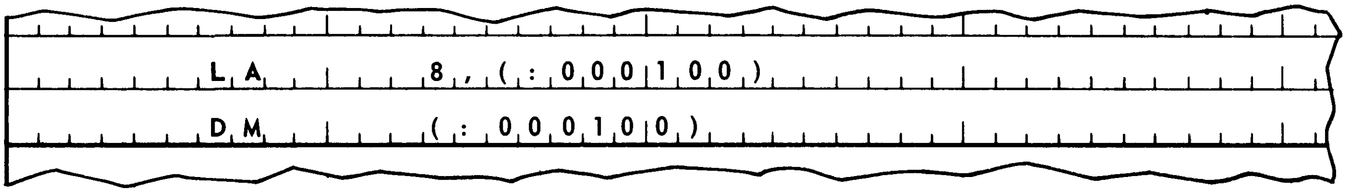
In addition, if the literal has a plus operation, it is not necessary to write the plus. Thus, in the UTMOST language the following two lines are identical.

```
             LA        8,(+:000020)
             LA        8,(:000020)
```

In a similar fashion, if a literal is used to write the increment and compare word for an IXC instruction, the operation ICW may be omitted. Thus, the following two lines are identical.

```
             IXC       8,(ICW   LIMIT,10)
             IXC       8,(LIMIT,10)
```

When it encounters a literal, the UTMOST assembler will generate the constant described by the literal, place it at the end of the programmer's program, and effect the proper addressing. In handling literals, the assembler does not create duplicate constants. For example, if the following two lines appear in a program:

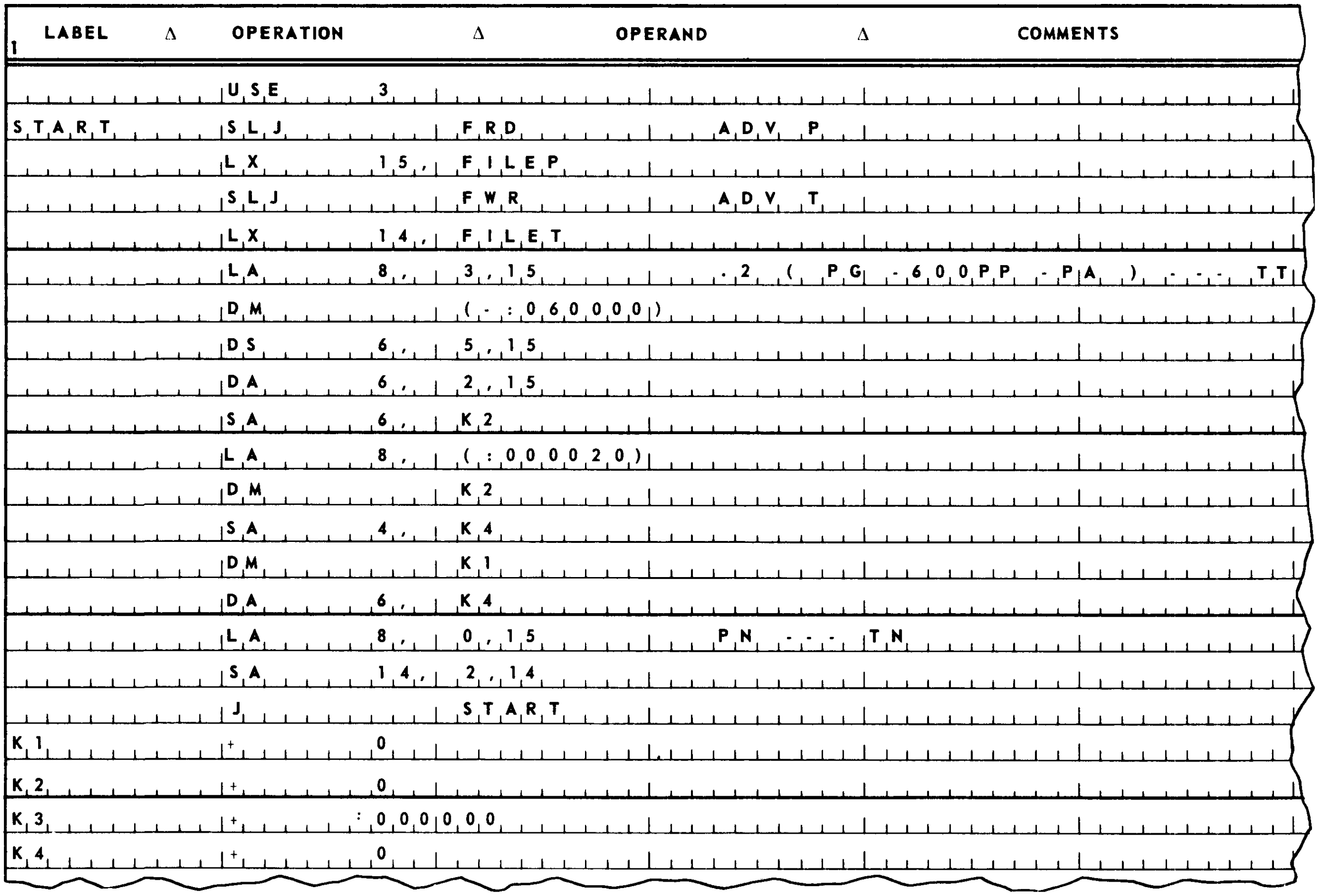


```
         LA     8, (: 000100)
         DM     (: 000100)
```

the assembler generates only one constant of + : 000100 and inserts the address of the storage location in which this constant is stored in the m portion of both the LA and the DM instructions.

An expression that is a literal should not have anything outside the parentheses.

The following is coding using literals for the example being used in this chapter.



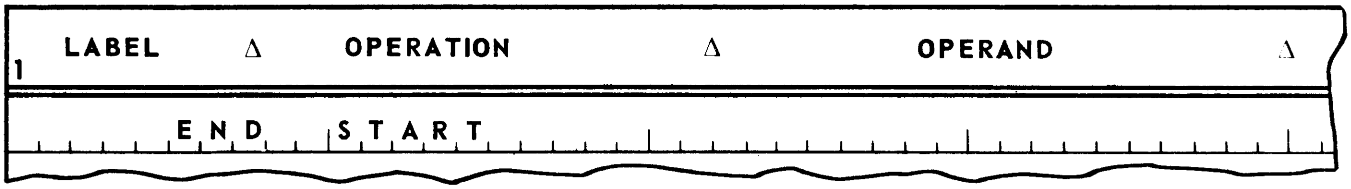
```
LABEL    Δ  OPERATION           Δ       OPERAND           Δ       COMMENTS
         USE    3
START    SLJ        FRD             ADV P
         LX     15, FILEP
         SLJ        FWR             ADV T
         LX     14, FILET
         LA     8,  3,15            .2 ( PG -600PP -PA ) - - - TT
         DM         (- : 060000)
         DS     6,  5,15
         DA     6,  2,15
         SA     6,  K2
         LA     8,  (: 000020)
         DM         K2
         SA     4,  K4
         DM         K1
         DA     6,  K4
         LA     8,  0,15            PN - - - TN
         SA     14, 2,14
         J          START
K1       +      0
K2       +      0
K3       +     :000000
K4       +      0
```

L. STUDENT EXERCISE

Code the previously stated student exercise using literals.

M. THE END DIRECTIVE

Another assembler directive is the END directive. The END directive is a sentinel that tells the assembler that the last line of a program to be assembled has been delivered to the assembler. The label of the first line to be executed in the program must be placed in the operand of the END directive. Thus, the previous coding should be followed by the line.



```
1  LABEL  Δ   OPERATION   Δ   OPERAND   Δ
          END  START
```

Being a communication between the programmer and the assembler, the END directive, as a consequence, does not cause the generation of any lines of object coding.

(Note: A separately assembled subroutine does not have an operand entry in the END directive.)

N. LINE CONTROL

To the UTMOST assembler a line consists of a lbel, an operation and an operand. This *assembler line* is to be distinguished from the 80 character positions that constitute a line on a sheet of UTMOST coding paper. This later type of line will be referred to as a *coding line*.

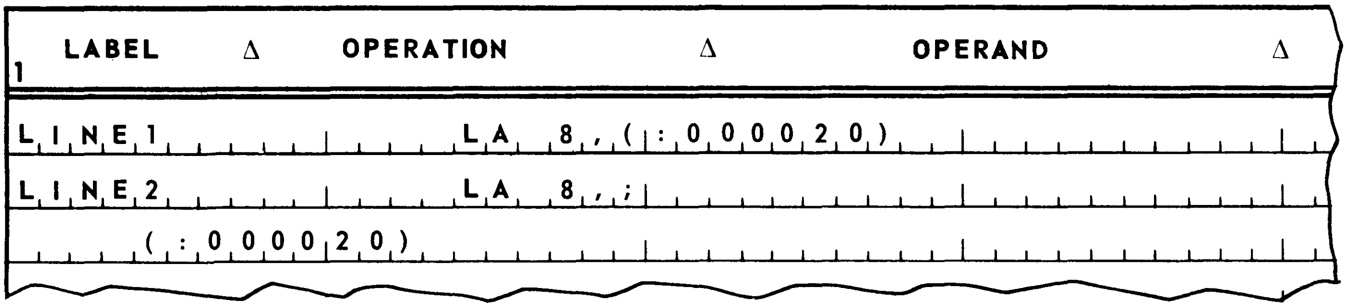
In general, an assembler line begins at the same point as a coding line. The assembler moves from left to right along the coding line and picks up characters one by one until it has isolated a label (or determines that a label is not present for this line), an operation, and as many expressions as are called for by the operation to constitute an operand. The next space character that the assembler encounters on the coding line constiutes the end of this assembler line. The assembler then goes to the first character of the next coding line to begin construction of the next assembler line. Thus, the assembler will not attend to any characters written on a coding line following the space that terminates the assembler line. The programmer may use this unused remainder of a coding line to write any comments he wishes.

If the assembler reaches the end of a coding line before it finds all the expressions required to make up the operand of an assembler line, the assembler gives the remaining expressions a value of zero. The programmer can cause the assembler to consider a coding line to be terminated at any point along the line he wishes by writing a "period" (.) followed by space. When the assembler encounters this "period-space", it considers the coding line complete and goes to the next coding line to begin construction of a new assembler line. The only place this device cannot be used is in the middle of an alphabetic expression, since in this case the assembler assumes that the "period-space" is part of the alphabetic expression. Thus:

' A . Δ

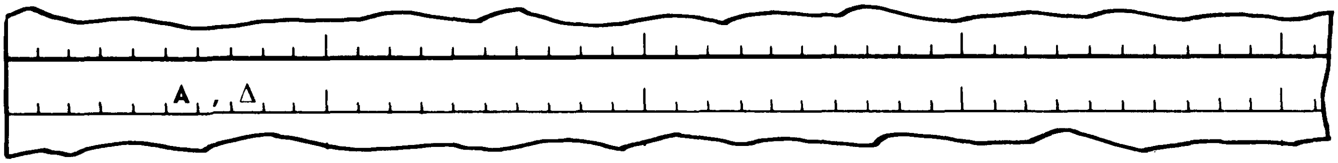
will not cause termination of an assembler line.

If the programmer cannot write everything he wants to be considered one assembler line on one coding line, he may terminate the first coding line with a "semicolon" (;) and continue writing the assembler line on the coding line that follows. Thus, the following two assembler lines are identical.



```
  LABEL     Δ     OPERATION              Δ                OPERAND              Δ
1
LINE1         LA  8,(:000020)
LINE2         LA  8,;
    (:000020)
```

The only place this *continuation mark* cannot be used is in the middle of an alphabetic expression. Thus:

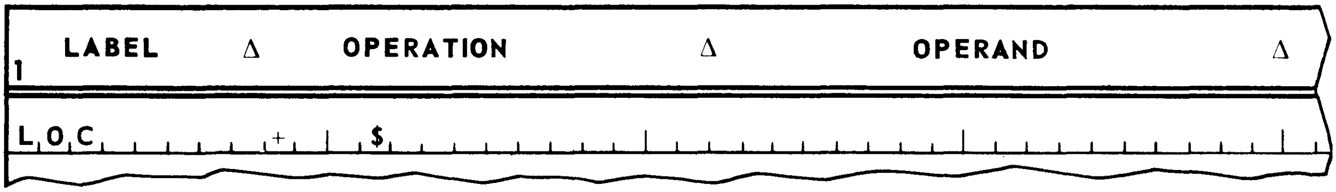


```

     A,Δ
```

will not cause continuation to the next coding line.

O. OTHER UNITS

If a "dollar sign" ($) is written as a unit of an expression in the operand, the UTMOST assembler will assign to the unit the binary value equivalent to the address of the storage location in which the line including this unit is ultimately stored when the object program is executed. This binary value is referred to as the present value of the location counter.

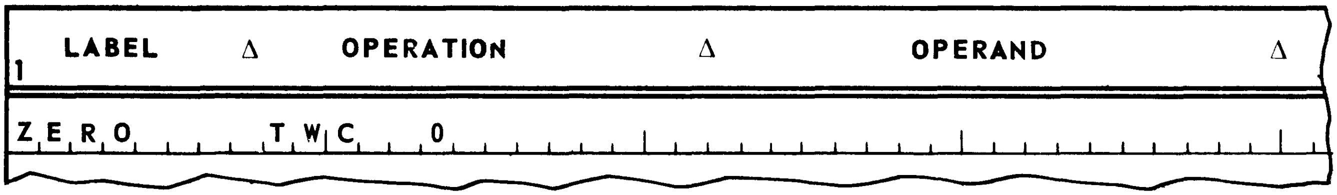


```
  LABEL     Δ     OPERATION              Δ                OPERAND              Δ
1
LOC     +  $
```

Presuming that at *object time* the line labelled LOC is stored in storage location 10000, this location will contain the binary equivalent of a decimal 10,000 right justified in the word and preceded by binary zeros.

P. TWO WORD CONSTANTS

A two word constant may be generated by placing TWC in the operation and the constant in the operand The assembler will generate the value of the operand, right justify this value in two words, fill with binary zeros, and assign the address of the first word to the label of the coding line. Both words will contain the same sign. For example, the line:



```
  LABEL     Δ     OPERATION              Δ                OPERAND              Δ
1
ZERO    TWC  0
```

will generate a two word constant of binary zeros. This constant could be loaded into AR's 8 and 4 by means of the following instruction.

```
     LA   12,ZERO+1
```

A two word constant may be written as a literal. For example, the following is equivalent to the above line of coding.

```
     LA   12,(TWC 0)
```

A floating point number may be represented in a unit by including a decimal point in the decimal value with TWC. The object code value will be in excess 50 floating point format with a ten digit mantissa and a two digit characteristic. For example, the line:

```
FLOAT     TWC  :3.14
```

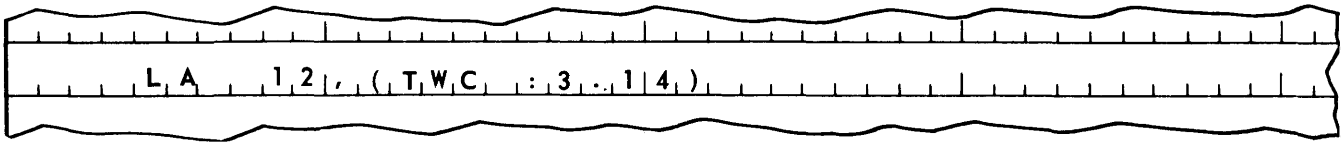
will cause the assembler to generate the following decimal digits:

$$\underbrace{51}_{\substack{\text{excess 50}\\\text{characteristic}}}\underbrace{314000000}_{\substack{\text{normalized}\\\text{mantissa}}}$$

(In standard floating point notation, this is the value .314 x 10^1). These digits are stored in two successive storage locations. The label assigned to the source code line is equated to the address of the first of the two words. Thus, this floating point number could be loaded into arithmetic registers 8 and 4 with the following instruction.

```
1  LABEL    Δ     OPERATION          Δ          OPERAND          Δ
     LA   12,FLOAT+1
```

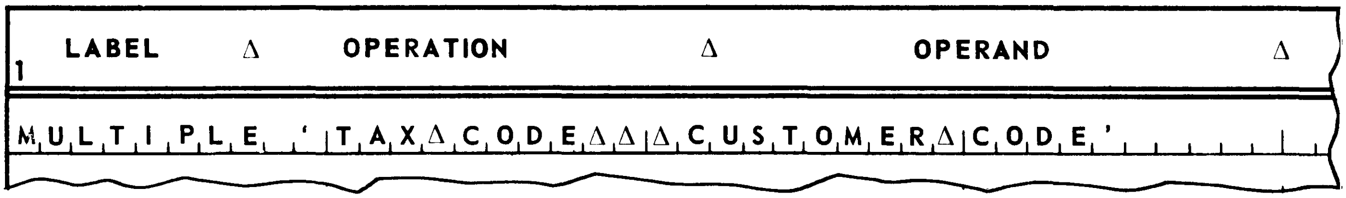
A floating decimal number may be written as a literal. For example, the following is equivalent to the above line of coding.



```
     LA   12,(TWC :3.14)
```

Q. MULTIPLE WORD CONSTANTS

A multiple word constant of a maximum of 78 alphanumeric characters can be generated by enclosing the characters in apostrophes. The resulting object code will be left justified in a sequence of words and will be filled with binary zeros to an integral number of words. For example:



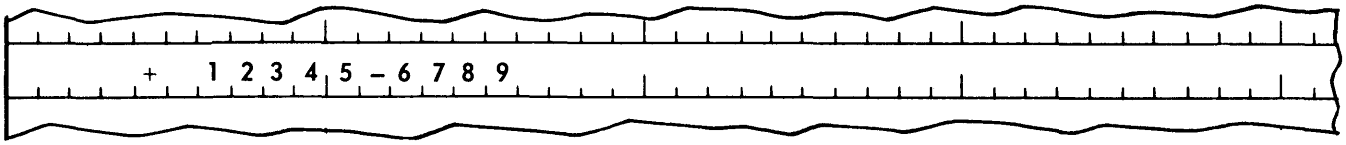
```
1 LABEL   Δ   OPERATION   Δ   OPERAND   Δ
MULTIPLE  'TAXΔCODEΔΔΔCUSTOMERΔCODE'
```

The label of a multiple word constant is associated with the zero word of the series of words generated in object code.

A multiple word constant cannot be written as a literal.

R. OTHER OPERATORS

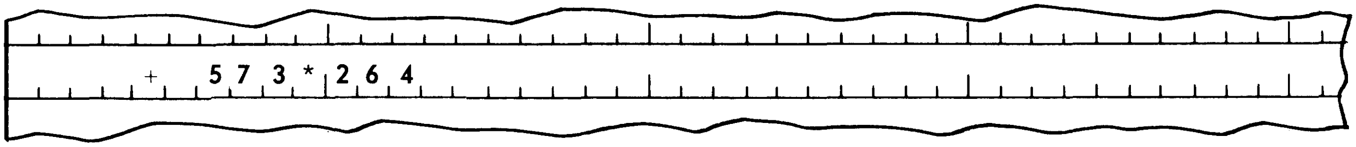
The arithmetic difference operator (–) may be used to subtract one unit from another. For example, the line:



```
     +  12345-6789
```

would generate a binary 5556 right justified in a binary zero filled word.

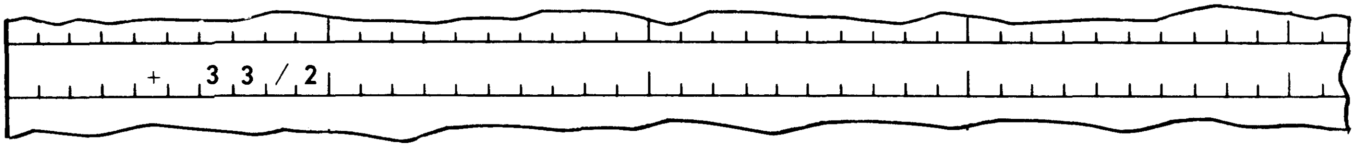
The arithmetic product operator (*) may be used to multiply one unit by another. For example, the line:



```
    + 573*264
```

will generate a binary 151272 right justified in a binary zero filled word.

The arithmetic quotient operator (/) may be used to divide one unit by another. The resulting unit will be the quotient of the division. The remainder is dropped. For example, the line:



```
    + 33/2
```

will generate a binary 16 right justified in a binary zero filled word.

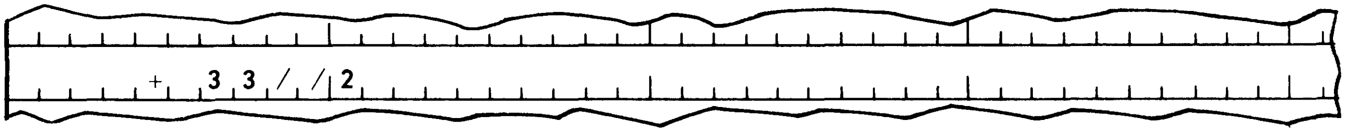
The covered quotient operator (//) may be used to divide one unit by another. The covered quotient is defined as follows.

$$A \,//\, B = (A + B - 1) \,/\, B$$

The result is as follows.

1. If arithmetic division produces no remainder, the arithmetic quotient and the covered quotient are the same.
2. If arithmetic division produces a remainder, the covered quotient is equal to the arithmetic quotient plus one.

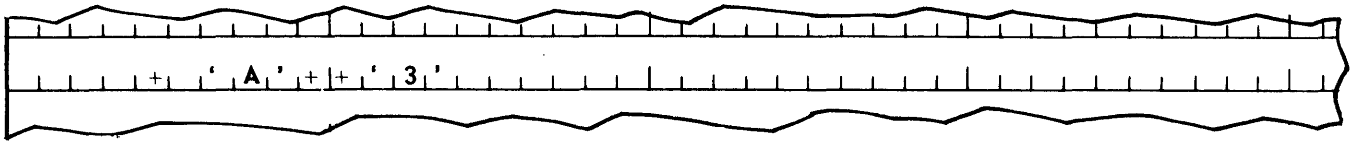
   For example, the line:



```
    + 33//2
```

will generate a binary 17 right justified in a binary zero filled word.

The logical sum operator (+ +) may be used to logically add one unit to another. For example, given the line:

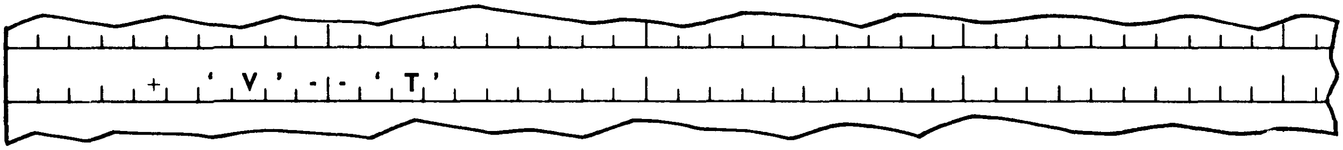


```
    + 'A'++'3'
```

The code for ' A ' is	010100
The code for ' 3 ' is	000110
Logical sum	010110

Consequently, a binary 22 will be right justified in a binary zero filled word.

The logical difference operator (– –) may be used to logically subtract one unit from another. For example, given the line:

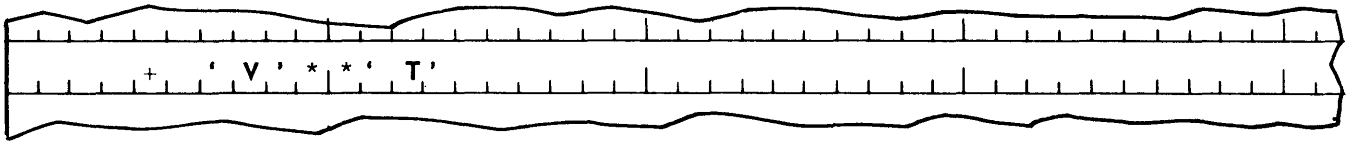


```
    + 'V'--'T'
```

The code for ' V ' is	111000
The code for ' T ' is	110110
Logical difference	001110

Thus, a binary 14 will be right justified in a binary zero filled word.

The logical product operator (* *) may be used to logically multiply one unit by another. For example, given the line:

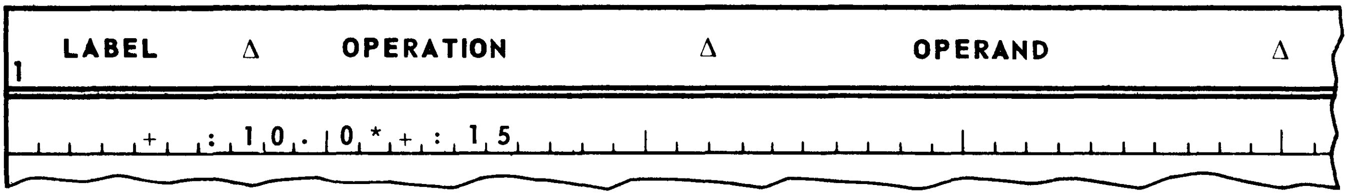


```
    + 'V'**'T'
```

The code for ' V ' is	111000
The code for ' T ' is	110110
Logical product	110000

Thus, a binary 48 will be right justified in a binary zero filled word.

The positive exponent operator (* +) may be used to generate a two word floating point constant in excess 50 notation. A * + B is equivalent to A * 10^B. For example, the line:

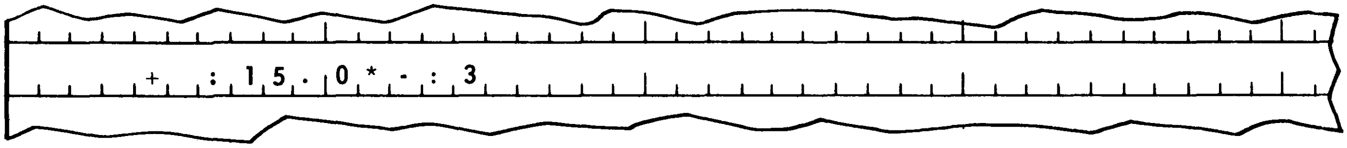


LABEL
Δ
OPERATION
Δ
OPERAND
Δ
1

```
    + :10.0*+:15
```

will generate the two word constant 671000000000.

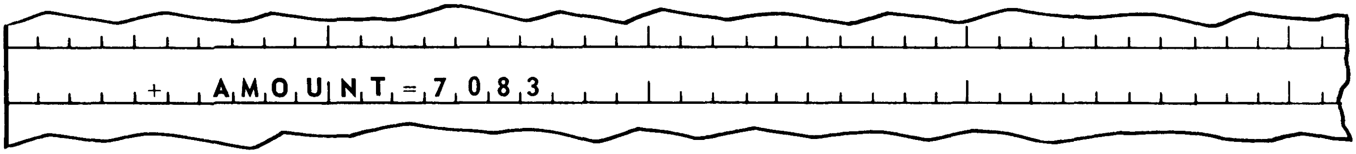
A * – B is equivalent to A * 10-^{B}. For example, the line:



```
    + :15.0*-:3
```

will generate the two word constant 491500000000.

The equals operator (=), the greater than operator (>), or the less than operator (<) may be used to compare two units. If the condition specified by the operator holds between the units, the result is a binary one. Otherwise, the result is a binary zero. For example, given the line:



```
    +  AMOUNT=7083
```

If the binary value assigned to the label AMOUNT is equal to the binary equivalent of the decimal number 7083, a binary one will be stored in the word generated for this line. Otherwise, a binary zero will be stored.

The *mode* of each unit in an expression can be different. One can be alphabetic, another decimal, another binary, and so on. As the assembler evaluates each unit a determination of the mode of the result is made. The determination depends on the operator and on the modes of the units. For purposes of making this determination, the operators are grouped as follows.

GROUP	OPERATORS
A	= > <
B	++ –– **
C	+ – * / //
D	*+ *–

The following chart shows the mode resulting from the combination of two units with an operator.

MODE OF FIRST UNIT	OPERATOR GROUP	MODE OF SECOND UNIT	MODE OF RESULT
Any	A	Any	Binary
Any	B	Any	Binary
Binary	C	Binary	Binary
Binary	C	BCD XS-3	Binary
BCD XS-3	C	Binary	Binary
BCD XS-3	C	BCD XS-3	BCD XS-3
Any	C	Floating	Floating
Floating	C	Any	Floating
Any	D	Any	Floating

So far in this chapter, all examples of expressions made up of units connected by operators have consisted of two units connected by one operator. However, there is no limit on the number of units and operators that may be combined to form an expression. For example, the following is a legitimate expression:

```
9 - 2 * 3
```

With an expression involving more than one operator, the question of *priority* of operators arises. For example, in the above illustration, is the expression nine minus the product of two multiplied by three, or is it three multiplied by the difference of two subtracted from nine? To assist in removing this ambiguity, the UTMOST assembler assigns priorities to the operators as follows.

PRIORITY	OPERATORS
1	*+ *-
2	* / //
3	+ -
4	**
5	++ --
6	= > <

The priorities are listed above from highest to lowest. Thus:

```
9 - 2 * 3
```

is nine minus the product of two multiplied by three.

The above list indicates that several operators have the same priority. For example "*", "/" and "//" all have priority two. This fact raises the question exemplified as follows. What is the following expression?

```
4 * 5 //2
```

Is it four multiplied by the covered quotient of five divided by two, or is it the covered quotient of two divided into the product of four multiplied by five? To remove this ambiguity, the UTMOST assembler interprets operators having the same priority from left to right. Thus:

```
4 * 5//2
```

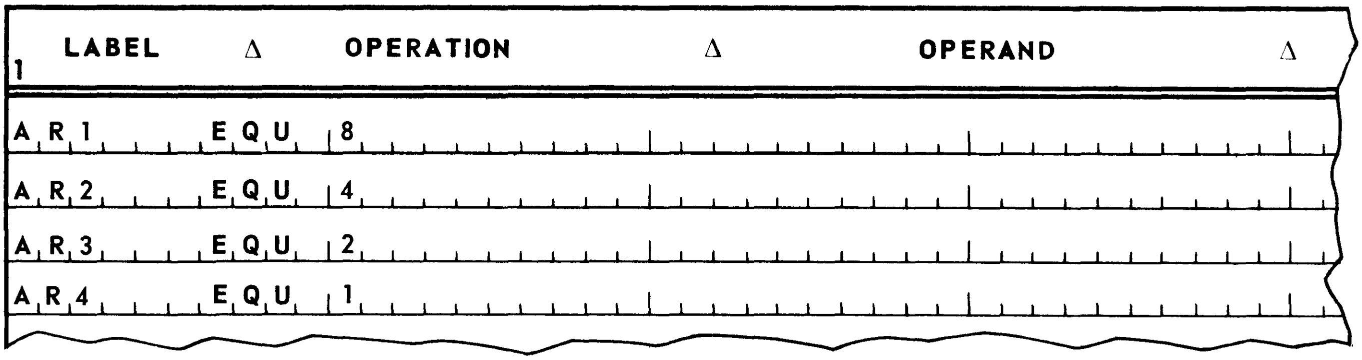
is the covered quotient of two divided into the product of four multiplied by five.

The above rules of priority can be overridden by use of parenthesization. For example, if the product of four multiplied by the covered quotient of five divided by two is desired, it may be written as follows:

```
4 * (5 // 2)
```

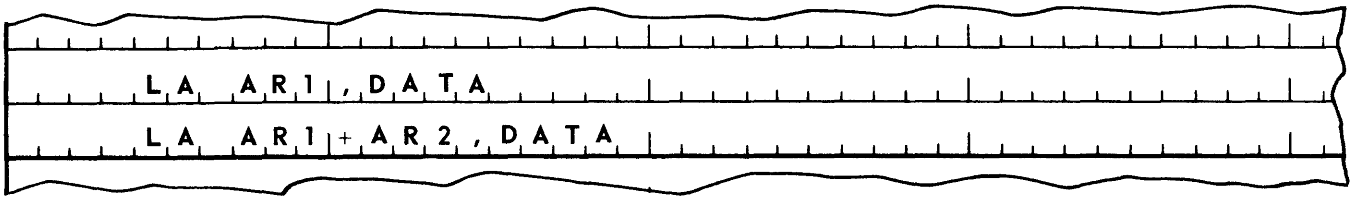
S. OTHER ASSEMBLER DIRECTIVES

The assembler directive EQU in the operation field of a line causes the assembler to equate the label in the label field of this line to the value of the expression in the operand field. All succeeding lines of coding with this label in the operand field will have this value substituted for the label. For example, given the following:

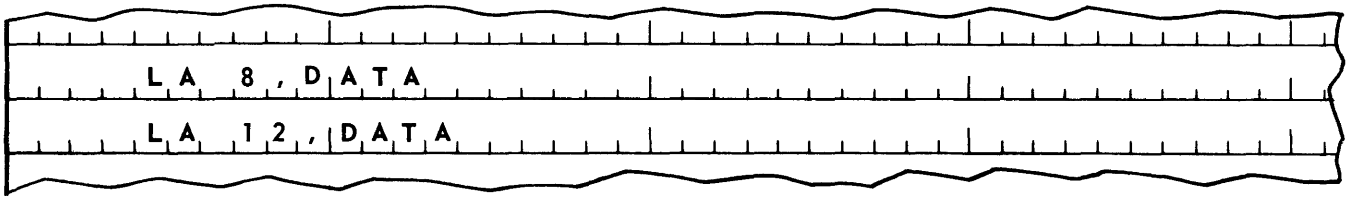


```
1 LABEL     Δ   OPERATION        Δ         OPERAND          Δ
AR1        EQU  8
AR2        EQU  4
AR3        EQU  2
AR4        EQU  1
```

Then the following two columns of instructions are equivalent.



```
        LA  AR1,DATA
        LA  AR1+AR2,DATA
```

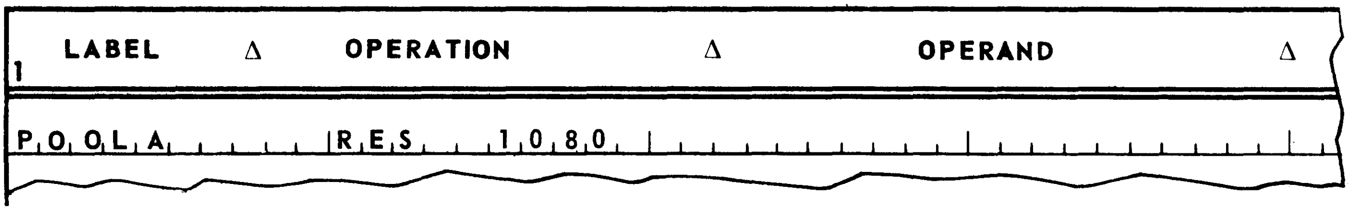


```
        LA  8,DATA
        LA  12,DATA
```

The EQU directive is a communication between the programmer and the assembler. As a consequence, it will not cause the generation of any lines of coding.

The EQU directive must appear prior to any use of the label being equated. Notice especially the use of a label as truly symbolic (i.e. it need not be related to a line of coding).

The assembler directive RES causes the assembler to set aside a number of consecutive storage locations equal to the value of the operand. They are set aside at the point at which the RES directive appears in the coding. For example, the line:



```
1 LABEL     Δ   OPERATION        Δ         OPERAND          Δ
POOLA          RES    1080
```

sets aside 1080 consecutive storage locations which can be addressed as POOLA, POOLA + 1, POOLA + 2, and so on, through POOLA + 1079.

The assembler directive FORM may be used to define a word format, label the format, and allow the format to be thereafter referenced by the label used as an operation. This directive must be given a label and must have an operand consisting of a series of expressions whose sum is 25. A single expression of 25 is not permissable. For example:

```
TAB2WORD        FORM 0,6,3,15
TAB             TAB2WORD 0,'W',0,TYPE
```

The line labelled TAB is a positive constant with the bit code for 'W' in bit positions 19-24, binary zeros in bit positions 16-18, and the address assigned to the label TYPE in bit positions 1-15. The FORM directive will not cause the generation of any lines of coding.

The assembler directive FLD may be used to define limits of a field, label this field definition, and thereafter use this label to refer to this field definition. This directive must be given a label. The operand consists of two expressions. The first specifies the leftmost bit of the field, the second the rightmost bit. After a FLD directive has been used to define a field, any succeeding line of coding may have an m portion consisting of the label of the FLD directive followed in parentheses by the designation of the word or words from which the field is to be selected. For example:

```
1 LABEL     Δ    OPERATION          Δ          OPERAND          Δ
LMT     FLD     12,1
        LA      8,LMT(VALUE)
```

is equivalent to:

```
K1      +       12,1,VALUE
        LA      8,*K1
```

The above line of coding could also be written as follows:

```
        LA      8,*(12,1,VALUE)
```

The assembler directive DO may be used to generate a line of coding a variable number of times. The format of the DO directive is as follows.

```
  LABEL     Δ     OPERATION        Δ          OPERAND          Δ
1
LABEL DO eΔ,line
```

The expression e specifies how many times the line is to be generated. The expression must be followed by a "space-comma".

The line has a normal form. That is, if it is to be labelled, the label must appear immediately after the comma. If not, the comma must be followed by at least one space. For example:

```
  LABEL     Δ     OPERATION        Δ          OPERAND          Δ
1
    DO   6 ,  +0
```

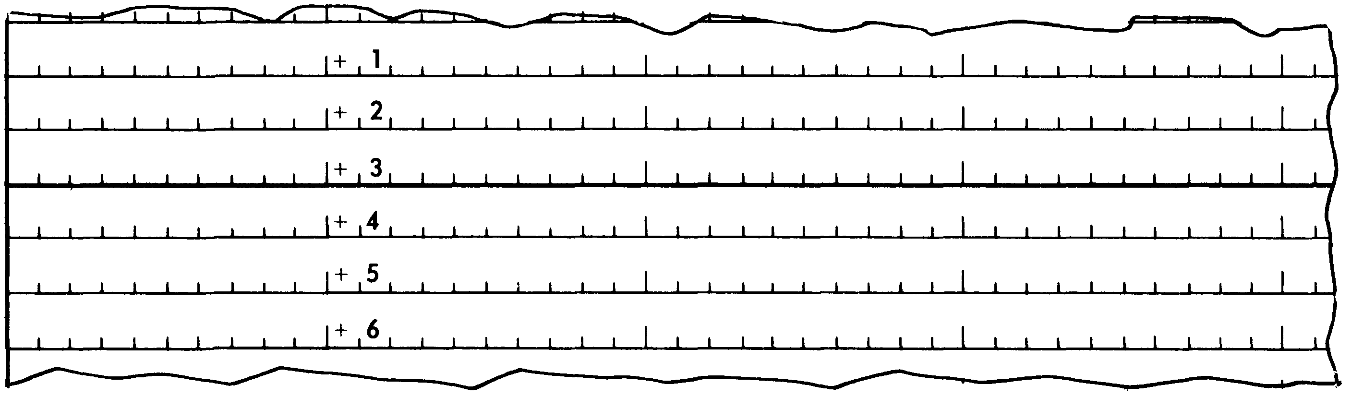
is equivalent to

```
           +0
           +0
           +0
           +0
           +0
           +0
```

Notice that both the DO directive and the line may have a label. The assembler treats the label of the line as the normal label. It treats the label of the DO directive in a special fashion by equating it to the number of times the line has been generated. For example:

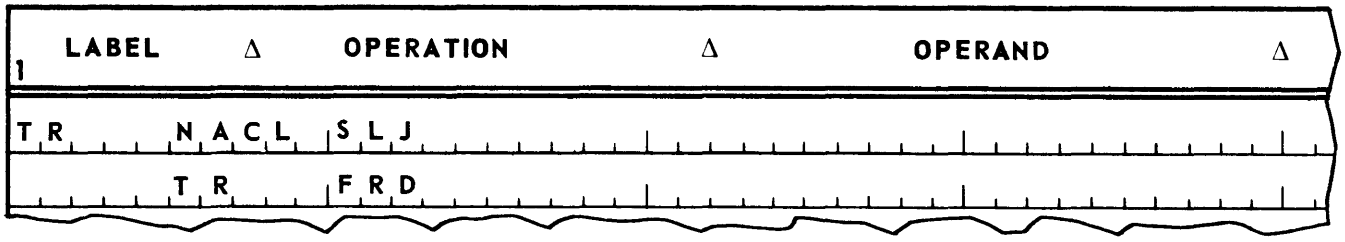
```
  LABEL     Δ     OPERATION        Δ          OPERAND          Δ
1
LABEL  DO 6  ,  +LABEL
```

is equivalent to:



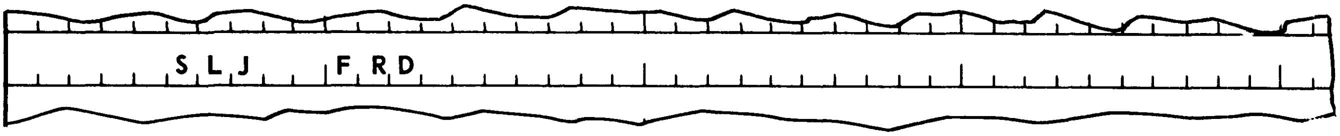
```
          +1
          +2
          +3
          +4
          +5
          +6
```

The assembler directive NACL may be used to substitute a different mnemonic for the standard UTMOST mnemonic used in the operation field of an instruction. The standard mnemonic is written in the operand of the NACL directive. The mnemonic to be substituted is written in the label. After a NACL directive has been submitted, all following instructions must use the new mnemonic at least until the occurence of another NACL directive redefining the mnemonic. For example:



```
1  LABEL     Δ     OPERATION          Δ               OPERAND            Δ
TR      NACL  SLJ
        TR    FRD
```

is equivalent to:



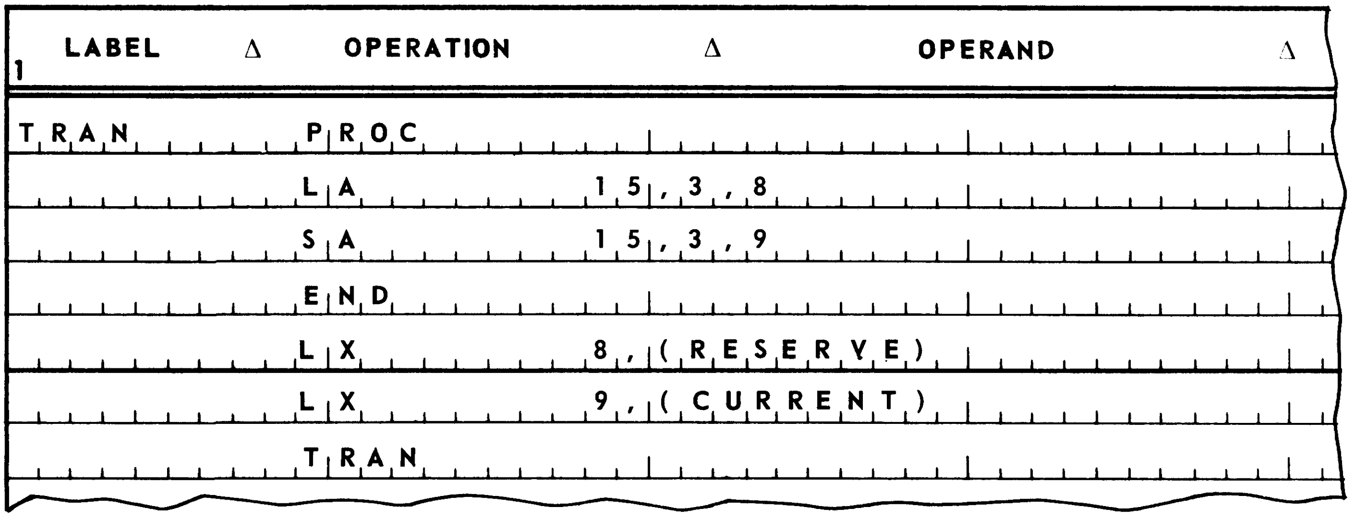
```
        SLJ   FRD
```

T. PROCEDURES

A part of a program may be written separately as a *procedure* rather than as an integral piece of the overall program. The beginning of a procedure is marked off by a PROC assembler directive. The end of a procedure is indicated by an END directive. However, unlike an END directive at the end of a program, the END directive at the end of a procedure has no entry in the operand.

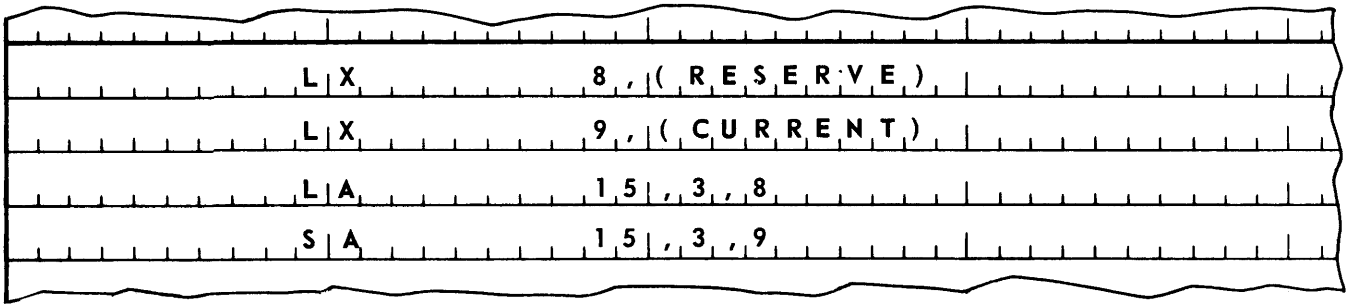
The PROC directive must have a label. The operand of a PROC directive may contain two expressions. The first can be used to specify the maximum number of "lists" associated with the procedure. (What a "list" is will be defined later in this section.) However, this operand is optional. The second expression of lines generated by the PROC, if the PROC generates a fixed number of lines. This expression is also optional.

A procedure is not assembled where it is written in a program. Instead, it is assembled when it is "referenced" in the program. A procedure may be referenced by writing its label in the operation field of a line. A procedure must appear in the source program before any reference to it. For example:



```
1 LABEL   Δ   OPERATION   Δ   OPERAND   Δ
TRAN          PROC
              LA          15,3,8
              SA          15,3,9
              END
              LX          8,(RESERVE)
              LX          9,(CURRENT)
              TRAN
```

is equivalent to:



```
              LX          8,(RESERVE)
              LX          9,(CURRENT)
              LA          15,3,8
              SA          15,3,9
```

Different versions of a process can be written in one procedure. The versions are distinguished by beginning them with a NAME assembler directive. The NAME directive must have a label and may have any entry desired in the operand. The version of the procedure desired is referenced by writing the label of the NAME directive in the operation of a line. Only that part of the procedure coding which appears below the NAME directive referenced is assembled at the reference point. For example, given the following procedure:

```
LABEL     Δ OPERATION       Δ      OPERAND         Δ
TRAN        PROC
EIGHT       NAME
            LA               15,7,8
            SA               15,7,9
FOUR        NAME
            LA               15,3,8
            SA               15,3,9
            END
```

The reference:

```
LABEL     Δ OPERATION       Δ      OPERAND         Δ
    EIGHT
```

is equivalent to:

```
    LA        15,7,8
    SA        15,7,9
    LA        15,3,8
    SA        15,3,9
```

While the reference:

```
    FOUR
```

is equivalent to:

```
    LA        15,3,8
    SA        15,3,9
```

The assembler directive GO may be used with a procedure to direct the assembler to transfer control in the assembly of a procedure. The operand of a GO directive must be the label of a NAME directive. For example, given the following procedure:

```
LABEL        OPERATION      OPERAND
TRAN         PROC
FOUR         NAME           0
             LA             15,3,8
             SA             15,3,9
             GO             ENDP
THREE        NAME           1
             LA             14,2,8
             SA             14,2,9
ENDP         NAME
             END
```

The reference:

```
             THREE
```

is equivalent to:

```
             LA             14,2,8
             SA             14,2,9
```

While the reference:

```
             FOUR
```

is equivalent to:

```
LABEL   Δ   OPERATION   Δ   OPERAND   Δ
          LA        15,3,8
          SA        15,3,9
```

A procedure may make use of *variables*, which are submitted to a procedure at the time it is referenced. The variables are submitted by means of *lists*, which make up the operand of the line referencing the procedure. A list consists of a series of expressions separated by commas. More than one list may follow a procedure reference. Lists are separated by spaces. For example, the following is a reference to the NAME directive labelled IT of the procedure labelled MOVE. (The MOVE procedure is shown in part S of this section.)

```
     IT   IN        OUT       50      14,13
```

This procedure reference is followed by four lists, The first contains one expression (IN), as does the second (OUT) and third (50). The fourth list contains two expressions (14, 13). In this case, the variables being submitted are as follows.

1. The label of the zero word of the area from which words are to be moved (IN).
2. The label of the zero word of the area into which words are to be moved (OUT).
3. The number of words to be moved (50).
4. The number of the index register to be used to address the "from" area (14).
5. The number of the index register to be used to address the "into" area (13).

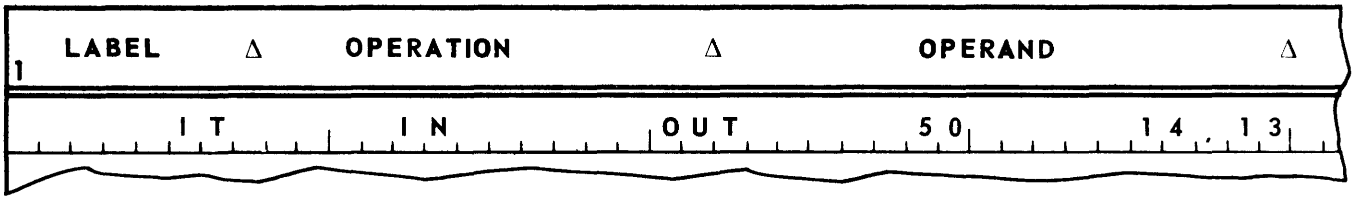
Within the procedure coding, variables are referenced by means of the following expressions.

label (s, e)

Where "label" is the procedure label, "s" is the number of the list desired, and "e" is the number of the desired expression within the list. For example, in the following line:

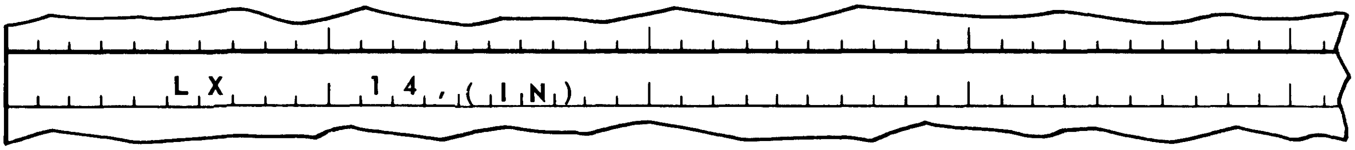
```
LABEL   Δ   OPERATION   Δ   OPERAND   Δ
     LX   MOVE(4,1),(MOVE(1,1))
```

MOVE (4, 1) references the first expression in the fourth list of the reference to the MOVE procedure. MOVE (1, 1) references the first expression in the first list. Thus, if the reference to MOVE procedure is:



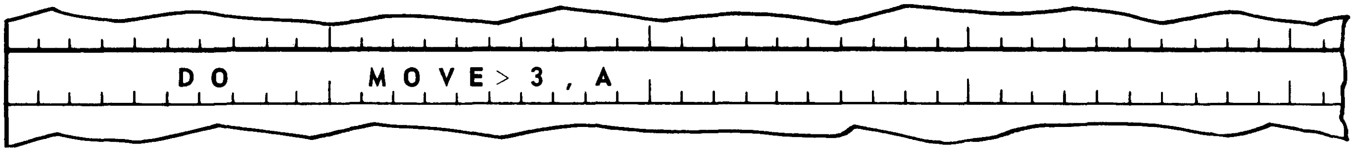
```
1 LABEL   Δ   OPERATION   Δ   OPERAND   Δ
     IT     IN      OUT     50     14,13
```

the line in the procedure becomes:



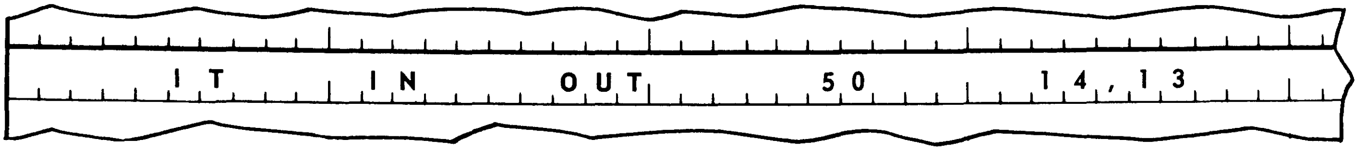
```
     LX    14,(IN)
```

To reference, within the procedure, the number of lists supplied by the reference to the procedure, the label of the procedure is used, For example, in the following line:



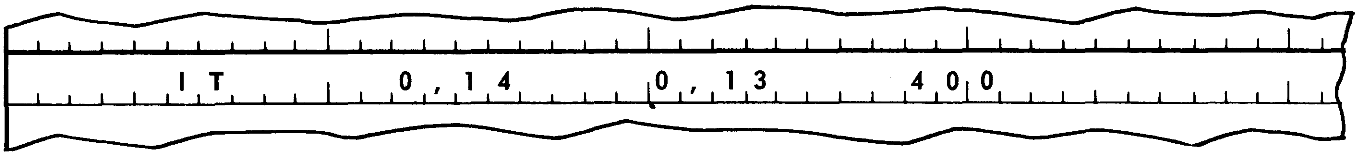
```
     DO    MOVE>3,A
```

the number of lists supplied by the reference to the procedure is substituted for MOVE. If the reference is:



```
     IT     IN    OUT      50     14,13
```

four is substituted for MOVE. If the reference is:



```
     IT     0,14    0,13    400
```

three is substituted for MOVE.

To reference, within a procedure, the number of expressions in a list, the following expression is used:

label (s)

where "label" is the label of the procedure and "s" is the number of the list desired. For, example, in the following line:

```
     DO       MOVE(1)>1,A
```

the number of expressions supplied in list one of the reference to the procedure is substituted for MOVE (1). Thus, if the reference is:

```
     IT       IN        OUT       50        14,13
```

one is substituted for MOVE(1). If the reference is:

```
     IT       0,14      0,13      400
```

two is substituted for MOVE (1).

To reference, within a procedure, the operand of the NAME directive whose label was used to reference the procedure, the label of the procedure followed by (0, 0) is used. For example, the first three lines of the MOVE procedure are:

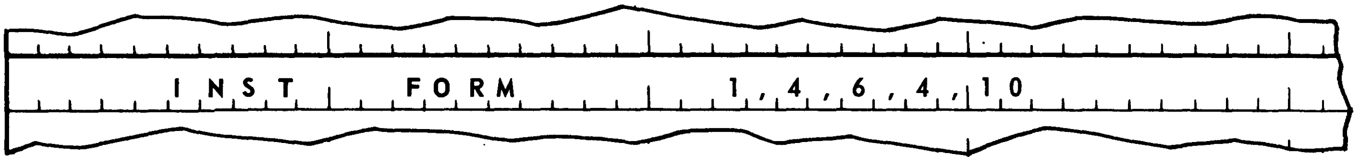
```
 LABEL    Δ   OPERATION          Δ            OPERAND              Δ
1
     MOVE     PROC
     IT       NAME                         0
     ST1      NAME                         1
```

In the following line:

```
     DO       MOVE(0,0)=0,A
```

1. If the MOVE procedure is referenced by the label IT, zero is substituted for MOVE (0, 0).
2. If referenced by the label ST1, one is substituted for MOVE (0, 0).

An expression in a list may be proceded by an asterisk (*). For example, suppose the following FORM directive, which lays out the "form" of an instruction:

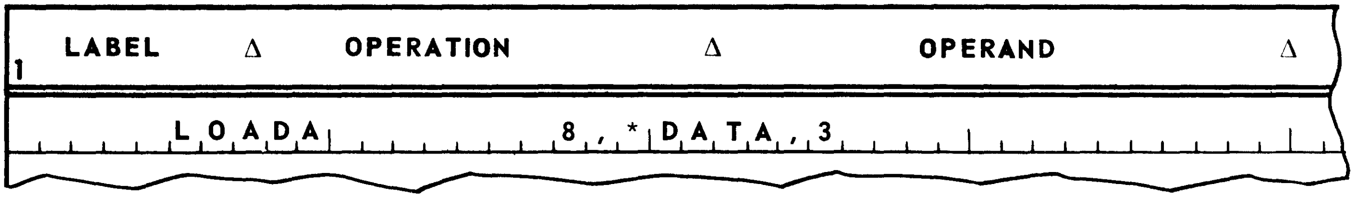


```
    INST   FORM    1,4,6,4,10
```

Also suppose a procedure labeled FAB containing a NAME directive labeled LOADA whose purpose it is to fabricate an LA instruction. This procedure expects one list consisting of the following three expressions to be specified in the order listed.

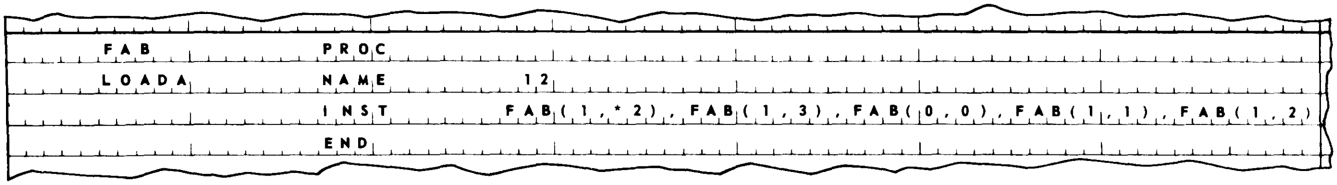
1. The arithmetic register(s) to be specified.
2. The label to appear in the m portion of the instruction. If indirect addressing is desired, the label is to be preceded by an asterisk.
3. The index register to be specified.

Thus, a call on this procedure by means of the NAME directive might appear as follows:



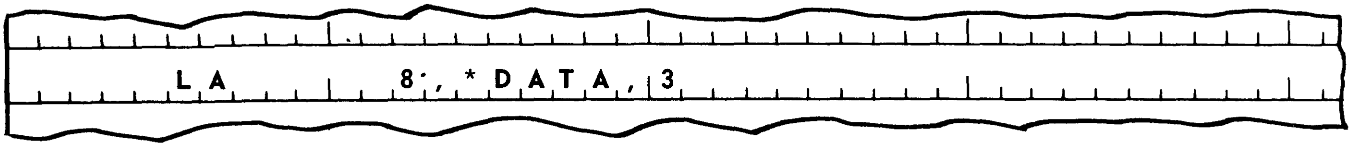
```
1  LABEL   Δ   OPERATION         Δ         OPERAND          Δ
    LOADA          8,*DATA,3
```

In the procedure, the reference FAB (1, 2) will supply only the label DATA. The preceding asterisk is referenced as follows: FAB (1, * 2). If there is an asterisk preceding expression 2, a binary one will be generated. If not, a binary zero is generated. The procedure FAB might appear as follows.



```
     FAB        PROC
     LOADA      NAME       12
                INST       FAB(1,*2),FAB(1,3),FAB(0,0),FAB(1,1),FAB(1,2)
                END
```

The above call would result in the generation of object code code equivalent to the following line:

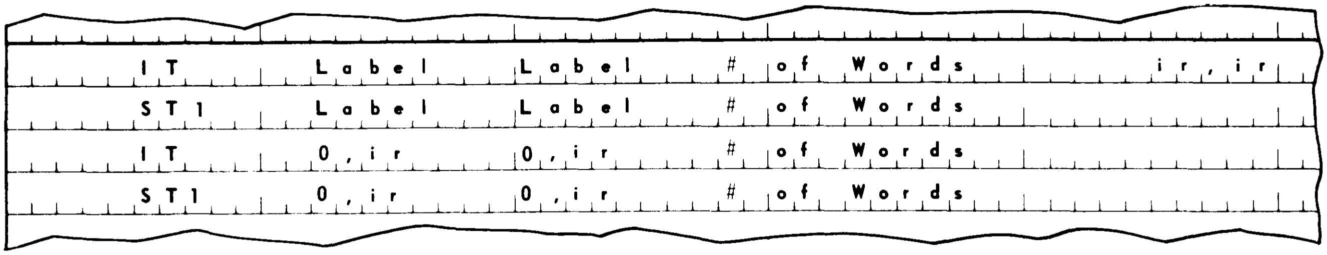


```
    LA       8,*DATA,3
```

If a procedure may be referenced with a variable number of lists, as is the case with the MOVE procedure, then the PROC directive for the procedure has no entry in the operand. As mentioned previously, the operand of a PROC directive may have no entry even if the number of lists for the procedure is not variable.

U. EXAMPLE

The MOVE procedure has as its function the movement of a specified number of words from one storage area to another. It may be referenced by any one of the four following coding lines:



```
      IT     Label     Label     # of Words           ir,ir
      ST1    Label     Label     # of Words
      IT     0,ir      0,ir      # of Words
      ST1    0,ir      0,ir      # of Words
```

If the MOVE procedure is referenced with the name IT, then iterative coding is used to make the move. If referenced with ST1, straight line coding is used. An exception to this rule occurs when iterative coding is called for and the number of words to be moved is less than or equal to 20, in which case straight line coding is supplied.

The first list specifies the address of the zero word of the area from which the words are to be moved. This specification can be made either as a label or as the number of an index register containing the address.

The second list specifies the address of the zero word of the area into which the words are to be moved. This specification, also, can be made as a label or as an index register number.

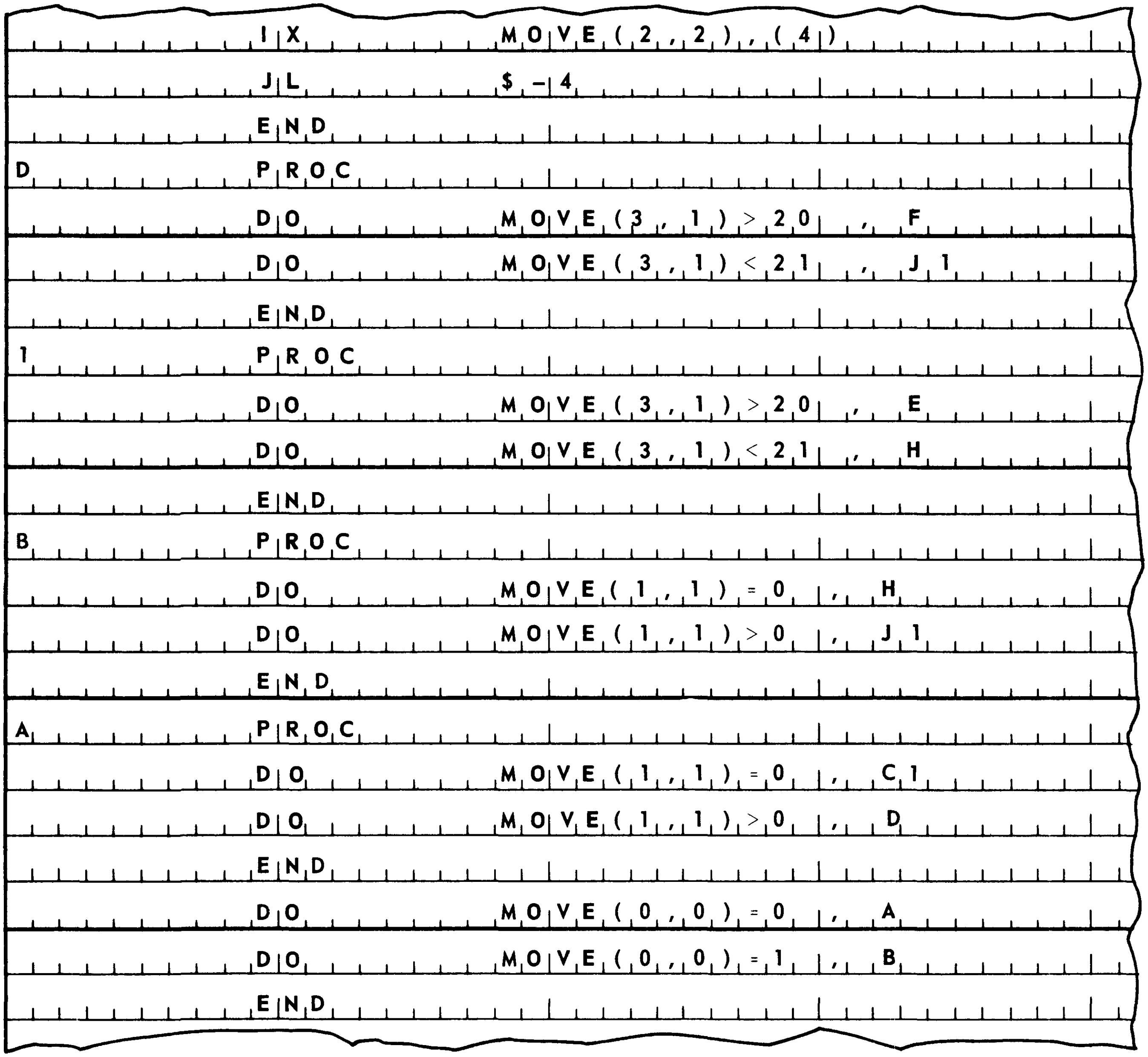
The third list specifies the number of words to be moved. If iterative coding is called for, and if the area locations are specified in label form, then a fourth list consisting of two expressions is necessary. The first expression specifies the number of the index register to be used to address the "from" area. The second specifies the number of the index register to be used to address the "into" area.

The coding for the MOVE procedure follows. An explanation of this coding follows the coding.

```
1 LABEL   Δ  OPERATION     Δ      OPERAND              Δ        COMMENTS
MOVE         PROC
IT           NAME          0
ST1          NAME          1
M            PROC
             LA            15,4*ADD-1,MOVE(1,2)
             SA            15,4*ADD-1,MOVE(2,2)
             END
L            PROC
             LA            L(1,1),MOVE(3,1)-1,MOVE(1,2)
             SA            L(1,1),MOVE(3,1)-1,MOVE(2,2)
             END
K            PROC
             LA            15,MOVE(1,1)+(4*COUNT-1)
             SA            15,MOVE(2,1)+(4*COUNT-1)
             END
G            PROC
             LA            G(1,1),MOVE(1,1)+MOVE(3,1)-1
             SA            G(1,1),MOVE(2,1)+MOVE(3,1)-1
             END
J1           PROC
             DO            MOVE(3,1)**3>0  ,  G  (3**MOVE(3,1))*5-4
COUNT        DO            MOVE(3,1)/4  ,  K
             END
H            PROC
             DO            MOVE(3,1)**3>0  ,  L  (3**MOVE(3,1))*5-4
ADD          DO            MOVE(3,1)/4  ,  M
             END
F            PROC
             LX            MOVE(4,1),(MOVE(1,1))
             LX            MOVE(4,2),(MOVE(2,1))
             LA            15,3,MOVE(4,1)
             SA            15,3,MOVE(4,2)
             IXC           MOVE(4,1),(((4*(MOVE(3,1)/4))+MOVE(1,1),4)
             IX            MOVE(4,2),(4)
             JL            $-4
             DO            MOVE(3,1)**3>0  ,  G  (3**MOVE(3,1))*5-4
             END
E            PROC
             DO            MOVE(3,1)**3>0  ,  L  (3**MOVE(3,1))*5-4
             SX            MOVE(1,2),TEMP
             LA            8,TEMP
             BA            8,(4*(MOVE(3,1)/4))
             BRR           8,16
             OR            8,(4)
             SA            8,TEMP
             J             $+2
TEMP         +             0
             LA            15,3,MOVE(1,2)
             SA            15,3,MOVE(2,2)
             IXC           MOVE(1,2),TEMP
```

(Continued on next page)

(MOVE PROC Continued)



```
            IX       MOVE(2,2),(4)
            JL       $-4
            END
D           PROC
            DO       MOVE(3,1)>20 , F
            DO       MOVE(3,1)<21 , J1
            END
1           PROC
            DO       MOVE(3,1)>20 , E
            DO       MOVE(3,1)<21 , H
            END
B           PROC
            DO       MOVE(1,1)=0 , H
            DO       MOVE(1,1)>0 , J1
            END
A           PROC
            DO       MOVE(1,1)=0 , C1
            DO       MOVE(1,1)>0 , D
            END
            DO       MOVE(0,0)=0 , A
            DO       MOVE(0,0)=1 , B
            END
```

The first three lines in the above coding define the MOVE procedure. There then follows a number of procedures to determine what coding is to be generated. The last two DO directives in the MOVE procedure determine whether STraight line or ITerative coding is called for. If ITerative coding is called for, procedure A is referenced. If STraight line, procedure B.

Procedure A determines whether the "from" and "into" areas are specified in terms of labels or index registers. If index registers, procedure C1 is referenced. If labels procedure D.

Procedure B makes the same determination as procedure A. If the areas are specified in terms of index registers, procedure H is referenced. If in terms of labels, procedure J1.

Procedure C1 determines whether the number of words to be moved is more than 20. If so, procedure E is referenced. If not, procedure H.

Procedure D makes the same determination as procedure C1. If the number of words is more than 20, procedure F is referenced. If not, procedure J1 is referenced.

Procedure E contains the coding provided if iterative coding is called for, if more than 20 words are to be moved, and if the areas are specified in terms of index registers. The DO directive at the beginning of procedure E determines whether the number of words to be moved is a multiple of four. If,not, procedure L is referenced to provide coding for the movement of the remaining words not a multiple of four.

Procedure F contains the coding provided if iterative coding is called for, if more than 20 words are to be moved, and if the areas are specified in terms of labels. The DO directive at the end of procedures F determines whether the number of words to be moved is a multiple of four. If not, procedure G is referenced to provide coding for the movement of the remaining words.

Procedure H contains two DO directives.The first determines whether the number of words to be moved is a multiple of four. If not, procedure L is referenced. The second DO directive references procedure M.

Procedure J1 also contains two DO directives. The first determines whether the number of words to be moved is a multiple of four. If not, procedure G is referenced. The second DO directive references procedure K.

Procedure G contains the coding to move nonmultiples of four words when the areas are specified in terms of labels.

Procedure K contains the coding provided to move the multiples of four words when straight line coding is called for and the areas are specified in terms of labels.

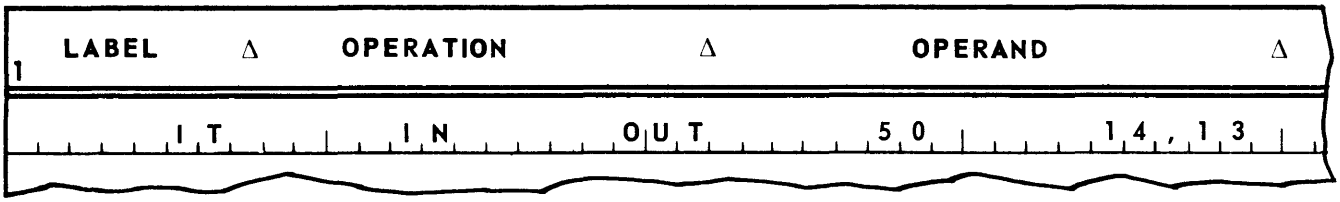
Procedure L contains the coding to move nonmultiples of four words when the areas are specified in terms of index registers.

Procedure M contains the coding provided to move the multiples of four words when straight line coding is called for and the areas are specified in terms of index registers.

The above discussion is summarized in the following table.

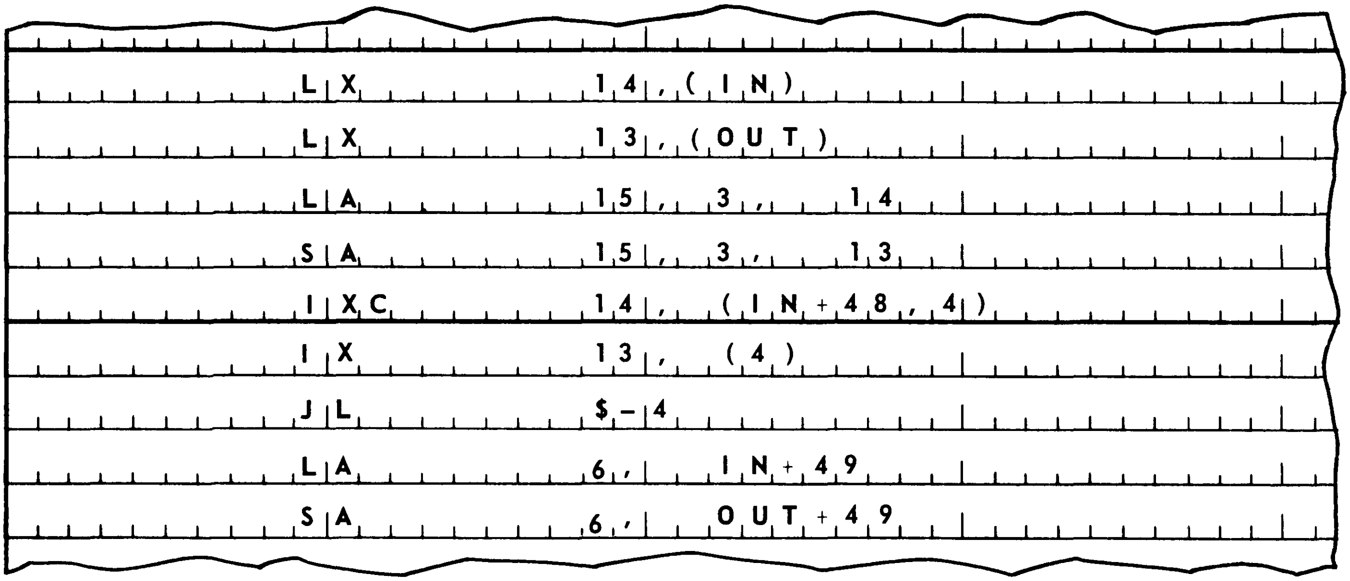
ITERATIVE CODING (A)				STRAIGHT LINE CODING (B)	
Areas specified in terms of Index Registers (C1)		Areas specified in terms of Labels (D)		Areas specified in terms of Index Registers (H)	Areas specified in terms of Labels (J1)
More than 20 words to be moved.	20 or less words to be moved (H)	More than 20 words to be moved.	20 or less words to be moved. (J1)	Nonmultiples of 4 moved by L; multiples moved by M	Nonmultiples of 4 moved by G; multiples moved by K
Nonmultiples of 4 moved by L; multiples moved by E.	(See column 5.)	Nonmultiples of 4 moved by G; multiples moved by F.	(See column 6.)		
Column 1	Column 2	Column 3	Column 4	Column 5	Column 6

For example, suppose the MOVE procedure is referenced as follows.



```
     LABEL     Δ   OPERATION          Δ          OPERAND            Δ
1
         IT          IN        OUT       50         14,13
```

The coding supplied would be as follows



```
           LX       14,(IN)
           LX       13,(OUT)
           LA       15,3,    14
           SA       15,3,    13
           IXC      14,  (IN+48,4)
           IX       13,  (4)
           JL       $-4
           LA       6,   IN+49
           SA       6,   OUT+49
```

If the MOVE procedure is referenced as follows:

```
LABEL      Δ   OPERATION          Δ         OPERAND              Δ
      IT        0,14        0,13       400 0
```

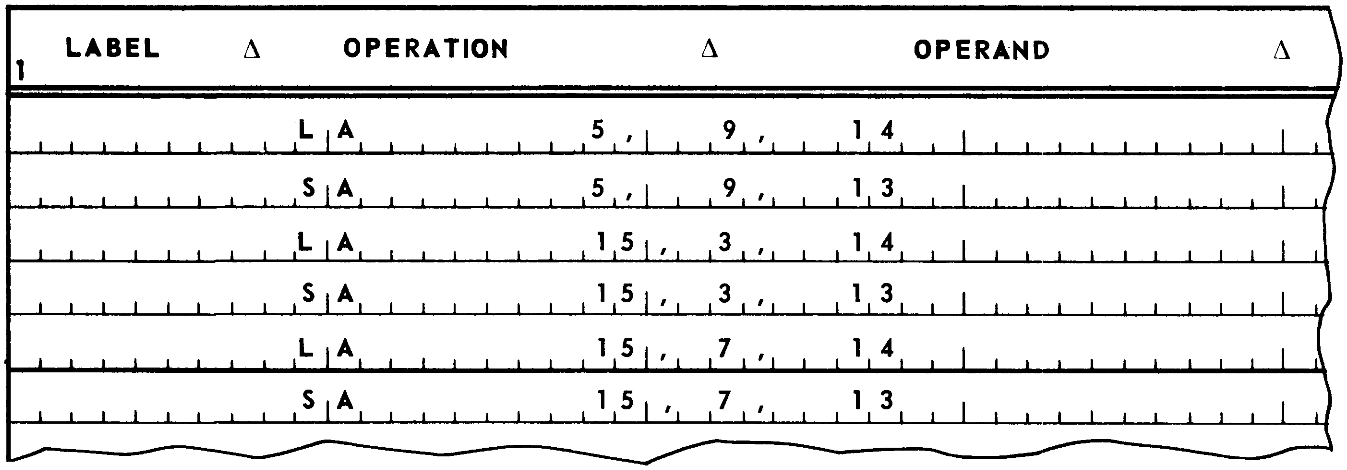
the coding supplied would be as follows.

```
LABEL      Δ   OPERATION          Δ         OPERAND              Δ
               SX              14,   TEMP
               LA              8,    TEMP
               BA              8,    (400)
               BRR             8,    16
               OR              8,    (4)
               SA              8,    TEMP
               J                     $+1
TEMP           +                     0
               LA              15,   3,14
               SA              15,   3,13
               IXC             14,   TEMP
               IX              13,   (4)
               JL              $-4
```

If referenced as follows:

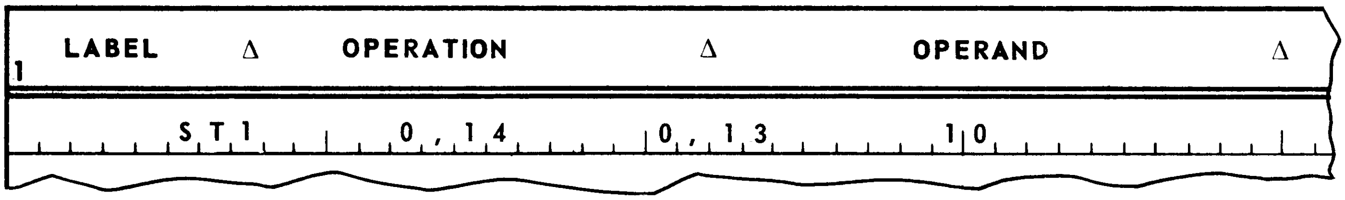
```
LABEL      Δ   OPERATION          Δ         OPERAND              Δ
      ST1       IN          OUT        20
```

the coding would be as follows:



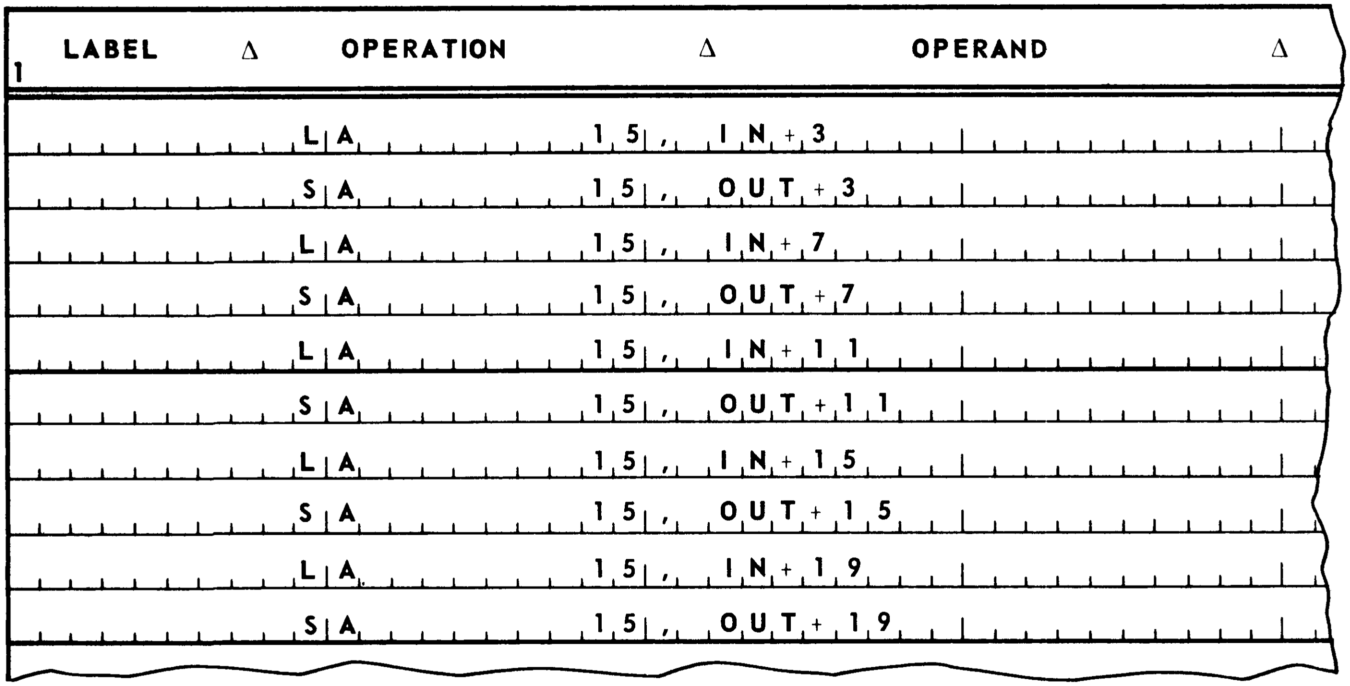
```
LABEL   Δ   OPERATION            Δ   OPERAND                Δ
1
            LA          5,9,14
            SA          5,9,13
            LA          15,3,14
            SA          15,3,13
            LA          15,7,14
            SA          15,7,13
```

If referenced as follows:



```
LABEL   Δ   OPERATION            Δ   OPERAND                Δ
1
    ST1         0,14            0,13            10
```

the coding would be as follows.

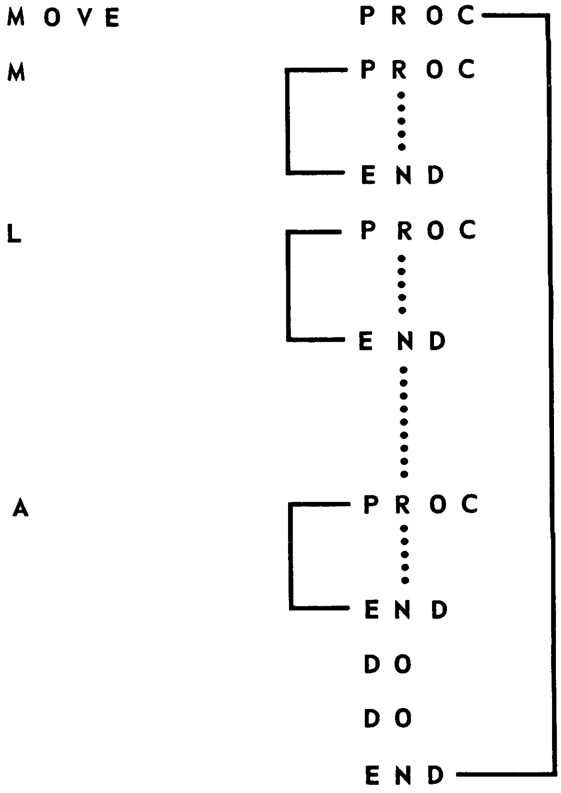


```
LABEL   Δ   OPERATION            Δ   OPERAND                Δ
1
            LA          15,IN+3
            SA          15,OUT+3
            LA          15,IN+7
            SA          15,OUT+7
            LA          15,IN+11
            SA          15,OUT+11
            LA          15,IN+15
            SA          15,OUT+15
            LA          15,IN+19
            SA          15,OUT+19
```

Notice that a procedure may contain labels. For example, the MOVE procedure uses the labels COUNT, TEMP and ADD. These labels belong to the procedure, the UTMOST assembler recognizes them as such, and the same labels can be used in a program referencing this procedure without fear of confusion.

The assembler makes a clear distinction between labels appearing in a procedure and references in that procedure to these labels as opposed to labels appearing in a program and references in this program to these latter labels.

The MOVE procedure consists of many nested PROCs which the MOVE PROC calls on. The only lines of coding in the MOVE PROC itself are the two DO lines before the last END line.



The first evaluation of a reference to the MOVE PROC occurs at the first DO line of the MOVE PROC.

9. TAPE FILE HANDLING

As its title indicates, this section is primarily concerned with programmer handling of magnetic tape input and output to and from the Processor. However, before discussing this subject, it is fruitful to spend some time on input and output from and to the console. This subject, in turn, depends on an understanding of "overflow" and "invalid operation", which will be discussed first in this section.

A. OVERFLOW

Addition or subtraction resulting in a carry beyond the capacity of the arithmetic register(s) specified in the addition or subtraction instruction results in a condition called *overflow*. In such a situation, the addition or subtraction is completed, and the correct sum or difference, less the carry causing the overflow, is stored in the arithmetic register(s) specified. It may be noted that when overflow occurs during addition or subtraction the carry that is lost is always a one.

Overflow can also occur during division. This will happen when the absolute value of the operand specified by m′ is less than or equal to the absolute value of the contents of arithmetic register 8. For example, the following will cause overflow:

1 LABEL Λ	OPERATION Λ	OPERAND Λ
DD	DIVISOR	

(AR8)i = 008762

(AR4)i = 900000

(DIVISOR)i = 008750

Notice that the absolute values concerned in the determination of whether overflow will occur are values as viewed by the Processor. The position of program decimal points has no bearing here.

If overflow occurs during the execution of an instruction, at the end of the instruction execution cycle (when the next instruction to be executed is normally accessed) there instead occurs something called *interrupt*. Interrupt is a hardware feature of the Processor. It causes the contents of the control counter (the address of the instruction that would normally be executed next) to be stored in a given storage location, and also causes a fixed address to be stored in the control counter. Thus, interrupt makes a record of where the program being executed was "interrupted", and forces control to go to the instruction stored in a given storage location. In the case of an overflow interrupt, the contents of the control counter are stored in storage location 18, and control goes to the instruction stored in location 19. An interrupt that performs in this way is called a *contingency interrupt*, to distinguish it from other interrupts (to be described later in this section) which cause control to go to the instruction stored in some other given storage location.

Storage locations 18 and 19 are in low order store, where the executive routine is stored. Thus, occurrence of interrupt causes control to be passed to the executive routine. Unless the programmer has planned on this occurrence (in which case he can notify the executive routine of this fact in a way to be described later in this section), the executive routine will type out on the console typewriter a message of the form:

CONTING.ΔmmmmmΔ00Δ00Δ0000b

where mmmmm is the address of the storage location in which is stored the instruction whose execution caused the overflow, b is an indicator.

In the case of overflow, the indicator is a 1.

B. INVALID OPERATION

Only certain binary configurations in the operator portion of an instruction word are recognized by the Processor as instructions. If the Processor accesses a word as an instruction, where the word has in the operator portion a binary configuration not recognized as an instruction, a condition called *invalid operation* occurs. For example, the following word would constitute an invalid operation if it were accessed as an instruction.

+ 0740000

An invalid operation causes a contingency interrupt. If the invalid operation is unplanned by the programmer, the executive routine will take the same action as it does with respect to unplanned overflow. In this case the indicator is a 2.

C. CONSOLE TYPEOUTS

A program can, via the executive routine, typeout messages on the console typewriter. This typewriter types characters at a speed of ten characters per second. It is capable of printing 51 different characters and has its format controllable by a carriage return, a tab, and a form feed. Printing is done on a sprocket-fed continuous form at a horizontal spacing of ten characters to the inch and a vertical spacing of six lines to the inch. The typewriter is capable of producing an original and at least five copies. The maximum number of characters that can be printed on one line is 72. The attempt to print more characters than the maximum on one line causes character pileup at the end of the line. The 51 printable characters and the format controls, together with their excess-three codes, are shown in Figure 9-1.

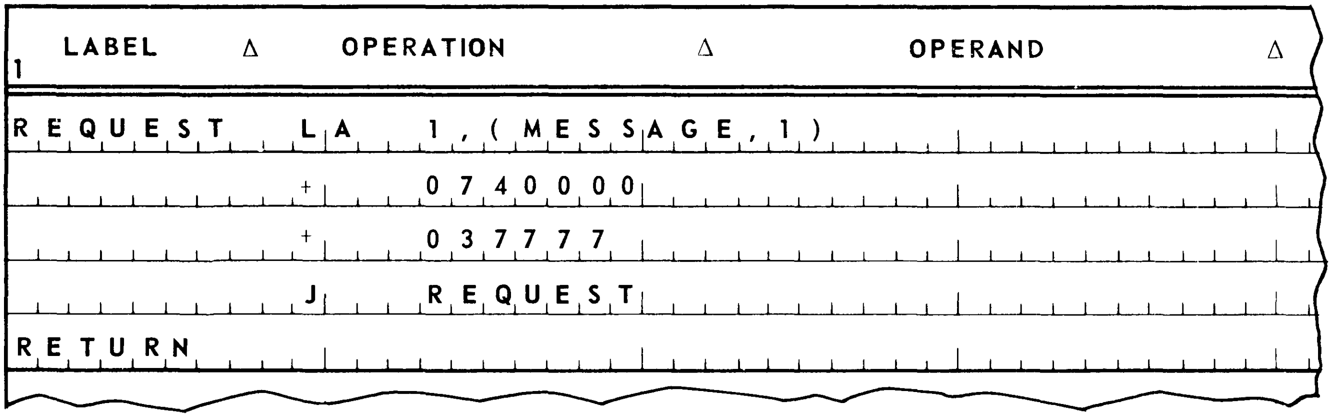
		ZONE			
		00	01	10	11
NUMERIC	0000	Δ	+	5	$
	0001	;	)	*	(
	0010	–	.	$	comma ,
	0011	0	carriage return	bell ring	' apostrophe
	0100	1	A	J	/
	0101	2	B	K	S
	0110	3	C	L	T
	0111	4	D	M	U
	1000	5	E	N	V
	1001	6	F	O	W
	1010	7	G	P	X
	1011	8	H	Q	Y
	1100	9	I	R	Z
	1101	:	=	2	:
	1110	<	–	tab	form feed
	1111	>	0	4	U

Figure 9-1. Console Typewriter Codes

To type a message on the console, the programmer places the message to be typed in a series of consecutive words. The first of these words should have a label. Following the last character of the message should be a carriage return character. This carriage return is recognized by the executive routine as marking the end of the message. It also is considered part of the message.

To effect a typeout of this message the programmer should load arithmetic register one with an indirect address control word with an l portion consisting of the label of the first word of the message and a specification of index register one in the index register portion. The programmer should then force an overflow or invalid operation. If the word immediately following the instruction that forces overflow or an invalid operation contains + 037777, the executive routine will recognize the resulting contingency interrupt as a request for a console typeout.

If after receiving this request, the executive routine determines that the typewriter is already busy, control is returned to the line following the line containing the + 037777 without any attempt to effect the typeout. This *busy return line* generally contains a jump back to reinitiate the typeout request. Thus, the program remains in a loop until the typewriter becomes free and the executive routine can initiate the typeout. At this point the executive routine returns control to the line following the busy return line. As a result, a request for a typeout generally takes the following form, where MESSAGE is the label of the first word of the message.



```
1 LABEL     Δ   OPERATION        Δ          OPERAND          Δ
REQUEST     LA  1,(MESSAGE,1)
            +   0740000
            +   037777
            J   REQUEST
RETURN
```

When the request has been serviced by the executive routine, control returns to the instruction in the line labelled RETURN.

The definition of some instructions that are useful in fabricating typewriter messages are as follows:

LOAD AR CONVERTING TO DECIMAL – LAD

The operand specified by m′ is always three words long. The information stored in these three words are presumed to be in six bit alphanumeric code. These three words are transferred to the arithmetic unit. During the transfer the zone bits of the characters are stripped, and the remaining numeric portions are compressed into two words. These two words are stored in the two arithmetic registers specified. The sign bits of these registers will contain the sign of the least significant word of the three word operand. No multiword operands other than that specified above are permissible. Field selection is not permissible. Indirect addressing may be used.

Example:

```
1  LABEL     Δ     OPERATION          Δ              OPERAND            Δ
       LAD  12, DATA
```

(AR8)i = + 012345	(AR8)f = + 123412
(AR4)i = + 678901	(AR4)f = + 345656
(DATA-2)i = – 1234	(DATA-2)f = – 1234
(DATA-1)i = – ABCD	(DATA-1)f = – ABCD
(DATA)i = + 56EF	(DATA)i = + 56EF

STORE AR CONVERTING TO ALPHANUMERIC – SAA

The arithmetic register operand is always two words long, the operand specified by m′ three words long. The two words constituting the arithmetic register operand are presumed to be in four bit numeric code. These two words are transferred to the store. During the transfer the words are expanded from two to three by having 00 zone portions inserted before the numeric portion of each decimal number. The sign bits of the resulting three words will contain the sign of the least significant word of the arithmetic register operand. No multiword operands other than that specified are permissible. Field selection is not permissible. Indirect addressing may be used.

Example:

```
1  LABEL     Δ     OPERATION          Δ              OPERAND            Δ
       SAA  12, DATA
```

(AR8)i = – 123412	(AR8)f = – 123412
(AR4)i = + 345656	(AR4)f = + 345656
(DATA-2)i = – 123456	(DATA-2)f = + 1234
(DATA-1)i = + ABCD	(DATA-1)f = + 1234
(DATA)i = – WXYZ	(DATA)f = + 5656

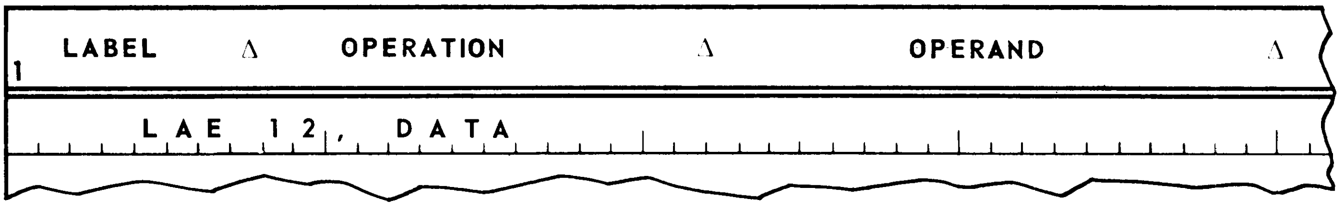
LOAD AR EDITED – LAE

Load the operand specified by m′ into the arithmetic register(s) specified. The operand from the store is presumed to be in six bit alphanumeric code. As the operand is transferred, it is scanned from most to least significant character. As long as only the following characters:

CHARACTER	CODE
Space (Δ)	00 0000
Semicolon (;)	00 0001
Dash (-)	00 0010
Zero (0)	00 0011
Comma (,)	11 0010

are encountered in the scan, each character is replaced by a space (Δ). As soon as some character other than the ones listed above is encountered in the scan, this operation of *zero suppression* ceases, and the rest of the operand is transferred to the arithmetic register(s) unaltered. Signs are not affected in this transfer. Multiword operands may be used with this instruction. However, unlike the addressing of multiword operands with all other instructions, the LAE instruction addresses multiword operands by specifying in m′ the address of the most significant word of the multiword operand. Field selection is not permissable with this instruction. Indirect addressing may be used.

Example:


1
LABEL Δ OPERATION Δ OPERAND Δ

```
LAE 12, DATA
```

(AR8)i	=	– 012345	(AR8)f f=	+	ΔΔΔΔ
(AR4)i	=	– 678901	(AR4)f =	+	ΔΔ12
(DATA-1)i	=	+ ABCD	(DATA-1)f =	+	ABCD
(DATA)i	=	+ Δ;–0	(DATA)f =	+	Δ;–0
(DATA+1)i	=	+ ,012	(DATA+1)f =	+	,012

1. Example

   Given five receipt amounts stored in consecutive storage locations in the form:

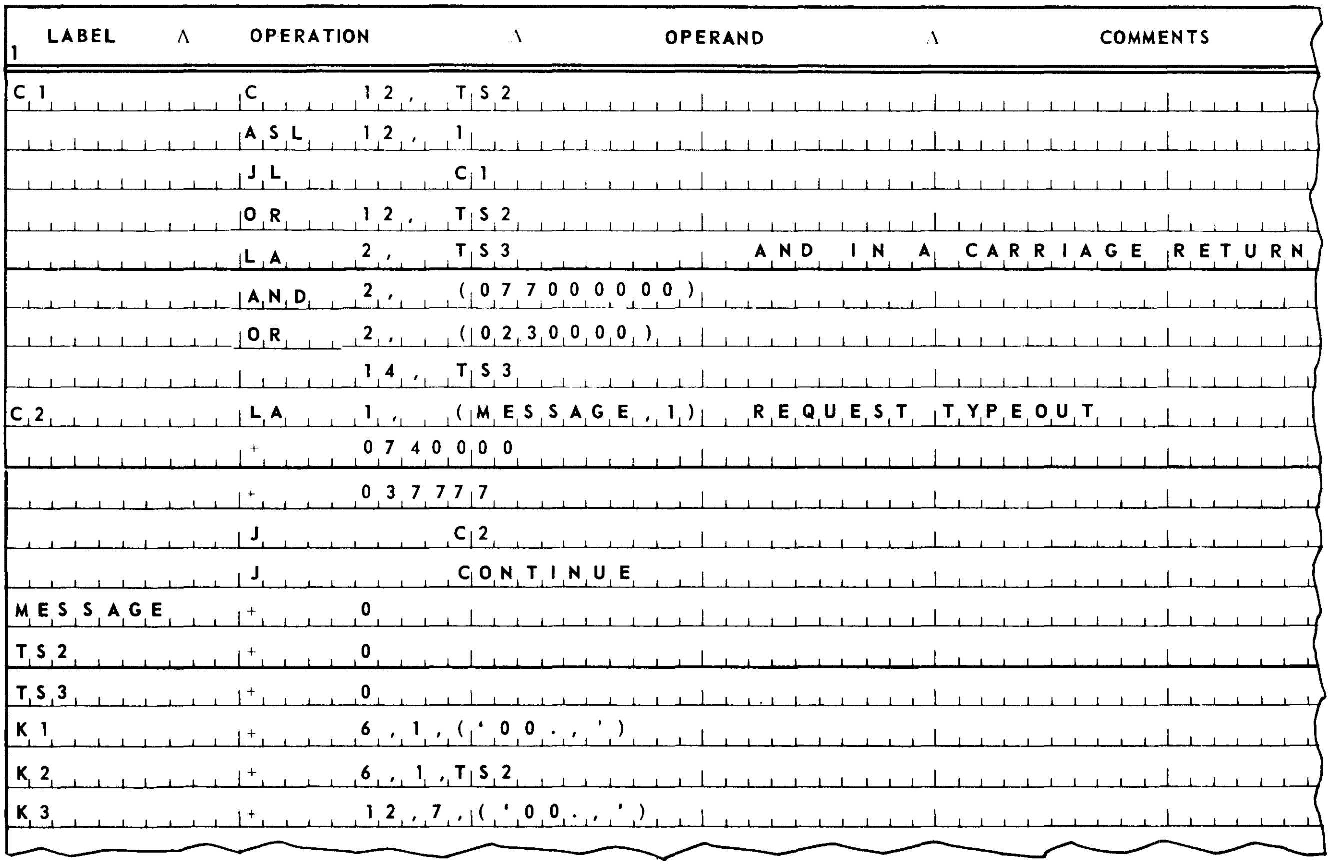
   + 0XXX‸XX

   type on the console typewriter the sum of the receipt amounts with a dollar sign preceding the most significant dollar digit, a comma between the thousands dollar digit and the hundreds dollar digit if the sum is $1,000 or more, and a decimal point between the least significant dollar digit and the most significant cents digit. Type both cents digits regardless of their value. If there are no dollar digits, type the dollar sign immediately before the decimal point. In any case, type enough spaces before the dollar sign so nine characters are always typed. The sum of the receipt amounts will never be more than 9999‸99. The address of the first receipt amount is stored in label FILEA. Assume that a USE directive has already occurred in your program. When finished, jump to the label CONTINUE. Label the first line of your coding TYPEOUT.

2. Coding

```
1 LABEL     OPERATION            OPERAND                  COMMENTS
TYPEOUT    LX    15, FILEA               SUM AMOUNTS
           LA    4,  0,15
           DA    4,  1,15
           DA    4,  2,15
           DA    4,  3,15
           DA    4,  4,15                       DDDDCC
           LA    8,  (:000000)             000000 DDDDCC
           DSL   12, 3                     000DDD DCC000
           SAA   12, TS3                 000D DDDC C000
           LA    14, TS3
           ASL   12, 1                   00DX DDCΔ C000
           LA    4,  TS2                 00DX DDDC C000
           LF    8,  *K1                 00D, DDDC C000
           ASL   12, 1                   0D,D DDXΔ C000
           LF    4,  *K2                 0D,D DDXC C000
           LF    4,  *K3                 0D,D DD.C C000
           SA    14, TS3                 ZERO SUPPRESS
           LAE   12, MESSAGE
           SA    12, TS2                 FLOAT DOLLAR SIGN
           LA    12, (TWCΔ'            $S')
```

(Coding Continued)



```
LABEL      OPERATION           OPERAND            COMMENTS
C1         C        12,  TS2
           ASL      12,  1
           JL            C1
           OR       12,  TS2
           LA       2,   TS3          AND IN A CARRIAGE RETURN
           AND      2,   (077000000)
           OR       2,   (0230000)
                    14,  TS3
C2         LA       1,   (MESSAGE,1)  REQUEST TYPEOUT
           +        0740000
           +        037777
           J             C2
           J             CONTINUE
MESSAGE    +        0
TS2        +        0
TS3        +        0
K1         +        6,1,('00.,')
K2         +        6,1,TS2
K3         +        12,7,('00.,')
```

3. Student Exercise

Given the following:

WORD	DATA
0	+ AAAA▲AA
1	+ BBBB▲BB
2	+ CCCC▲CC

where

A is an amount

B is another amount

C is a third amount

type on the console typewriter the smallest amount in the form:

$SSSS.SS

where S is the smallest amount. The address of the first amount is stored in label FILEA. Assume that a USE directive has already occurred in your program. When finished, jump to the label CONTINUE. Label the first line of your coding TYPEOUT.

4. Test

Given the following:

WORD	DATA
0	00HHHH▲
1	00NNNN▲
2	00RRRR▲

where

H is the onhand quantity

N is the onorder quantity

R is the required quantity for the next 60 days.

Type on the console typewriter the quantity, H + N – R, which will be positive and four digits at a maximum. The address of the onhand quantity is stored in label FILEI. Assume that a USE directive has already occured in your program. When finished, jump to the label CONTINUE. Label the first line of your coding TYPEOUT.

D. CONSOLE TYPEINS

The console contains a keyboard as shown in Figure 9-2. From this keyboard messages can be typed into the store. Typeins are accepted from the console by the executive routine.

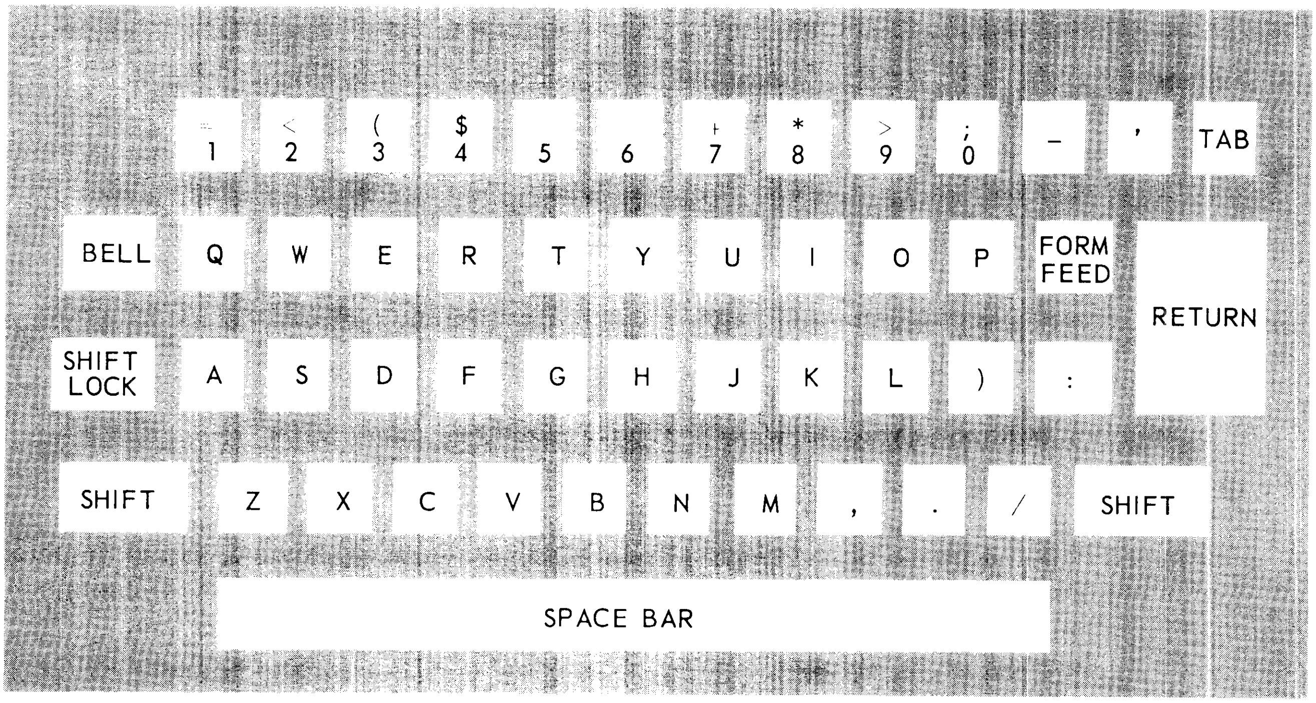


Figure 9-2. Console Keyboard

The executive routine then routes the message to the proper program by means of a *flag* character typed in conjunction with the message.

All typeins to programs begin with the character "R". (An exception will be discussed in Section 13 Symbionts). The next character typed in is the flag character. A space is typed next. Then the message is typed.

The program must tell the executive routine what character it wishes to use as its flag. This is done by storing in the label TABB a word with the desired flag character in bits 19 through 24. This operation must be done before the program expects any typeins. Characteristically, if a program expects typeins, one of the first operations to be done in the program is to store its flag in TABB.

At the same time that the flag is delivered to the executive routine, the address of a word in the program is also delivered. This is the address of the typein word. The typein address is delivered in bits one through 15 of the word stored in TABB. The typein word is used by the program to tell the executive routine when it is ready to accept a typein. This is done by storing binary zeros in the typein word. If the typein word contains anything other than binary zeros, the executive routine will not deliver a typed in message to the program even if a message flagged for the program has been received.

If the typein word contains binary zeros, when the executive routine receives a message flagged for the program, it will do an SLJ to the typein address. Consequently, following the typein word of a program should be a short *acceptance routine* designed to accept the typein. When the acceptance routine receives control, index register one will contain the typein address and index register two will contain the address of the zero word of the area in which the executive routine has stored the typed in message. No other index registers will be properly loaded. Consequently, the cover index register for the acceptance routine will be index register one, all index register specifications in the instructions constituting the acceptance routine must be explicit, and only expressions involving binary numbers and *reflexive addressing* (use of the dollar sign element) can appear in the operands of these instructions. As a consequence, the acceptance routine should be designed to do little more than transfer the message from the executive routine typein area to an area in the program and set a connector in the program to indicate that the typein has been received. Control should then be returned to the executive routine by executing a jump with indirect addressing to the typein address. It should be noted that, in doing an SLJ to the typein address, the executive routine changes the contents of the typein word to something other than binary zeros.

When the executive routine transfers control to the acceptance routine, the zero word of the executive routine's typein area will contain ΔRfΔ, where f is the program's flag. Consequently, the message the program is looking for actually begins in word one of the executive routine's typein area. Moreover, before accepting a typein from the console into the typein area, the executive routine will clear the typein area to all spaces. Consequently, any character positions following the typein in the typein area will contain space symbols. If proper preparation for receiving a message has not been programmed, the message will be lost i.e., no indication of the presence of message is transmitted to the program.

1. Example

   A program is designed to execute one of two branches dependent on a typein. If FORCE is typed in, the program is to jump to the label FORCED. If RECHECK is typed in, the program is to jump to the label RECHECK. The typein is requested by typing out TYPEIN.

2. Coding

```
LABEL          OPERATION      OPERAND
               USE            3
TABBWORD       FORM           1,6,3,15
START          LA             8,(TABBWORD 0,'W',0,TYPEIN)
               SA             8,TABB
C1             LA             1,   (MESSAGEOUT,1)
               +              0740000
               +              037777
               J                   C1
               SZ                  TYPEIN
C2             IXC            0,   TYPEIN
               JE                  C2
               LA             12,  MESSAGEIN
               C              12,  ('FORCE')
               JE                  FORCED
               C              12,  ('RECHECK')
               JE                  RECHECK
               J                   C1
MESSAGEOUT     +              'TYPE'
               +              'IN+0  '
TYPEIN         +              'BUSY'
               LA             12,  2,2
               SA             12,  5,1
               J                   *0,1
               +              0
MESSAGEIN      +              0
```

E. THE UNISERVO IIIA TAPE UNIT

Magnetic tape is the means of introducing and removing large volumes of data to and from the Processor's store. The tape handler designed to handle the reading and writing of magnetic tape is the UNISERVO IIIA tape unit. The tape used on these units has a MYLAR * base, is one mil thick, is 0.5 inch wide, and has a maximum length of 3600 feet.

Information can be recorded on 3500 of this 3600 feet. The information is recorded on the tape in *frames*. A frame is a recording of bits across the width of the tape. Nine bits of information can be recorded in one frame. Thus, one word of information is recorded in three frames on tape. A little over 1000 frames of information can be recorded on one inch of tape.

Information is recorded on tape in units called *blocks*. A block is the unit of information that the Processor reads from or writes on tape in one operation. A block consists of some number of words. The number is variable at the programmer's option. Since it may be necessary to stop tape movement between the reading or writing of blocks, and since the tape cannot be stopped instantaneously, blocks are separated on tape by *interblock gaps*, lengths of tape on which no data is recorded. The length of this interblock gap is variable and depends on whether the tape must be stopped between the writing of one block and the writing of the next. If stopping is required, the interblock gap is about 0.6 inch long. If not required, about 0.4 inch long.

The format in which a block of data is recorded on tape is shown in Figure 9-3. Although both are completely variable at the programmer's option; the size in number of words of the items recorded on one tape and the size in number of items of the blocks recorded on the tape are generally fixed. Under such circumstances, the approximate number of blocks that can be recorded on 3500 feet of tape is given by the following formula:

$$N = \frac{14 \times 10^6}{n(w + 1) + 333G + 3}$$

where

N is the number of blocks

n is the number of items per block

w is the number of words per item

G is the interblock gap length in inches

For example, a 3500 foot tape blocked at 20 items per block, each item consisting of 25 words, can hold approximately 20,300 blocks. (For purposes of this calculation an interblock gap length of 0.5 inch was assumed.)

When being read or written, tape is moved at a speed of 100 inches per second. Time to pass over the interblock gap varies depending on interblock gap length and on whether the tape is stopped between blocks. Minimum interblock gap time is 4 milliseconds, maximum 8 milliseconds. Given a constant item size, the approximate time to read or write a block is given by the following formula:

$$T = 0.03 \, [\, n\,(w + 1) + 3 \,] + g$$

* *Registered trademark of the E. I. du Pont de Nemours & Company, Inc. Wilmington, Delaware.*

where

T is block time in milliseconds

n is the number of items in the block

w is the number of words per item

g is interblock gap time in milliseconds

For example, a 20 item block, each item consisting of 25 words, can be read or written in approximately 21.7 milliseconds. (For purposes of this calculation an interblock gap time of 6 milliseconds was assumed.)

The end of a tape is marked bv an *end-of-tape warning window*. Following the end-of-tape warning window there is about 25 feet of tape on which information can be recorded.

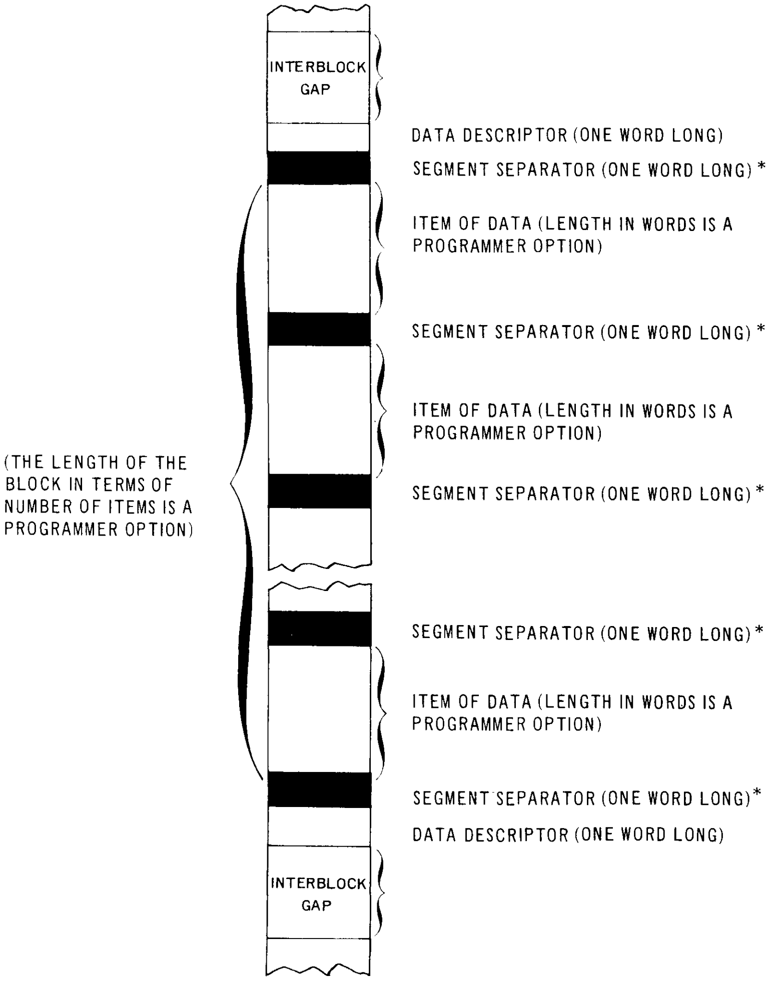

INTERBLOCK GAP
DATA DESCRIPTOR (ONE WORD LONG)
SEGMENT SEPARATOR (ONE WORD LONG) *
ITEM OF DATA (LENGTH IN WORDS IS A PROGRAMMER OPTION)
SEGMENT SEPARATOR (ONE WORD LONG) *
ITEM OF DATA (LENGTH IN WORDS IS A PROGRAMMER OPTION)
SEGMENT SEPARATOR (ONE WORD LONG) *
(THE LENGTH OF THE BLOCK IN TERMS OF NUMBER OF ITEMS IS A PROGRAMMER OPTION)
SEGMENT SEPARATOR (ONE WORD LONG) *
ITEM OF DATA (LENGTH IN WORDS IS A PROGRAMMER OPTION)
SEGMENT SEPARATOR (ONE WORD LONG)*
DATA DESCRIPTOR (ONE WORD LONG)
INTERBLOCK GAP

Figure 9-3. Block Recording

** The one word segment separator is created by the tape synchronizers for tape control; it is not moved to or from store.*

A schematic of the UNISERVO IIIA Tape Unit is shown in Figure 9-4. Tape is said to be moving forward when the tape is traveling from the lefthand supply reel to the righthand tape reel, backward when the tape is traveling from the righthand reel to the lefthand reel. The righthand reel is permanent. Consequently, a reel of tape to be read from or written on is mounted on the lefthand hub. The tape is connected to a prethreaded leader fastened to the righthand reel. Because of the prethreaded leader, removal of a reel and the mounting of a new reel takes only about 15 seconds. The reel is removed by pressing the center of the hub.

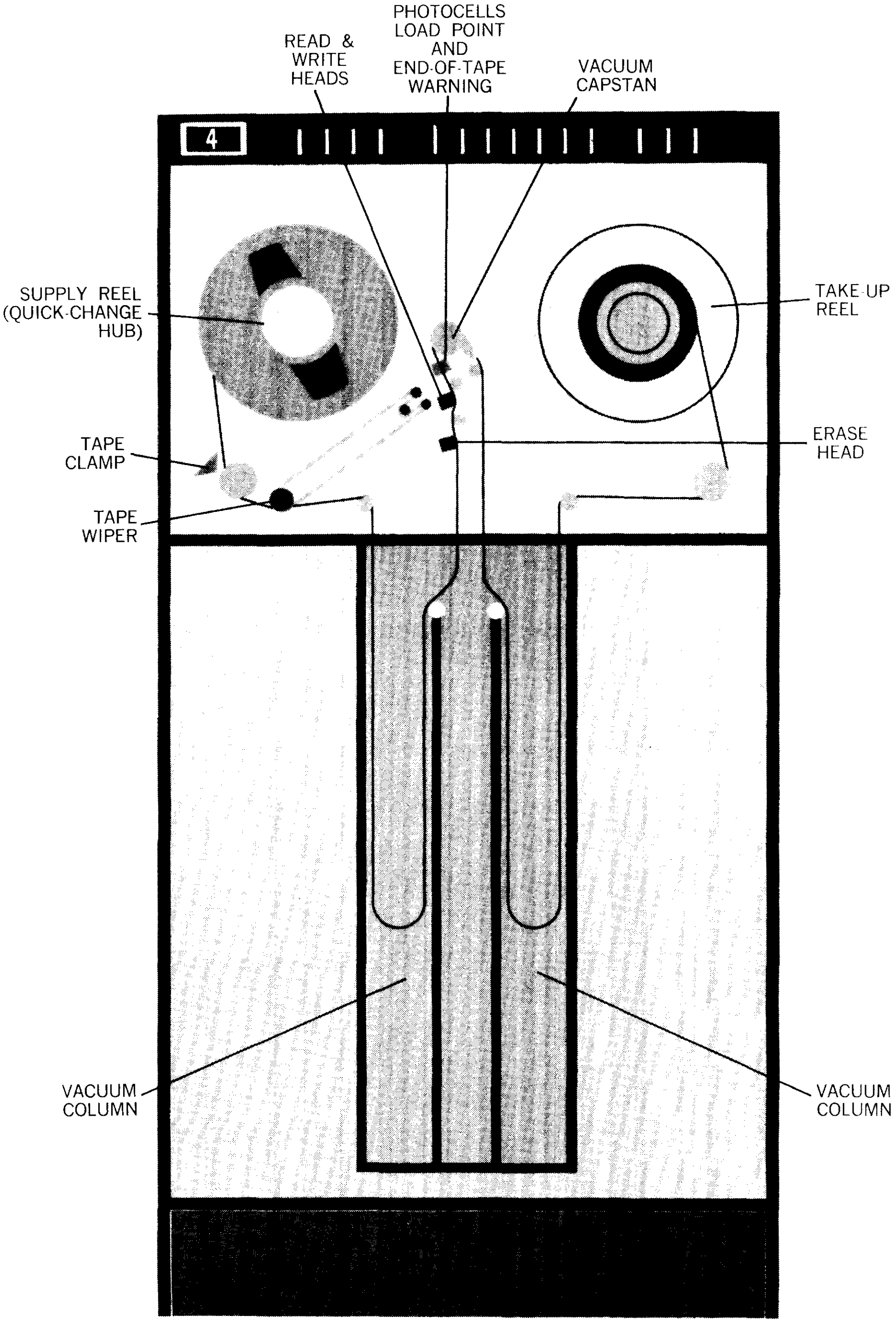


Figure 9-4. Tape Path

If a reel of tape is read or written in a forward direction and is then to be dismounted, it must first be *rewound* onto the left hand reel. A tape of maximum length is rewound in 125 seconds.

Tape may be written forward, read forward, or read backward. When a block is read backward, the words in each item in the block are stored in the store in the same order as they are stored when the block is read forward.

As many as 16 UNISERVO IIIA Tape Units may be attached to the Processor by a *UNISERVO IIIA Synchronizer.* The Synchronizer has two channels, one for reading and one for reading or writing. Thus, the Processor can be reading information from one UNISERVO IIIA Tape Unit at the same time it is writing information on another. In addition, transfer of information between the store and the UNISERVO IIIA Tape unit is *buffered* by the Synchronizer. This allows the reading and writing of tapes to occur simultaneously with the use of the store by the Processor to execute instructions.

Any number of UNISERVO IIIA Tape Units may be rewound simultaneously. Once rewind on a UNISERVO IIIA Tape Unit is initiated, the Synchronizer is free to control reading, writing, and rewinds on other UNISERVO IIIA Tape Units. The UNISERVO IIIA Tape units are identified by number, the numbers being 0 through 15.

F. TAPE HANDLING

The data making up a tape file may be recorded on one or more tapes. If recorded on one tape, the file is known as a *single reel file*. If recorded on many, as a *multireel file*. Following the last block of data recorded on the last reel of a tape file are recorded two *end-of-file sentinel blocks*. Following the last block of data recorded on an intermediary tape in a multireel file are recorded two *end-of-reel sentinel blocks*. Both end-of-reel and end-of-file sentinel blocks are one word blocks, and both are negative words. Bit positions 25 and 24 of an end-of-reel sentinel block contain 10. Of an end-of-file sentinel block, 11.

The first block of data on a tape may be preceded by a *label block*. A label block has the following format.

WORD	CONTENTS
0	A negative binary zero
1	The first four characters of the file identification
2	The date of cycle in decimal format
3	The reel count in decimal format
4	Free
5	Free
6	Free
7	Free
8	Free
9	Free
10	The last four characters of the file identification
11	A negative binary zero

The file identification, date of cycle, and reel count make up an identification of the data that follows on the tape. The file identification is usually arbitrary and is assigned by the user to distinguish between files. For example, the file identification of the master employee file would be different from the file identification of the master inventory file. The date of cycle is generally the date on which the data was written on the tape. The reel count is 000001 for the first reel in a file, 000002 for the second reel, 000003 for the third, and so on. Words four through nine of the label block are available for whatever use the user wishes to make of them.

Processing of input data to produce output data is a programmer responsibility. Reading of data from tape into the store and writing of data from memory onto tape can be done by means of standard input/output routines which, for the time being, can be considered part of the executive routine. (Where these input/output routines are actually located will be described later in this manual.)

Although input/output handling is not a programmer responsibility, the user must allocate those areas which will contain the data to be read or written. This function is accomplished by means of the RES directive. Since information must be read and written a block at a time, sufficient area to contain one block must be reserved for each input and output area.

To achieve the advantages of simultaneous reading, writing and processing, sufficient area to contain at least two blocks should be reserved for each input and output area. To minimize the possibility of *read interlock,* area for storage of more than two blocks can be reserved for an input area if such space is available. (Read interlock occurs when the program is ready to process another input item and there are no more input items in the store to be delivered for processing. In such a situation, the Processor must wait in an ''interlocked'' state until the read of the next block of input items is completed.) Because of the structure of input tapes (namely, that there are only two sentinel blocks at the end of a tape), the maximum area that can be usefully reserved for an input area is area to hold one less than the number of items in three blocks of input.

To minimize the possibility of *write interlock,* area for storage of more than two blocks can be reserved for an output area if such space is available. (Write interlock occurs when the program is ready to deliver another item for output and all the output area is already full of items waiting to be written.) There is no maximum limit to the size of the area reserved for an output area. The zero word of the area reserved for an input or output area should be labeled.

The input and output routines partition off the input and output areas reserved by the programmer. As part of the partitioning, the routines use two words per item to store control information. Thus, each item area reserved for input or output storage must be two words longer than the size of the item to be stored. Thus, if 20 item blocks are to be read, each item consisting of 25 words, and sufficient input area is to be reserved to store two blocks of data, then 1080 words should be reserved.

In addition to reserving input and output areas, the programmer must supply to the input/output routines a three word *control packet* for each area reserved. This packet has the following form:

```
+    0
+    0
+    x
```

where x is the number of words in an item. The first word in this packet should be labelled. Manipulation of information in this packet is a function of the input/output routines. The programmer's sole responsibility is to provide the packet(s) for these routines.

Finally, for each file to be read or written, the programmer must supply a *file description table.* This table is either 16 or 28 words long and is used by the input/output routines in reading and writing files. The programmer must supply certain information in this table for the routines. The format of the table is as follows.

Word 0:

Initially binary zeros can be stored here. This word should be labelled.

Word One:

The number of words in the item of the file should be entered here in binary. Maximum item size is 511 words.

Word Two:

The number of items per block of the file should be entered here in binary.

Word Three:

The label assigned to the zero word of the control packet associated with the input/output area to be used by the file should be entered here.

Words Four-Seven:

Initially binary zeros can be stored here.

Word Eight:

If this is an output file, binary zeros can be stored here. If an input file, the number of additional item areas, over and above the one initial block, which provide for advance reading of this file. This entry may be anything from zero through one less than the number of items in two blocks.

Word Nine:

If this is an input file and the program is to be notified everytime an input reel is exhausted, a binary one should be entered in bit position three of this word. If this is an output file and the program is to be notified each time an end-of-tape warning window is detected on an output reel, a binary one should be entered in bit position four of this word. All other bit positions of this word must have binary zeros in them.

Word Ten:

Starting at storage location 0200 (octal), the executive routine maintains a *tape assignment table*. Each word in this table refers to a UNISERVO IIIA tape unit. For the time being, it can be assumed that the word in storage location 0200 refers to the UNISERVO IIIA tape unit numbered 0, that the word in location 0201 refers to the UNISERVO IIIA tape unit numbered 1, that the word in 0202 refers to the UNISERVO IIIA tape unit numbered 2, and so on. (This is one possible arrangement of the tape assignment table. When the table is so arranged, it is referred to as being *canonized*.) The entry in word ten of the file description table should be the address of the entry in the tape assignment table to be associated with this file. Thus, if the tape assignment table is canonized, and if the first reel of the file is to be found on the UNISERVO IIIA tape unit numbered 1, the entry in word ten of the file description table should be an octal 0201.

Word Eleven:

Initially binary zeros can be stored here.

Word Twelve:

The first four characters of the file identification to be found in the label block of the file should be stored here. If the file has no label block, this entry is immaterial.

Word Thirteen:

The last four characters of the file identification to be found in the label block should be stored here. If the file has no label block, this entry is immaterial.

Word Fourteen:

Initially binary zeros can be stored here. When the input/output routines begin the reading of an input reel or the writing of an output reel, they will store here the reel number of the reel being read or written. Note that this is not necessarily the reel number to be found in the label block on the reel of tape. This is an internal reel count. For example, when an input routine begins the reading of the first reel of a file, it will store 000001 here. When it moves on to the next reel, it will store 000002 here. When it moves to the third, it will store 000003. And so on.

Word Fifteen:

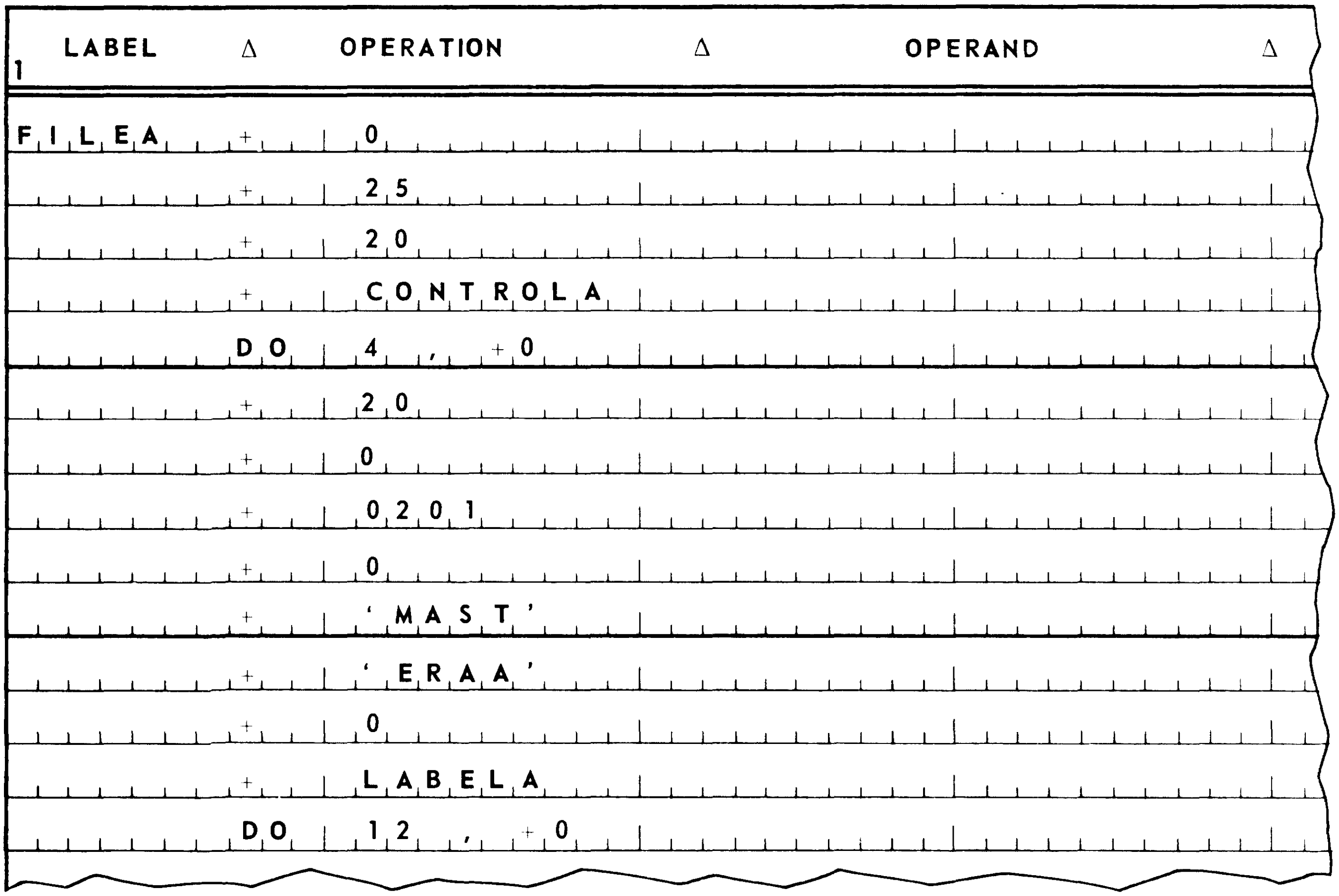
If binary zeros are entered here for an input file, the input routine assumes the file has no label block. If the file has a label block, the label of a closed *label check subroutine* should be entered here. When opening the reel of the file, the input routine will read the label block and do an SLJ to the label check routine. The label check routine then has the opportunity to determine if the reel read has the proper label. On completion of the check, the label check routine returns control to the input routine by means of a jump with indirect addressing to the labelled line.

If binary zeros are entered here for an output file, the output routine assumes the file is to have no label blocks. If the file is to have label blocks, the label of a *closed label creation subroutine* should be entered here. When opening the reel of a file, the output routine will do an SLJ to the label creation routine. The label creation routine then has the opportunity to create a label for the reel. (Note: Word Fourteen may be used by the label creation subroutine.) After creation, the label check routine returns control to the output routine by means of a jump with indirect addressing to the labelled line, at which point the output routine will write the label created.

Words Sixteen - Twenty Seven:

For an input file with label blocks, the label block will be read into these words. For an output file with label blocks, the label block will be written from these words. Otherwise, these words may be omitted from the file description table. Initially binary zeros can be stored here.

For example, the file description table of an input file mounted on UNISERVO tape unit 1, recorded at 20 items per block, an item consisting of 25 words; with labels, an expected file identification of MASTERAA, a control packet labelled CONTROLA, a label check routine labelled LABELA, and an input area capable of storing 40 items might appear as follows:



```
1 LABEL    Δ  OPERATION  Δ  OPERAND  Δ
FILEA         +             0
              +             25
              +             20
              +             CONTROLA
              DO            4 , +0
              +             20
              +             0
              +             0201
              +             0
              +             'MAST'
              +             'ERAA'
              +             0
              +             LABELA
              DO            12 , +0
```

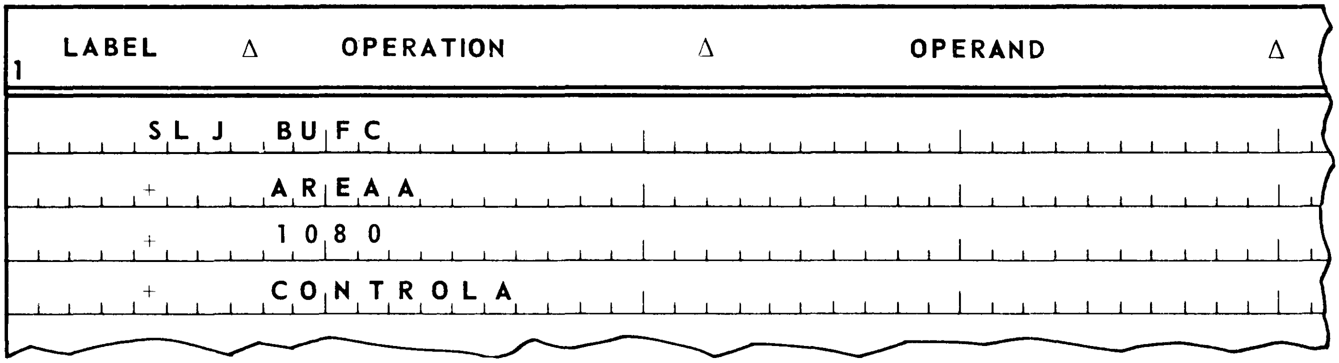
Before any files may be read or written, the input and output areas must be partitioned and the proper control set up. This may be done by executing an SLJ BUFC for each input and output area reserved. The SLJ BUFC must be followed by three constants written in the order listed.

1. The label of the zero word of the area reserved.

2. The number of words in the area reserved entered in binary.

3. The label of the zero word of the control packet associated with the area reserved.

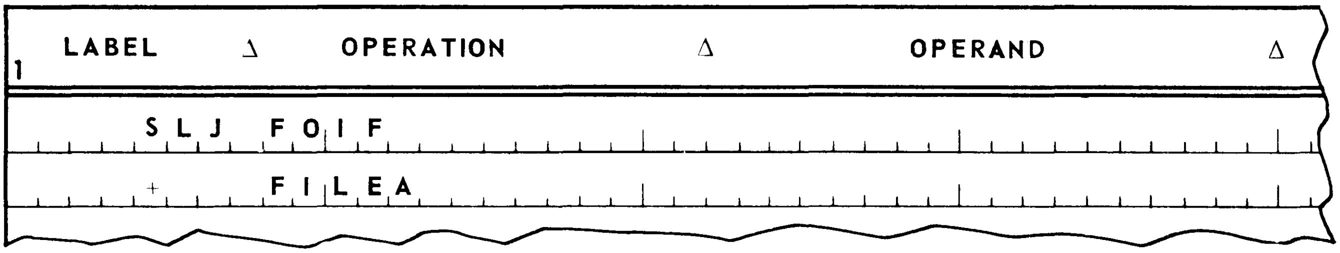
For example, to execute BUFfer Control for a file whose reserved area is labelled AREAA, in which 1080 words have been reserved, and whose control packet is labelled CONTROLA, the following coding would be used:



```
LABEL   Δ   OPERATION   Δ   OPERAND   Δ
     SLJ  BUFC
      +   AREAA
      +   1080
      +   CONTROLA
```

After buffer control has been executed, control will be returned unconditionally to the line immediately following the third constant.

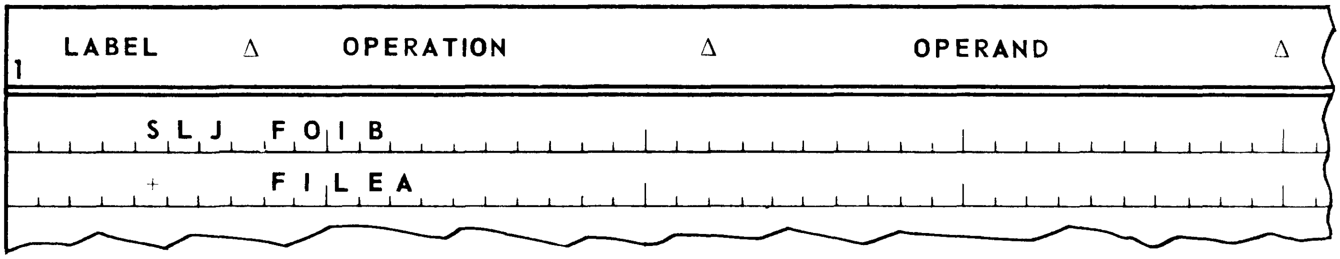
An input file may be opened by executing an SLJ FOIF. This instruction must be followed by a constant that contains the label of the zero word of the file description table of the file to be opened. Execution of this instruction will cause the specified file description table to be initialized for reading forward. If the file is labelled, the label block will be read, and the label check routine executed. If the sign of the constant is minus, the tape will be rewound before any reading occurs. If such a rewind is not desired, the sign of the constant should be plus, For example, to open forward an input file (**F**ile **O**pen **I**nput **F**orward) with a file description table labelled FILEA, where the tape is not to be rewound before reading occurs, the following coding would be used:



```
LABEL   Δ   OPERATION   Δ   OPERAND   Δ
     SLJ  FOIF
      +   FILEA
```

After open of the input file forward has been executed, control will be returned unconditionally to the line immediately following the constant.

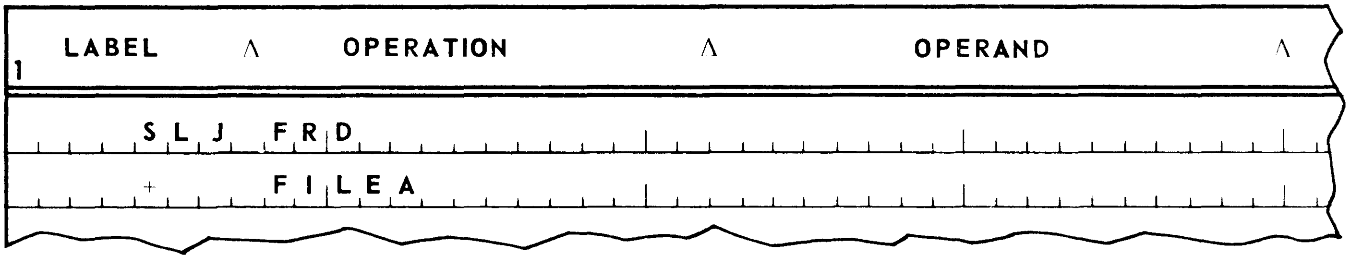
If a tape has been read or written forward and it is now desired to read the tape backward, the file on this tape must be closed and then the file description table for the file must be initialized for backward reading. This may be done by executing an SLJ FOIB. This instruction must be followed by a constant that contains the label of the file description table of the file to be opened backward. For example, to open backward an input file (**F**ile **O**pen **I**nput **B**ackward) with a file description table labelled FILEA, the following coding would be used:



```
LABEL      OPERATION                OPERAND
     SLJ FOIB
     +   FILEA
```

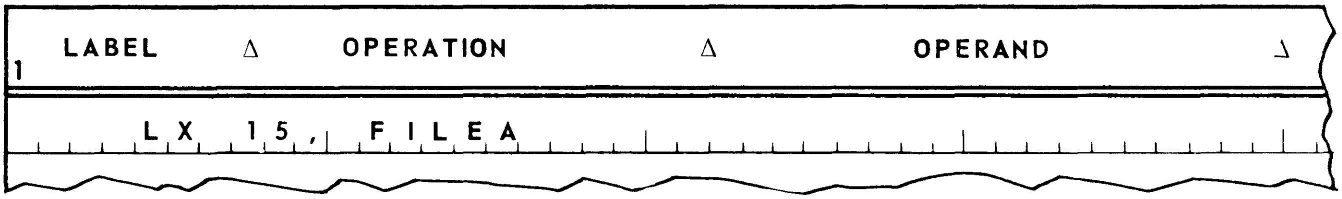
After open of the input file backward has been executed, control will be returned unconditionally to the line immediately following the constant. Execution of this initialization subroutine does not cause any tape movement. If the read head of the tape unit is positioned behind the sentinel blocks, these blocks will be passed over when backward reading begins. Only data blocks will be delivered for processing. It should be noted that backward reading is not a buffered operation. Thus, backward reading and processing cannot occur simultaneously. (This is a function of the backward read routine, not the computer hardware.)

After an input file has been opened, either forward or backward, an item to be processed can be acquired by executing an SLJ FRD. This instruction must be followed by a constant that contains the label of the file description table of the file. For example, to acquire an item to be processed (File ReaD) from a file with a file description table labelled FILEA, the following coding would be used:



```
LABEL      OPERATION                OPERAND
     SLJ FRD
     +   FILEA
```

After the file read has been executed, control is normally returned unconditionally to the second line following the constant. (This line is called the *normal return line.*) At this point the program can find the address of the zero word of the next item to be processed in the zero word of the file description table. Thus, the *base address* of the item could be loaded into index register 15 with the following instruction:



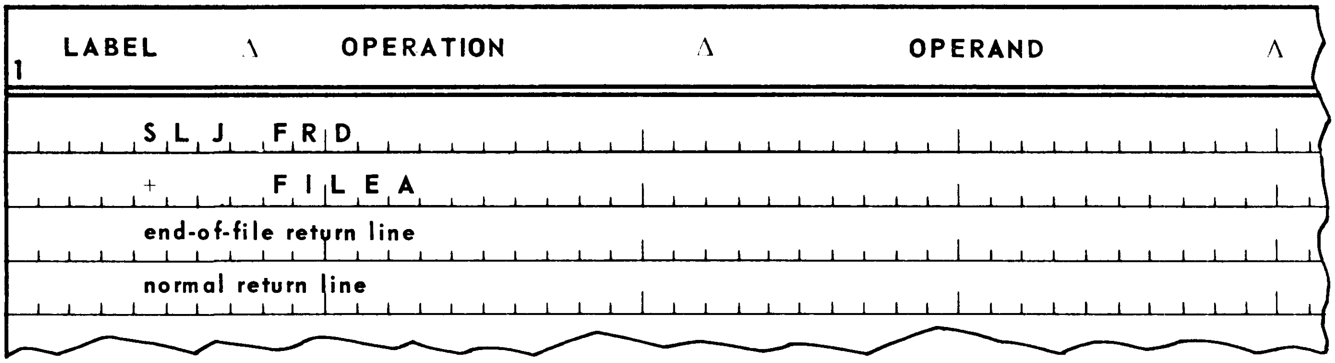
```
LABEL      OPERATION                OPERAND
     LX  15, FILEA
```

The item could then be addressed by using index register 15 as its cover index register. The zero word of the item could also be addressed indirectly by using the contents of FILEA as an indirect address control word.

Regardless of whether the file is being read forward or backward, items will always be delivered in the same format. That is, an item will be arranged in sequential storage locations, the zero word of the item being in the first storage location, the one word of the item being in the second location, the two word of the item being in the third, and so on.

If a file is being read forward, the first SLJ FRD will deliver the address of the first item on the tape, the second SLJ FRD will deliver the address of the second item, the third SLJ FRD will deliver the address of the third, and so on. If a file is being read backward, the first SLJ FRD will deliver the address of the last item on the tape (the last item being defined here as the last item in the block immediately to the left of the read head of the tape unit when the file was opened backward), the second SLJ FRD will deliver the address of the next to last item, the third will deliver the address of the second from the last item, and so on.

If when control is transferred to the input routine via an SLJ FRD to read a file forward the input routine discovers that there are no more items on the file, then instead of returning control to the normal return line, the input routine will return control unconditionally to the line immediately following the constant. This line is called the *end-of-file return line*. Thus, a schematic of the coding for an SLJ FRD would appear as follows:



```
1  LABEL      Λ   OPERATION          Λ          OPERAND          Λ
        SLJ   FRD
        +        FILEA
        end-of-file return line
        normal return line
```

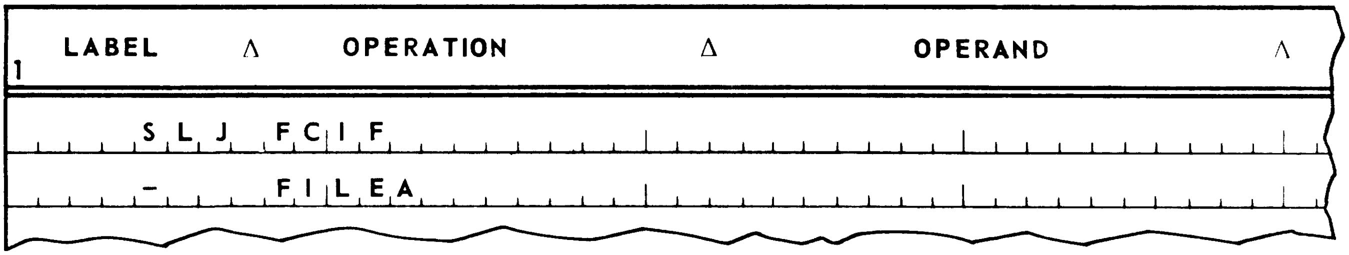
The input routine will return control to the end-of-file return line on detection of an end-of-file sentinel block at the end of a tape in the file.

If a tape is being read backward, no end-of-file notice will be given by the input routine. Consequently, if a tape is to be read backward at some point in a data processing system,care should be taken so that, when this tape is initially written, some type of unique item is written at the beginning of the tape so the program can recognize the beginning of the tape when it is reached. If the tape is labelled, the label block can be used for this purpose. Notice that a multireel file may be read forward, but only a single reel file may be read backward.

If a one has been placed in bit position three of word nine of the file description table, then the input routine will return control to the end of file return line, not only when an end-of-file sentinel block is detected, but also whenever an end-of-reel sentinel block is detected. Thus, this return line is more appropriately called the *end-of-file (reel) return line*. At the time control is returned to the end-of-file (reel) return line, the sentinel can be checked to determine whether this is an end-of-reel or an end-of-file return. It is accessed as follows:

```
LX n, FILE + 4 (5th word of file description table)
LA 1, 1, n
```

When a programmer desires to close an input file, this may be done by executing an SLJ FCIF. This instruction must be followed by a constant that contains the label of the file description table. If the sign of the constant is minus, the tape will be rewound. If plus, the tape will not be rewound. For example, to close an input file (File Close Input File) with a file description table labelled FILEA where the tape is to be rewound, the following coding would be used:

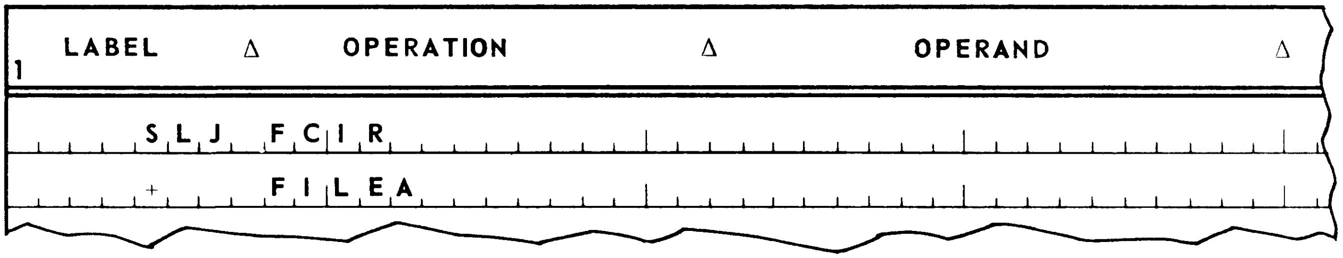


```
1 LABEL      Δ    OPERATION         Δ        OPERAND          Δ
     SLJ    FCIF
     -      FILEA
```

After the file has been closed, control is returned unconditionally to the line immediately following the constant.

The programmer may close an input file at any time he wishes. However, he should close the file after an end-of-file return. Normally, in such a situation, the programmer would indicate that the tape is to be rewound so it can be removed from the UNISERVO tape unit. Exceptions would be if the programmer subsequently wishes to read the tape backward in either this program or a subsequent program. After an input file has been closed, it may be reopened, either as an input file or an output file.

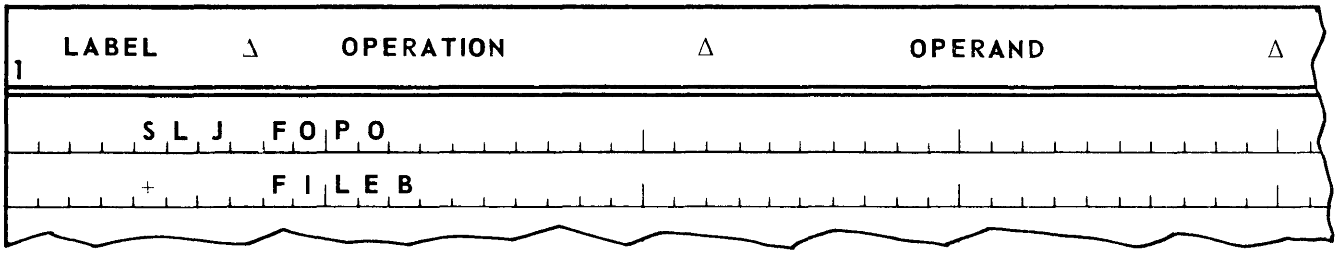
If a multireel file is being read forward and an end-of-reel return has not been requested, the input routine will automatically close a reel and rewind it on detection of end-of-reel sentinels. It will then automatically open the next reel of the file, execute the label check routine if one is called for, and deliver the address of the first item on this next reel via the normal return line. However, if an end-of-reel return has been requested, it then becomes the programmer's responsibility to see that these end-of-reel functions are done. This he can do by means of an SLJ FCIR. This instruction must be followed by a constant that contains the file description table label. For example, to close an input reel (File Close Input Reel) of a file with a file description table labelled FILEA, the following coding would be used:



```
1 LABEL      Δ    OPERATION         Δ        OPERAND          Δ
     SLJ    FCIR
     +      FILEA
```

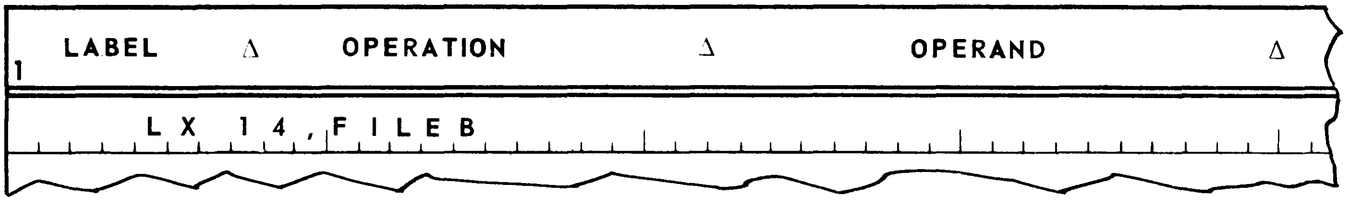
After the reel has been closed, control is returned unconditionally to the line immediately following the constant.

An output file may be opened by executing an SLJ FOPO. This instruction must be followed by a constant that contains the file description table label. Execution of this instruction will cause the specified file description table to be initialized for writing. If the file is to be labelled, the label creation routine will be executed. On return from the label creation routine, the label created in words 16-27 of the file description table will be written. If the sign of the constant is minus, the tape will be rewound before any writing occurs. If such a rewind is not desired, the sign of the constant should be plus. For example, to open an output file (File Open Output) with a file description table labelled FILEB, where the tape is not to be rewound before writing occurs, the following coding would be used:



```
1  LABEL      Δ    OPERATION              Δ              OPERAND               Δ
          SLJ  FOPO
           +   FILEB
```

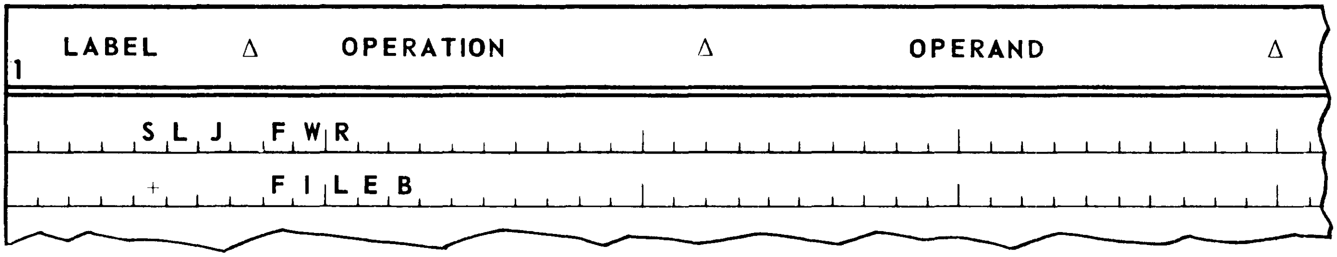
After the output file has been opened, control will be returned unconditionally to the line immediately following the constant. At this point the program can find in the zero word of the file description table the address of the zero word of an output item area in which the first output item may be built up. Thus, the base address of the output item area could be loaded into index register 14 with the following instruction:



```
1  LABEL      Δ    OPERATION              Δ              OPERAND               Δ
          LX   14,FILEB
```

The item area could then be addressed by using index register 14 as its cover index register. The zero word of the item area could also be addressed indirectly by using the contents of FILEB as an indirect address control word.

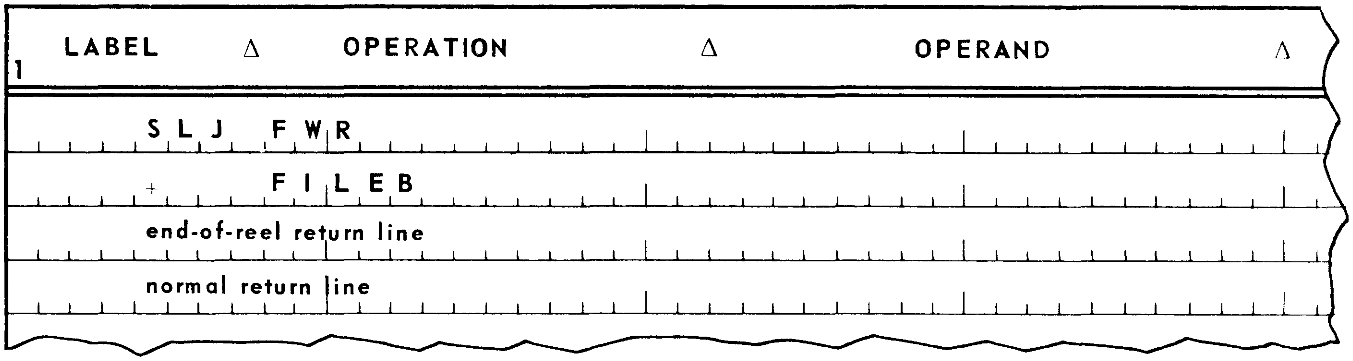
The address of an output item area in which to build up the next output item can be acquired by executing an SLJ FWR. This instruction must be followed by a constant that contains the file description table label. For example, to obtain the address of the next output item area (File WRite) for a file with a file description table labelled FILEB, the following coding would be used:



```
1  LABEL      Δ    OPERATION              Δ              OPERAND               Δ
          SLJ  FWR
           +   FILEB
```

After the file write has been executed, control is normally returned to the second line following the constant (the normal return line). At this point the program can find the address of the zero word of the next output item area in word zero of the file description table.

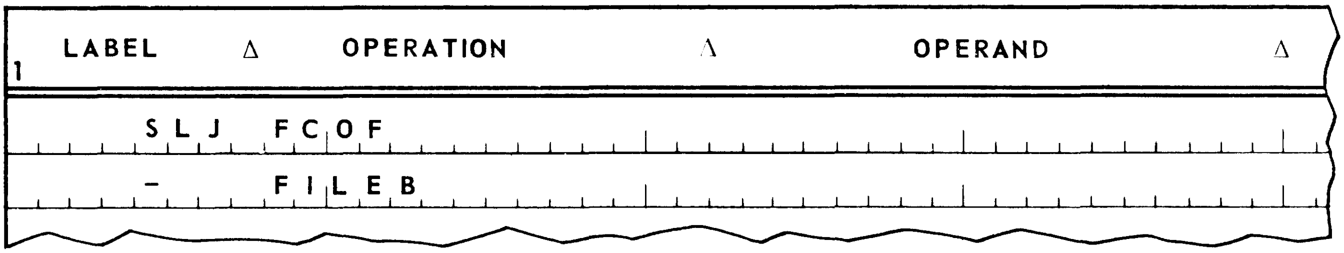
If a one has been placed in bit position four of word nine of the file description table, when control is transferred to the output routine via an SLJ FWR and the output routine detects the end-of-tape warning window on the output tape, then instead of returning control to the normal return line, the output routine will return control unconditionally to the line immediately following the constant. This line is called the *end-of-reel return line* Thus, a schematic of the coding for an SLJ FWR would appear as follows:



```
1  LABEL    Δ    OPERATION          Δ          OPERAND          Δ
        SLJ   FWR
         +    FILEB
        end-of-reel return line
        normal return line
```

When control returns to the end-of reel return line, the address of the zero word of the next output item area will be in the zero word of the file description table just as it is when control returns to the normal return line. This end-of-reel return is normally used to allow the program to write a few more blocks on the tape following the end-of-tape warning window prior to closing the reel. One instance of this use is where a multireel file to drive the printer is being written and it is desired to have end-of-reel coincide with the end of a printing form. If an end-of-reel return is not requested, the contents of the end-of-reel return line are immaterial.

When a program desires to close an output file, this may be done by executing an SLJ FCOF. This instruction must be followed by a constant that contains the file description table label. If the sign of the constant is minus, the tape will not be rewound. For example, to close an output file with a file description table labelled FILEB, where the tape is to be rewound, the following coding would be used:

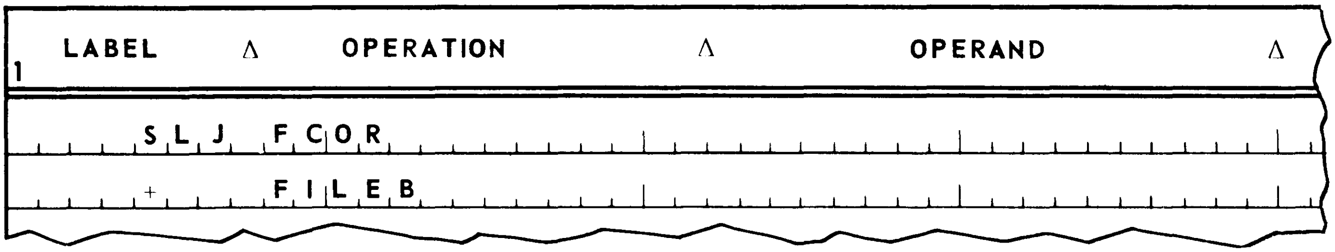


```
1  LABEL    Δ    OPERATION          Δ          OPERAND          Δ
        SLJ   FCOF
         -    FILEB
```

After the file has been closed, control is returned unconditionally to the line immediately following the constant. Normally, the programmer would rewind the tape when closing an output file so it can be removed from the UNISERVO tape unit. Exceptions would be if the programmer subsequently wishes to read the tape backward in either this program or a subsequent program.

Closing an output file causes the output routine to write two end-of-file sentinel blocks on the tape. After an output file has been closed, it may be reopened, either as an input file or an output file.

If a multireel file is being written and an end-of-reel return has not been requested, the output routine will, on detection of the end-of-tape warning window, automatically close a reel, write end-of-reel sentinel blocks on the tape, and rewind it. It will then automatically open the next reel of the file, execute the label creation routine if one is called for, and deliver via the normal return line, the address of an item area in which the first item to be written on this next reel is to be built up. However, if an end-of-reel return is requested, it then becomes the programmer's responsibility to see that these end-of-reel functions are done. This he can do by means of an SLJ FCOR. This instruction must be followed by a constant that contains the file description label. For example, to close an output reel (File Close Output Reel) of a file with a file description table labelled FILEB, the following coding would be used:



```
1  LABEL    Λ    OPERATION        Λ        OPERAND        Λ
        SLJ  FCOR
         +   FILEB
```

After the reel has been closed, control is returned unconditionally to the line immediately following the constant.

The input/output routines use the arithmetic registers; the high, low, equal and sense indicators; and index registers one and two. Consequently, when the program executes an SLJ to the input/output routines, it should not have anything significant in these registers and indicators, since their contents and settings may be destroyed before control is returned to the program.

G. END OF JOB

When a program has completed its work it should turn over control to the executive routine. This is done by executing an SLJ EOJ.

1. Example

Given on UNISERVO tape unit 1 a single reel file with no label block and containing six word items blocked at 83 items per block. The data has the following form:

WORD	DATA
0	NNNNNN
1	GGGGGG▲
2	GGOOOO
3	PP▲OOOO
4	OOAAAA▲
5	AAOOOO

where

N is a taxpayer identification

G is gross income

P is the number of dependents

A is the amount of deductions other than for dependents

Produce on UNISERVO tape unit 2 a file with no label block and containing three word items blocked at 166 items per block. The item has the following form:

WORD	DATA
0	NNNNNN
1	OOOOTT
2	TTTT▲TT

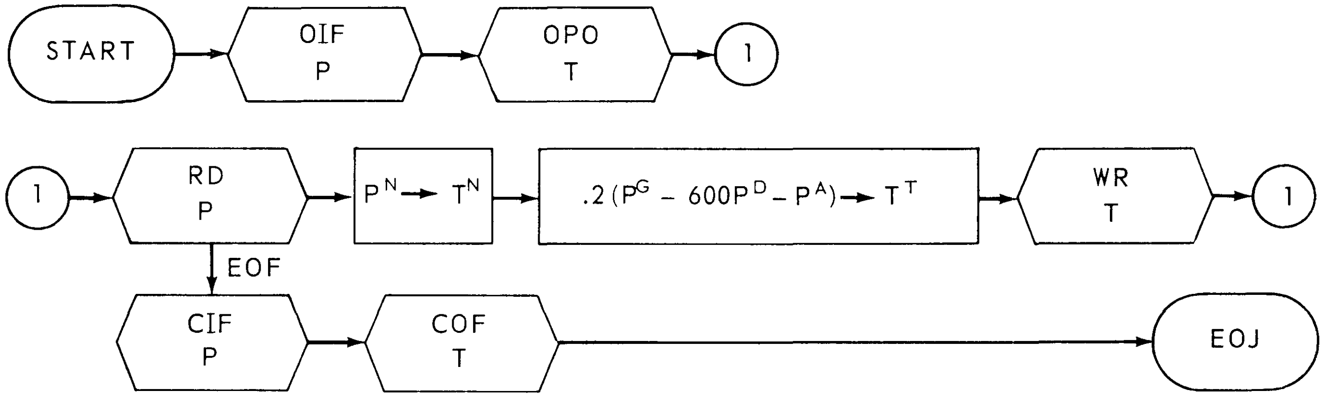
where

N is a taxpayer identification

T is the unrounded tax

A deduction of $600 is allowed for each dependent. The tax is 20% of the taxable income taxable income.

2. Flowchart



LEGEND

P – a taxpayer item

P^N – the identification of P

P^G – the gross income of P

P^D – the number of dependents of P

P^A – the amount of the other deductions of P

T – a tax item

T^N – the taxpayer identification of T

T^T – the tax of T

EOF is an abbreviation for End-of-File

3. Coding

```
LABEL          OPERATION             OPERAND             COMMENTS
AREAP          RES       1328
AREAT          RES       1660
CONTROLP       +         0
               +         0
               +         6
CONTROLT       +         0
               +         0
               +         3
FILEP          +         0
               +         6
               +         83
               +         CONTROLP
               DO        4  ,  +0
               +         83
               +         0
               +         0201
               DO        5  ,  +0
FILET          +         0
               +         3
               +         166
               +         CONTROLT
               DO        6  ,  +0
               +         0202
               DO        5  ,  +0
               USE       3
START       SLJ          BUFC
            +            AREAP
            +            1328
            +            CONTROLP
            SLJ          BUFC
            +            AREAT
            +            1660
            +            CONTROLT
            SLJ          FOIF               OIF P
```

(Coding Continued)

```
1 LABEL     \  OPERATION      Δ        OPERAND          Δ        COMMENTS
              +            FILEP
              SLJ          FOPO                OPO T
              +            FILET
C1            LX     14,   FILET
              SLJ          FRD                 RD P
              +            FILEP
              J            EOFT
              LX     15,   FILEP
              LA     8,    3,     15           .2 ( PG - 600PD - PA ) --- TTT
              DM           (-:060000)
              DS     6,    5,     15
              DA     6,    2,     15
              SA     6,    TS2
              LA     8,    (:000020)
              DM           TS2
              SA     4,    TS4
              DM           TS1
              DA     6,    TS4
              LA     8,    0,     15           PN --- TN
              SA     14,   2,     14
              SLJ          FWR                 WR T
              +            FILET
              +            0
              J            C1
EOFT          SLJ          FCIF                CIF P
              -            FILEP
              SLJ          FCOF                COF T
              -            FILET
              SLJ          EOJ                 EOJ
TS1           +            0
TS2           +            0
TS3           +            :000000
TS4           +            0
              END          START
```

H. COVERING INPUT/OUTPUT AREAS

One nuisance in programming is the specification of the index register when addressing an input or output area. This nuisance can be avoided by using the technique exemplified as follows, which is a recoding of the above exercise.

```
1 LABEL      \ OPERATION       \ OPERAND              \ COMMENTS

             USE               15
PN           EQU               $
PG           EQU               $+2
PD           EQU               $+3
PA           EQU               $+5
AREAP        RES               1328
             USE               14
TT           EQU               $+2
AREAT        RES               1660
CONTROLP     +                 0
             +                 0
             +                 6
CONTROLT     +                 0
             +                 0
             +                 3
FILEP        +                 0
             +                 6
             +                 83
             +                 CONTROLP
             DO                4,+0
             +                 83
             +                 0
             +                 0201
             DO                5,+0
FILET        +                 0
             +                 3
             +                 166
             +                 CONTROLT
             DO                6,+0
             +                 0202
```

(Coding Continued)

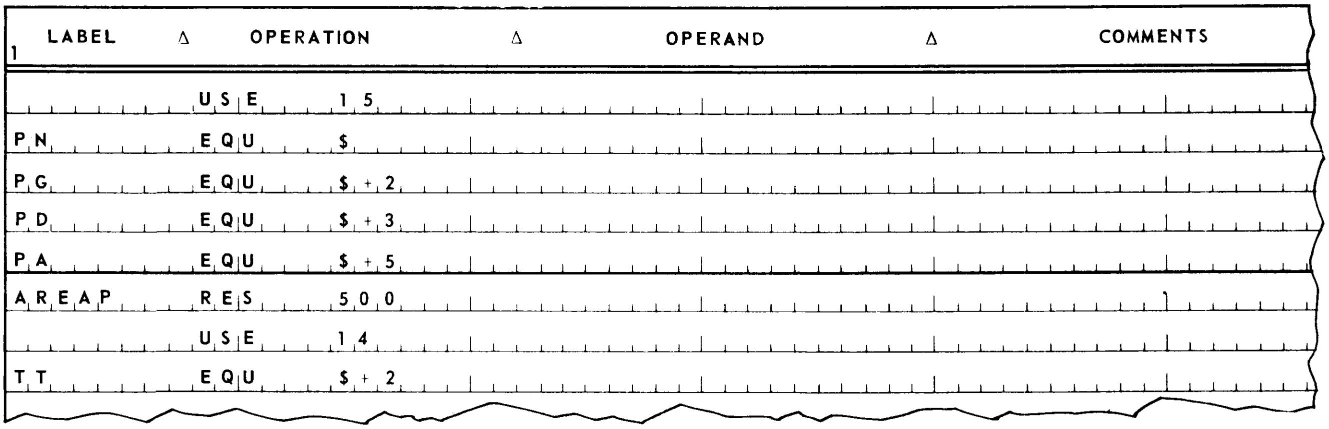
```
  LABEL    Δ   OPERATION       Δ          OPERAND          Δ          COMMENTS
1
               DO              5,   0
               USE             3
START          SLJ             BUFC
                               AREAP
                               1328
                               CONTROLP
               SLJ             BUFC
                               AREAT
                               1660
                               CONTROLT
               SLJ             FOIF               OIF  P
                               FILEP
               SLJ             FOPO               OPO  T
                               FILET
C1             LX      14,     FILET
               SLJ             FRD                RD  P
                               FILEP
               J               EOFT
               LX      15,     FILEP
               LA      8,      PD           .2  ( PG  - 600PD  -  PA )  ---  TT
               DM              (-:060000)
               DS      6,      PA
               DA      6,      PG
               SA      6,      TS2
               LA      8,      (:000020)
               DM              TS2
               SA      4,      TS4
               DM              TS1
               DA      6,      TS4
               LA      8,      PN           PN  ---  TN
```

(Coding Continued)

```
LABEL          OPERATION          OPERAND                          COMMENTS
               SA   14,     TT
               SLJ          FWR        WR ---  T
               +            FILET
               +            0
               J            C1
EOFT           SLJ          FCIF       CIF  P
               -            FILEP
               SLJ          FCOF       COF  T
               -            FILET
               SLJ          EOJ        EOJ
TS1            +            0
TS2            +            0
TS3            +            :000000
TS4            +            0
               END          START
```

The mechanism used in the above coding works as follows. The USE 15 directive tells the UTMOST assembler to assume that the address of the line tagged AREAP is stored in index register 15. The PN EQU $ directive tells the assembler that the label PN is equivalent to the label AREAP. The PG EQU $ + 2 directive says that the label PG is equivalent to the expression AREAP + 2. The PP EQU $ + 3 directive says that label PP is equivalent to the expression AREAP + 3. And so on. Consequently, when the assembler encounters the label PN in the m portion of an instruction, it will substitute a binary zero in the m portion of the instruction and insert a specification of index register 15 in the index register portion of the instruction. The assembler will do likewise for the other labels defined by the EQU directives. Since the program keeps the base address of the current taxpayer item in index register 15, the fields in the item can therefore be properly addressed with the labels defined by the EQU directives. The USE 14 directive and its associated EQU directives set up the labels to address the tax item in a similar fashion.

In the above example more than 1024 storage locations were reserved between the USE 15 directive and the USE 14 directive. Suppose this were not the case. Suppose instead the situation were as follows:



```
1 LABEL   Δ OPERATION   Δ OPERAND   Δ COMMENTS
          USE     15
PN        EQU     $
PG        EQU     $+2
PD        EQU     $+3
PA        EQU     $+5
AREAP     RES     500
          USE     14
TT        EQU     $+2
```

In the above example, the label TT is set equal to a relative address that is covered by both index registers 14 and 15. Is there any assurance that, when the assembler encounters the label TT in the m portion of an instruction, it will not insert index register 15 in the index register portion of the instruction rather than index register 14? Happily, there is.

The assembler keeps a list of cover index registers. This list is always arranged by index register number. When the assembler encounters a label in the m portion of an instruction it begins searching down its cover index register list to determine if there is an index register that covers this label. It always searches the list from top to bottom, and if the list contains more than one index register that covers the same label, it always selects for insertion into the index register portion of the instruction the number of the first index register it encounters which covers the label. Thus, in the above example, the assembler will always find that index register 14 covers the label TT before it finds that index register 15 covers the label.

I. MASTER FILE TAPE HANDLING

The above example handled the taxpayer file (a transaction file read into the store but never written out) and the tax file (a report file written out of memory but never read in). For transaction and report files the techniques so far described are adequate. Each file is accociated with a reserved area, the items in the file are accessed by means of the instruction SLJ FRD for input files and SLJ FWR for output files, and any information desired from an input item or desired in an output item is moved out of or into the item by means of LA and SA instructions.

However, now consider the handling of a master file, which is both read into and written out of store. To handle such a file with the same techniques used for transaction and report files would require that every master item read in be moved to an output area before it can be written out. To avoid this movement of master items, the UTMOST system provides a special input/output technique. Utilization of this technique involves reserving a common input/output area for both the input and the output master items. To obtain simultaneous read, write, and compute, this area should have the capacity to hold two blocks of data for each input file sharing the common area, and one block of data for each output file sharing the common area. To minimize interlock, more area can be set aside for each input and output file if such space is available. The maximum area that can be set aside for an input file is an area capable of storing one less than the number of items in three blocks.

As a consequence of having a common input/output area for both input and output master files, only one control packet and one SLJ BUFC operation is required for the area. However, a file description table must be provided for each file using the common area.

Each input file using the common area must be opened with an SLJ FOIF; be closed with an SLJ FCIF; and if the end of reel return has been requested, each reel of the file must be closed with an SLJ FCIR. Each output file using the common area must be opened with an SLJ FOPO; be closed with an SLJ FCOF; and if the end of reel return has been requested, each reel must be closed with an SLJ FCOR. If an input item is to be deleted from the master file, it can be skipped and the address of another input item acquired by means of an SLJ FRD. If an item is to be added, it may be moved to the current output item area, and the address of another output item area can be acquired by means of an SLJ FWR. For writing a master item that has been read, the following calling sequence is provided:

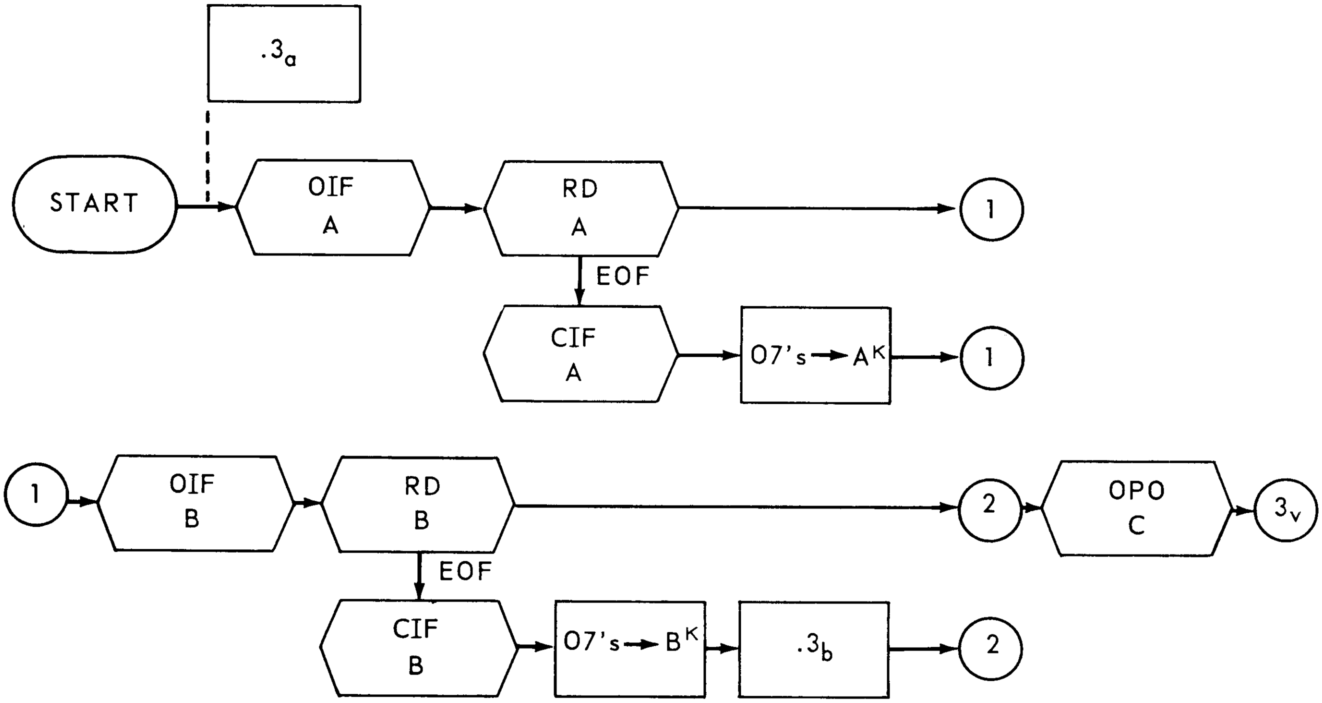
```
1  LABEL    Δ    OPERATION         Δ         OPERAND         Δ
           SLJ    FRDW
           +      i
           +      o
           end of file (reel) return
           normal return
```

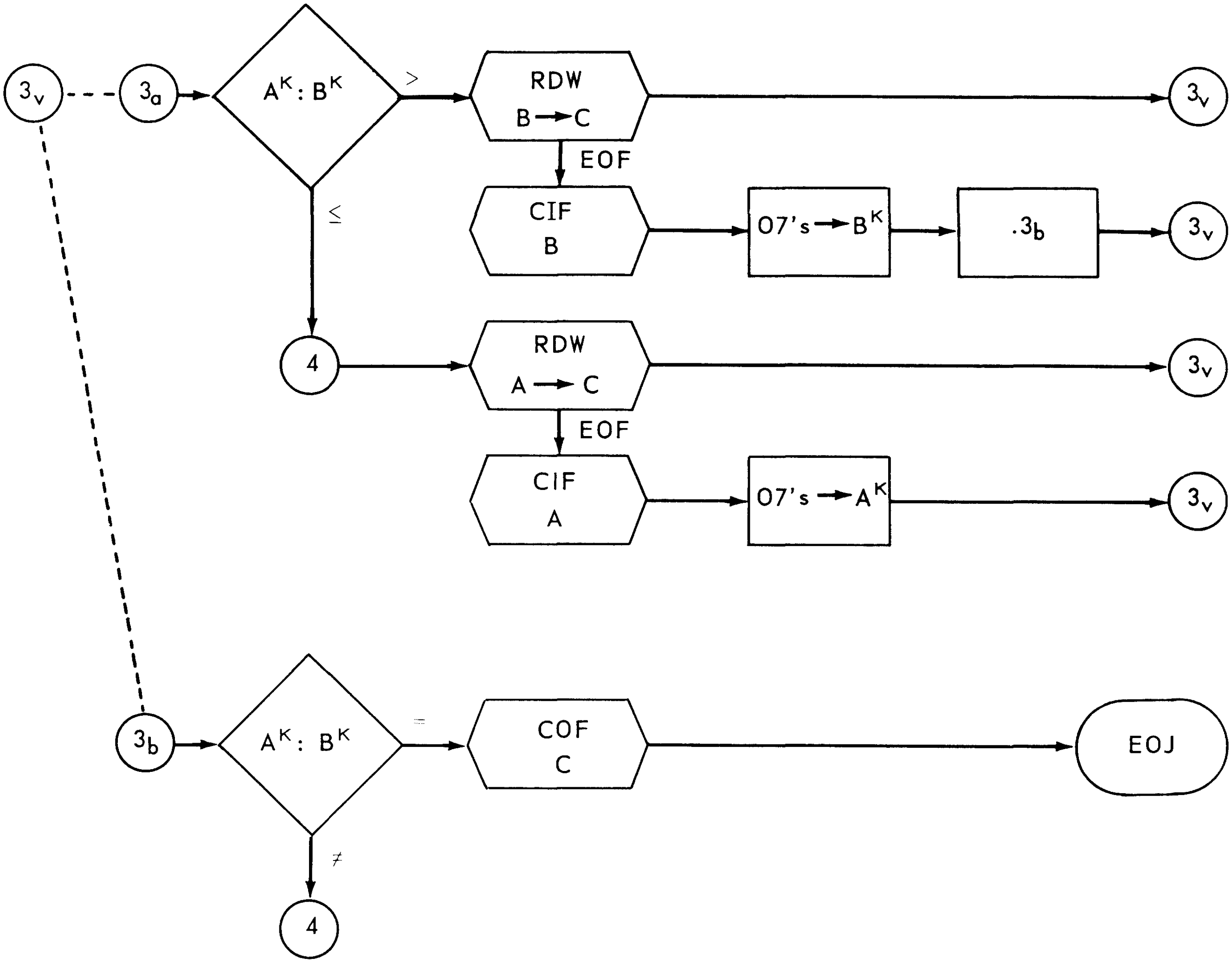
where *i* is the label of the zero word of the file description table for an input file and *o* is the label of the file description table for an output file. Execution of an SLJ FRDW will cause the output routine to see to it that the item the address of whose zero word is in the zero word of the input file description table specified is written on the output file whose file description table is specified. When control is returned to the program, the address of the zero word of the next input item can be accessed in the zero word of the input file description table. Normally, control is returned to the normal return line. If end of file sentinels are detected on the input file, control will return to the end of file (reel) return. If an end of reel return has been requested for the input file, control will also return to the end of file (reel) return on detection of end of reel sentinels. If end of reel return has been requested for the output file, control will be returned to the end of file (reel) return on detection of the end of tape warning window. If end of reel return has been requested for both the input and the output file, the contents of arithmetic register 8 can be inspected after an end of file (reel) return to determine whether the input or output generated this return. If generated by the input, arithmetic register 8 will contain a binary one. If generated by the output, a binary two. If end of reel occurs on both input and output simultaneously, arithmetic register 8 will contain a binary three. (See FRD on page 9-24 for a description of how to distinguish between an end-of-file and an end-of-reel return.)

1. Example

   Given on tape unit 1 a single reel file called the A file, on tape unit 2 a single reel file called the B file. Neither has a label block. Both contain 25 word items blocked at 20 items per block. The first two words of each item is a key. Each file is in ascending order by key. Merge the files to produce a new file called file C, on tape unit 3. File C is also blocked at 20 items per block.

2. Flowchart





LEGEND

A – an item

A^K – the key of A

B – another item

B^K – the key of B

C – a third item

07's is an abbreviation for a positive all binary ones.

3. Coding

```
LABEL          OPERATION          OPERAND          COMMENTS

               USE                15
AK             EQU                $+1
AREA           RES                25
               USE                14
BK             EQU                $+1
               RES                2700
CONTROL        +                  0
               +                  0
               +                  25
FILEA          +                  0
               +                  25
               +                  20
               +                  CONTROL
               DO                 4,+0
               +                  20
               +                  0
               +                  0201
               DO                 5,+0
FILEB          +                  0
               +                  25
               +                  20
               +                  CONTROL
               DO                 4,+0
               +                  20
               +                  0
               +                  0202
               DO                 5,+0
FILEC          +                  0
               +                  25
               +                  20
```

(Coding Continued)

```
1 LABEL      OPERATION      OPERAND                          COMMENTS
             +              CONTROL
             DO             6 ,  +0
             +              0203
             DO             5 ,  +0
             USE            3
START        SLJ            BUFC
             +              AREA
             +              2700
             +              CONTROL
             SLJ            FOIF              OIF A
             +              FILEA
             SLJ            FRD               RD A
             +              FILEA
             J              EOFAI
             LX     15,     FILEA
C1           SLJ            FOIF              OIF B
             +              FILEB
             SLJ            FRD               RD B
             +              FILEB
             J              EOFBI
             LX     14,     FILEB
C2           SLJ            FOPO              OPO C
             +              FILEC
C3           LA     12,     AK                AK : BK
             C      12,     BK
L1           JG             L2
             SLJ            FRDW              RDW A - - - C
             +              FILEA
             +              FILEC
             J              EOFA
```

(Coding Continued)

LABEL	OPERATION	OPERAND	COMMENTS
	LX 15,	FILEA	
	J	C3	
L2	SLJ	FRDW	RDWB---C
	+	FILEB	
	+	FILEC	
	J	EOFB	
	LX 14,	FILEB	
	J	C3	
EOFAI	SLJ	FCIF	CIFA
	–	FILEA	
	LX 15,	((TWC 07777777777777777))	07'S---A
	J	C1	
EOFBI	SLJ	FCIF	CIFB
	–	FILEB	
	LX 14,	((TWC 07777777777777777))	07'S---BK
	LA 8,	(JE L3)	.3B
	SA 8,	L1	
	J	C2	
EOFA	SLJ	FCIF	CIFA
	–	FILEA	
	LX 15,	((TWC 07777777777777777))	07'S---AK
	J	C3	
EOFB	SLJ	FCIF	CIFB
	–	FILEB	
	LX 14,	((TWC 07777777777777777))	07'S---BK
	LA 8,	(JE L3)	.3B
	SA 8,	L1	
	J	C3	
L3	SLJ	FCOF	COFC
	–	FILEC	
	SLJ	EOJ	EOJ
	END	START	

J. LABEL HANDLING

How labels are handled is a user option. The following example shows one possible approach. This example involves reopening a file after it has been closed.

1. Example

Given on tape unit 1 a file called the A file, on tape unit 3 a file called the B file. Both contain 25 word items blocked at 20 items per block. The first two words of each item is a key. The file identification of the A file is MASTERAA, of the B file MASTERBB. The date of cycle of each is 630201. If the label is not as is expected, type out:

```
LABL  IS  tf  td  tr  SHD  BE  pf  pd  pr
```

where

tf is the file identification on the tape

td is the date of cycle on the tape

tr is the reel count on the tape

pf is the file identification the program expects

pd is the date of cycle the program expects

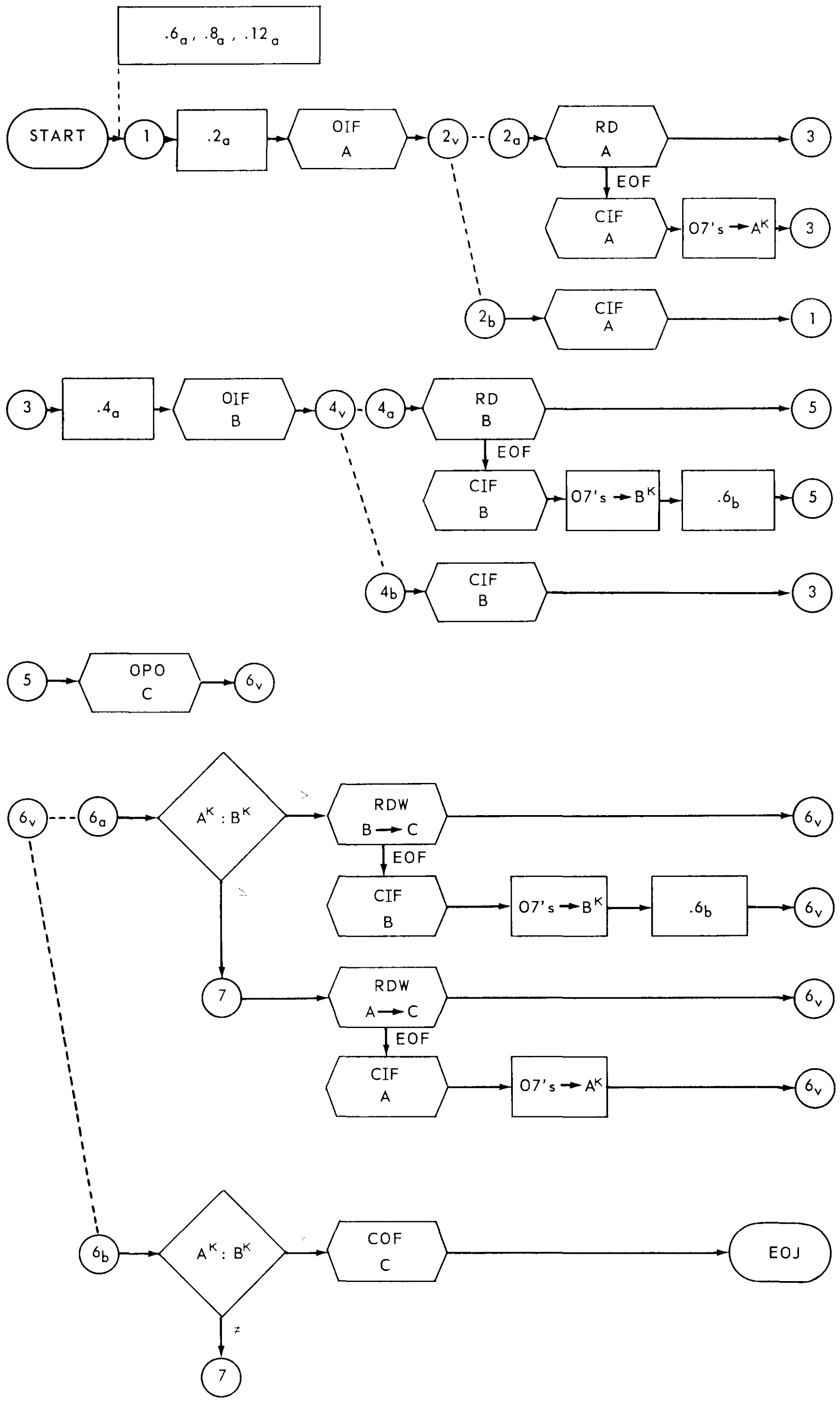
pr is the reel count the program expects

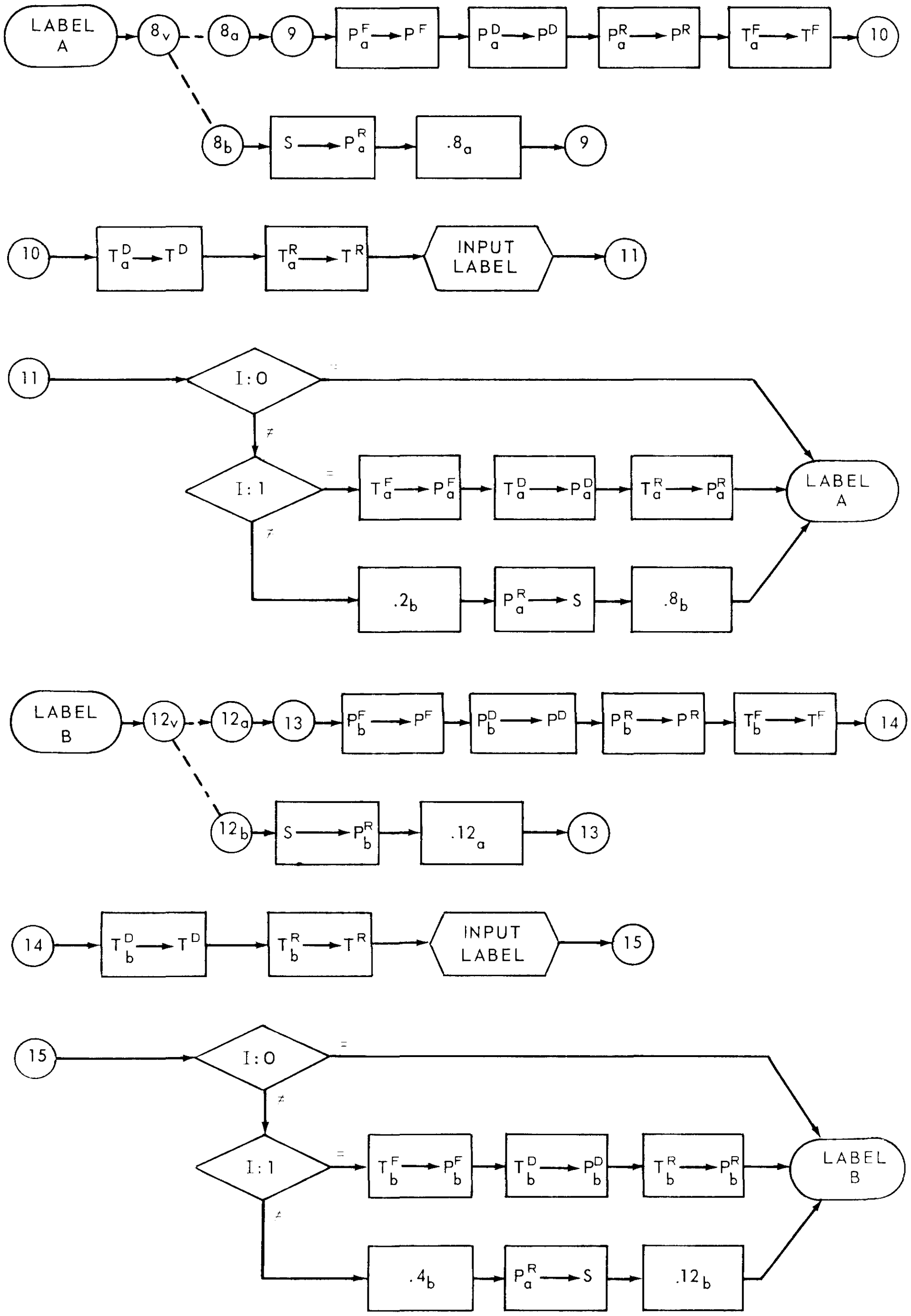
Then accept a typein, which may be FORCE or RECHECK. If the typein is FORCE, substitute the file identification, date of cycle, and reel number on the tape for those that the program expects, and continue with the program. If the typein is RECHECK, close the file and reopen it. If anything else is typed in, reinitiate the typeout.

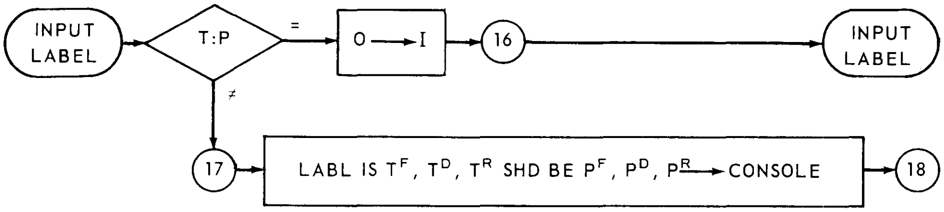
Merge the files to produce a new file, called file C, on tape unit 5. File C is also blocked at 20 items per block, its file identification is MASTERCC, and its date of cycle 630202.

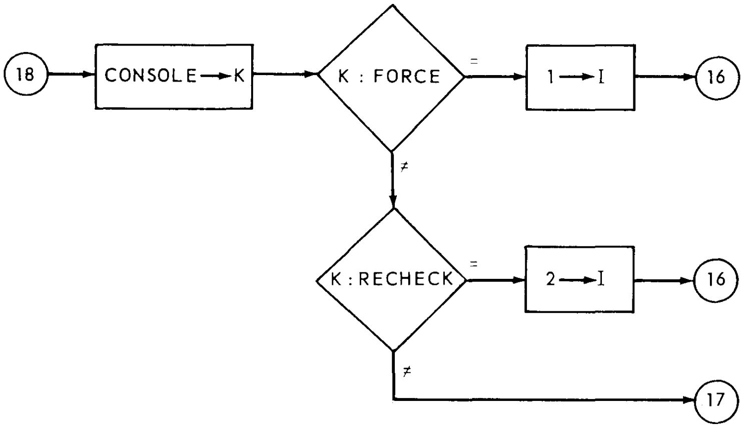
Subsequent references to this example in later portions of this manual will refer to it as the *two way merge.*

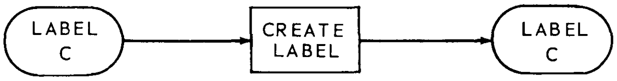
2. Flowchart


.6a , .8a , .12a
START
1
.2a
OIF A
2v
2a
RD A
3
EOF
CIF A
07's → A^K
3
2b
CIF A
1
3
.4a
OIF B
4v
4a
RD B
5
EOF
CIF B
07's → B^K
.6b
5
4b
CIF B
3
5
OPO C
6v
6v
6a
A^K : B^K
>
RDW B → C
6v
EOF
≤
CIF B
07's → B^K
.6b
6v
7
RDW A → C
6v
EOF
CIF A
07's → A^K
6v
6b
A^K : B^K
=
COF C
EOJ
≠
7


LABEL A
8_v
8_a
9
$P_a^F \to P^F$
$P_a^D \to P^D$
$P_a^R \to P^R$
$T_a^F \to T^F$
10
8_b
$S \to P_a^R$
$.8_a$
9
10
$T_a^D \to T^D$
$T_a^R \to T^R$
INPUT LABEL
11
11
I:0
=
≠
I:1
=
$T_a^F \to P_a^F$
$T_a^D \to P_a^D$
$T_a^R \to P_a^R$
LABEL A
≠
$.2_b$
$P_a^R \to S$
$.8_b$
LABEL B
12_v
12_a
13
$P_b^F \to P^F$
$P_b^D \to P^D$
$P_b^R \to P^R$
$T_b^F \to T^F$
14
12_b
$S \to P_b^R$
$.12_a$
13
14
$T_b^D \to T^D$
$T_b^R \to T^R$
INPUT LABEL
15
15
I:0
=
≠
I:1
=
$T_b^F \to P_b^F$
$T_b^D \to P_b^D$
$T_b^R \to P_b^R$
LABEL B
≠
$.4_b$
$P_a^R \to S$
$.12_b$







LEGEND

A	– an item
A^K	– the key of A
P_a	– the expected A label
P_a^F	– the file identification of P_a
P_a^D	– the date of cycle of P_a
P_a^R	– the reel number of P_a
T_a	– the A tape label
T_a^F	– the file identification of T_a
T_a^D	– the date of cycle of T_a
T_a^R	– the reel number of T_a
B	– another item
B^K	– the key of B
P_b	– the expected B label
P_b^F	– the file identification of P_b
P_b^D	– the date of cycle of P_b
P_b^R	– the reel number of P_b
T_b	– the B tape label
T_b^F	– the file identification of T_b
T_b^D	– the date of cycle of T_b
T_b^R	– the reel number of T_b
C	– a third item
P	– a general expected label
P^F	– the file identification of P
P^D	– the date of cycle of P
P^R	– the reel number of P
T	– a general tape label
T^F	– the file identification of T
T^D	– the date of cycle of T
T^R	– the reel number of T
S	– a storage
I	– an indicator
K	– a typein

3. Coding

```
  LABEL   Δ OPERATION    Δ    OPERAND     Δ    COMMENTS
1                                                          72
                    USE       15
AK                  EQU       $+1
AREA                RES       25
                    USE       14
BK                  EQU       $+1
                    RES       2700
CONTROL             +         0
                    +         0
                    +         25
FILEA               +         0
                    +         25
                    +         20
                    +         CONTROL
                    DO        4  ,  +0
                    +         20
                    +         0
                    +         0201
                    +         0
                    +         'MAST'
                    +         'ERAA'
                    +         0
                    +         LABELA
                    DO        12  ,  +0
FILEB               +         0
                    +         25
                    +         20
                    +         CONTROL
                    DO        4  ,  +0
                    +         20
                    +         0
                    +         0203
                    +         0
                    +         'MAST'
                    +         'ERBB'
                    +         0
                    +         LABELB
                    DO        12  ,  +0
FILEC               +         0
                    +         25
                    +         20
                    +         CONTROL
                    DO        6  ,  +0
                    +         0205
                    +         0
                    +         'MAST'
                    +         'ERCC'
                    +         :000001
                    +         LABELC
```

(Coding Continued)

```
LABEL      OPERATION      OPERAND                                   COMMENTS
           DO             12 , +0
           USE            3
START      SLJ            BUFC
           +              AREA
           +              2700
           +              CONTROL
TABBWORD   FORM           1,6,3,15
           LA             8,(TABBWORD 0,'W',0,TYPEIN)
           SA             8,TABB
C1         SZ             C2                                        .2A
           SLJ            FOIF                                      OIF A
           +              FILEA
C2         NOP            0
           SLJ            FRD                                       RD A
           +              FILEA
           J              EOFAI
           LX     15,     FILEA
C3         SZ             C4                                        .4A
           SLJ            FOIF                                      OIF B
           +              FILEB
C4         NOP            0
           SLJ            FRD                                       RD B
           +              FILEB
           J              EOFBI
           LX             14,FILEB
C5         SLJ            FOPO                                      OPO C
           +              FILEC
C6         LA     12,     AK                                        AK : BK
           C      12,     BK
L1         JG             L2
           SLJ            FRDW                                      RDW A --- C
           +              FILEA
           +              FILEC
           J              EOFA
           LX     15,     FILEA
           J              C6
L2         SLJ            FRDW                                      RDW   --- C
           +              FILEB
           +              FILEC
           J              EOFB
           LX     14,     FILEB
           J              C6
EOFAI      SLJ            FCIF                                      CIF A
           -              FILEA
           LX     15,     ((T C 07777777777777777))                 0 7'S --- AK
           J              C3
```

(Coding Continued)

```
LABEL      Δ   OPERATION      Δ        OPERAND          Δ         COMMENTS
1                                                                          72
EOFBI              SLJ      FCIF                    CIF  B
                   -        FILEB
                   LX  14,  ((TWC 0777777777777777))          0 7'S --- BK
                   LA  8,   (JE L3)                 .6B
                   SA  8,   L1
                   J        C5
EOFA               SLJ      FCIF                    CIF  A
                   -        FILEA
                   LX  15,  ((TWC 0777777777777777))          0 7'S --- AK
                   J        C6
EOFB               SLJ      FCIF                    CIF  B
                   -        FILEB
                   LX  14,  ((TWC 0777777777777777))          0 7'S --- BK
                   LA  8,   (JE L3)                 .6B
                   SA  8,   L1
                   J        C6
L3                 SLJ      FCOF                    COF  C
                   -        FILEC
                   SLJ      EOJ                     EOJ
LABELA             NOP      0
C8                 LA  11,  FILEA+19                TAF --- TF ; TAD --- TD
                   LA  4,   FILEA+26                TAR --- TR
                   SA  15,  TAPELABEL
                   LA  12,  FILEA+13                PAF --- PF ; PAR     PR
                   LA  2,   DATEA                       PAD --- PD
                   SA  15,  PROGRAMLABEL
                   SLJ      INPUTLABEL                  INPUTLABEL
                   C   8,   (0)                         I : 0
                   JE       *LABELA
                   C   8,   (1)                         I : 1
                   JE       L4
                   LA  8,   (J C2B)                     .2B
                   SA  8,   C2
                   LA  8,   FILEA+14                    PAR --- S
                   SA  8,   S
                   LA  8,   (J C8B)                     .8B
                   SA  8,   C8
                   J        *LABELA
L4                 LA  15,  TAPELABEL                   TAF --- PAF : TAD --- PAD
                   SA  12,  FILEA+13                    TAR --- PAR
                   SA  2,   DATEA
                   SA  1,   FILEA+14
                   J        *LABELA
C8B                LA  8,   S                           S --- PAR
                   SA  8,   FILEA+14
                   LA  8,   (LA 11,FILEA+19)      .8A
                   SA  8,   C8
                   J        C8
DATEA              +        :630201
```

(Coding Continued)

```
 LABEL            OPERATION          OPERAND                    COMMENTS
S                 +                  0
C2B               SLJ                FCIF                       CIF A
                  -                  FILEA
                  J                  C1
LABELB            NOP                0
C12               LA    11,          FILEB+19                   TBF --- TF ; TBD --- TD
                  LA    4,           FILEB+26                   TBR --- TR
                  SA    15,          TAPELABEL
                  LA    12,          FILEB+13                   BBF --- PF ; PBR --- PR
                  LA    2,           DATEB
                  SA    15,          PROGRAMLABEL
                  SLJ                INPUTLABEL                 INPUTLABEL
                  C     8,           (0)                        I : 0
                  JE                 *LABELB
                  C     8,           (1)                        I : 1
                  JE                 L5
                  LA    8,           (J C4B)                    .4B
                  SA    8,           C4
                  LA    8,           FILEB+14                   PBR --- S
                  SA    8,           S
                  LA    8,           (J C12B)                   .12B
                  SA    8,           C12
                  J                  *LABELB
L5                LA    15,          TAPELABEL                  TBF --- PBF ; TBD --- PBD
                  SA    12,          FILEB+13
                  SA    2,           DATEB
                  SA    1,           FILEB+14
                  J                  *LABELB
C12B              LA    8,           S                          S --- PAR
                  SA    8,           FILEB+14
                  LA    8,           (LA 11,FILEB+19)           .12A
                  SA    8,           C12
                  J                  C12
DATEB             +                  :630201
C4B               SLJ                FCIF                       CI B
                  -                  FILEB
                  J                  C3
INPUTLABEL        NOP                0
                  LA    15,          TAPELABEL                  T : P
                  C     15,          PROGRAMLABEL
                  JE                 L7
                  SA    12,          MESSAGEOUT+3               LABL IS TF, TD, TR SHD BE PF,
                  SA    3,           DATEANDREEL                PD, PR --- CONSOLE
                  SLJ                EDITDATEANDREEL
                  LA    14,          OUTPUT
```

(Coding Continued)

```
LABEL              OPERATION      OPERAND                          COMMENTS
                   SA     14,     MESSAGEOUT+6
                   LA     15,     PROGRAMLABEL
                   SA     12,     MESSAGEOUT+10
                   SA      3,     DATEANDREEL
                   LA     14,     OUTPUT
                   SA     14,     MESSAGEOUT+13
C17                LA      1,     (MESSAGEOUT,1)
                   +              0740000
                   +              037777
                   J              C17
                   SZ             TYPEIN                           CONSOLE --- K
L6                 IXC     0,     TYPEIN
                   JE             L6
                   LA     12,     MESSAGEIN                        K : FORCE
                   C      12,     ('FORCE')
                   JE             L8
                   C      12,     ('RECHECK ')                     K : RECHECK
                   JE             L9
                   J              C17
L7                 LA             8,(0)                            0 --- I
                   J              *INPUTLABEL
L8                 LA             8,(1)                            1 --- I
                   J              *INPUTLABEL
L9                 LA             8,(2)                            2 --- I
                   J              *INPUTLABEL
                   +              0
                   +              0
                   +              0
TAPELABEL          +              0
                   +              0
                   +              0
                   +              0
PROGRAMLABEL       +              0
MESSAGEOUT         +              'LABL'
                   +              ' IS '
                   DO             5 , +0
                   +              ' SHD'
                   +              ' BE '
                   DO             5 , +0
                   +              023000000
EDITDATEANDREEL    NOP            0
                   LA      8,     DATEANDREEL-1
                   LA      4,     (0)
                   DSR    12,     1
                   AND    12,     (TWC 037777777400000000)
                   SAA    12,     OUTPUT
                   LA      8,     DATEANDREEL
                   DSL     8,     2
                   SAA    12,     TS3
                   LA      8,     TS1
```

(Coding Continued)

```
1 LABEL   \   OPERATION      \        OPERAND        \        COMMENTS     72
                    SA  8,       OUTPUT
                    J            *EDITDATEANDREEL
                    +            0
DATEANDREEL         +            0
                    +            0
                    +            0
OUTPUT              +            0
TS1                 +            0
TS2                 +            0
TS3                 +            0
TYPEIN              +            'BUSY'
                    LA  12,      2,    2
                    SA  12,      5,    1
                    J            *0,   1
                    +            0
MESSAGEIN           +            0
LABELC              NOP          0
                    LA  8,       (-0)                      CREATE LABEL
                    SA  8,       FILEC+16
                    SA  8,       FILEC+27
                    LA  12,      FILEC+13
                    SA  8,       FILEC+17
                    SA  4,       FILEC+26
                    LA  8,       (:630202)
                    LA  4,       FILEC+14
                    SA  12,      FILEC+19
                    J            *LABELC
                    END          START
```

Exercise

Given on tape unit 1 a master file, on tape unit 2 a change file. Both are single reel files; are blocked at 20 items per block; and contain 17 word items, each of the following form:

WORD	DIGIT	DATA
0-1		Policy Number
2	1	Billing Code
2	4	Mode Code
2	5	Identification Code
3-7		Name
8-16		Address

In both files there may be several items with the same policy number. Only those items with an identification code of 1 are to be acted on. All master items with an identification code other than 1 are to be written on an updated master file on tape unit 3 with no other action taken. Change items with an identification code other than 1 are to be skipped. For each policy number in the master file, there is one and only one item with an identification code of 1. Both input files are in ascending order by policy number. If the mode code of an active change item (a change item with an identification code of 1) is 1, substitute the name field of the change item for the name field of the active master item with the matching ploicy number. If the mode code of an active change item is 4, substitute the address field of the change item for the address field of the active master item with the matching policy number. If the mode code of an active change item is other than 1 or 4, delete the active master item with the matching policy number. An active master item may or may not have an active change item with a matching policy number. However, no active master item has more than one active change item with a matching policy number. If the policy number of an active change item is not equal to the policy number of any active master item, add the change item to the master file. All items written on the updated master file that have an identification code of 1 and a billing code of 0 are also to be written on a billing file on tape unit 4. Both output files are single reel files and are to be blocked at 20 items per block. The file identification of the input master file is MASTERAA, the date of cycle 620303. Of the change file CHANGEIN, 620303. The updated master MASTERAA, 620304. The report REPORTOT, 620304. If the label of an input file is not as is expected, typė out:

LABL IS tf td tr SHD BE pf pd pr

where

tf is the file identification on the tape

td is the date of cycle on the tape

tr is the reel count on the tape

pf is the file identification the program expects

pd is the date of cycle the program expects

pr is the reel count the program expects

Then accept a typein, which may be FORCE or RECHECK. If the typein is FORCE, substitute the file identification, date of cycle, and reel number on the tape for those that the program expects, and continue with the program. If the typein is RECHECK, close the file and reopen it. If anything else is typed in, reinitiate the typeout.

5. Test

A single reel, unlabeled inventory file mounted on tape unit 1 is blocked at 50 items per block and contains items of the form:

WORD	DATA
0	NNNNNN
1	0 HHHHH▲
2-9	XXXXXX

where

N is the stock number

H is the onhand quantity

X is other data

A single reel, unlabeled sales file mounted on UNISERVO tape unit 2 is blocked at 250 items per block and contains items of the form:

WORD	DATA
0	NNNNNN
1	0 QQQQQ▲

where

N is the stock number

Q is the sales quantity

The items on both files are arranged in ascending order by stock number. There may be more than one sales item for a given inventory item. Produce on tape unit 3 a single reel, unlabeled updated inventory file blocked at 50 items per block. The sales items will exhaust before the inventory items.

K. PROCESSOR ERRORS

The Processor checks itself against the occurrence of certain failures, or errors. If the Processor detects the occurrence of such an error, a *processor error interrupt* occurs. The contents of the control counter are stored in storage location 16, and control is transferred to the instruction stored in location 17. As a result of a processor error interrupt, the executive routine will type out notification of the processor error, the locations at which it occurred and will enter a closed loop to await operator action.

L. RERUN

Under certain circumstances, such as the occurrence of a processor error, it becomes necessary to start the execution of a program over again. If the time to execute a program is long, it may be undesirable to always restart the program from the beginning. To protect against such a necessity, points can be set up during program execution at which the program can be "restarted". These points are called *rerun points*. To set up a rerun points, it is necessary to make a record of:

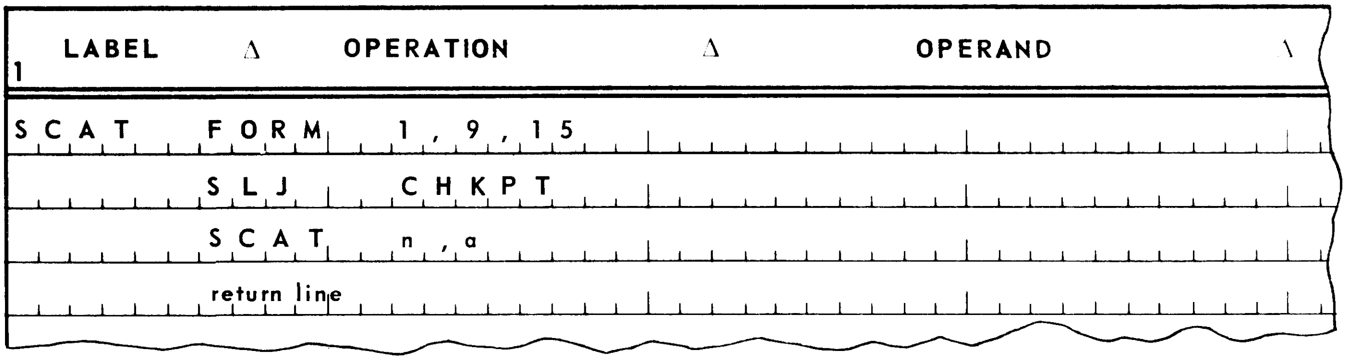
1) The state of the program at the rerun point.

2) The tapes mounted at the rerun point.

3) The position of the read write head on these tapes at the rerun point.

All this information can be recorded by writing the pertinent contents of the memory on tape. Such a record is called a *memory dump*. After a memory dump has been made, if *rerun* from the memory dump becomes necessary, all that is required is to reconstitute the store from the dump, mount the proper tapes, and reposition them.

Taking memory dumps, reconstituting the store from a memory dump, and repositioning tapes are all functions of the executive routine. The sole programmer responsibility is to request memory dumps from the executive routine at those points at which the setting up of rerun points is desired. The request is made by the following calling sequence.



```
1 LABEL   Δ   OPERATION         Δ         OPERAND        Δ
SCAT      FORM   1,9,15
          SLJ    CHKPT
          SCAT   n,a
          return line
```

where

a is the label of a list of labels of the zero words of the file description tables in the program

n is the number of entries in this list

After the executive routine takes the memory dump, control is returned unconditionally to the return line.

13. SYMBIONTS

The UNIVAC III System can be used in one of two configurations.

1. *As a concurrent processor with online peripherals.*

2. *As a tape processor using the UNIVAC 1050 System as a satellite to handle peripheral operations.*

The purpose of this chapter is to discuss the use of the UNIVAC III System in the first of these configurations.

To develop the principles to be stated in this chapter, a simple, abstract computer application is assumed. This application has the following characteristics. A transaction file is brought to the computer in the form of a card deck. The transaction file is applied to a master file for updating purposes. The master file is stored on magnetic tape, and the updated master file is produced on the same medium. As a by product of the updating process, a printed report is prepared.

A. MAGNETIC TAPE COMPUTERS WITH OFFLINE PERIPHERAL OPERATIONS

One common computer hardware configuration is a central processor, the only mass input and output of which is magnetic tape. Typically, such a central processor is serviced by a collection of peripheral devices each of which has the capability to perform one conversion function between magnetic tape and some other medium. Thus, there is a card to magnetic tape converter, a printer driven by magnetic tape, a paper tape to magnetic tape converter, a magnetic tape to card converter, and so on. Or the Central Processor can be serviced by a satellite computer, which combines in its features the ability to do the required conversions: card to magnetic tape, magnetic tape to printed copy, paper tape to magnetic tape, magnetic tape to card, and so on. The UNIVAC I and II Systems are examples of such computers with special purpose input/output devices. The UNIVAC III System used as a tape processor in conjunction with a UNIVAC 1050 System as a satellite is an example of such a computer with a satellite.

On such a computer configuration, the example application described at the beginning of this chapter is implemented in the following way. The transaction card deck is converted to tape, either on the special card to tape converter or on the satellite computer. The resulting transaction tape is applied to the master file by the central processor. The output of this processing is an updated master file and another magnetic tape with the information to be printed in the report recorded on it. This report tape is then used to produce the printed report, either on the special magnetic tape driven printer or on the satellite computer. A process chart of this procedure is shown in Figure 13-1.

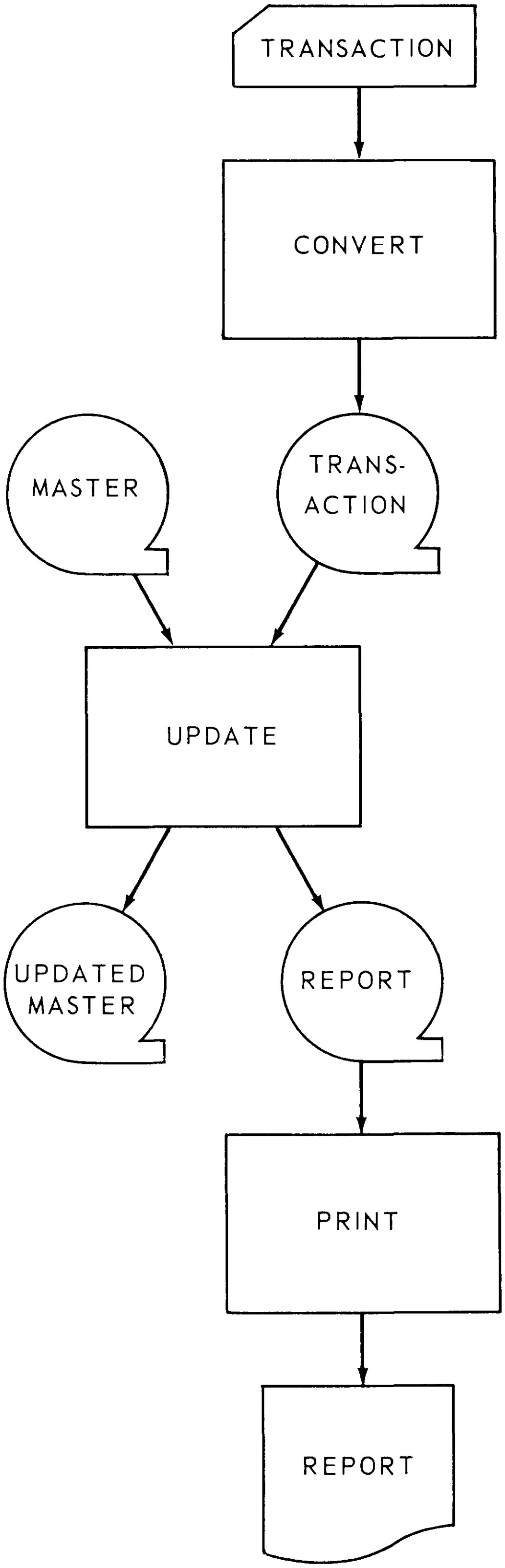


Figure 13-1. Process Chart for a Magnetic Tape Computer Serviced by Offline Peripherals or Satellite Computer

B. NONCONCURRENT COMPUTERS WITH ONLINE PERIPHERALS

Another common computer configuration type is one in which, in addition to having magnetic tape input and output, all the peripherals units, the card reader, the printer, the paper tape reader, the card punch, and so on, are online to the computer. In such a configuration the central processor can read information from magnetic tape and directly from cards, paper tape, documents, and so on. It can write information on magnetic tape, print reports, punch cards and paper tape, and so on, directly. The utilization of such a computer depends on whether the computer has concurrent processing capability. First, consider such a computer that does not have this facility. In such a case, the computer is continuously under the control of a single program. The UNIVAC Solid State System is an example of such a computer.

It is possible to use such a computer to perform the simple file maintenance used as an example here in the same way the computer with offline peripheral equipment is used. In such a case, the process would be as shown in Figure 13-1. In this case, the computer would first be used as a card to tape converter to convert the transaction deck to magnetic tape. The computer would then be used as a magnetic tape computer to update the master file and produce the magnetic tape report file. Finally, the computer would be used as a magnetic tape driven printer to produce the printed report. The procedure results in a three run system. It is not hard to see that this approach is not the best utilization of the equipment.

Another approach to the utilization of a computer with online peripherals is shown in Figure 13-2. Here the transaction file is read into the updating process in card form. The report is produced directly on the printer.

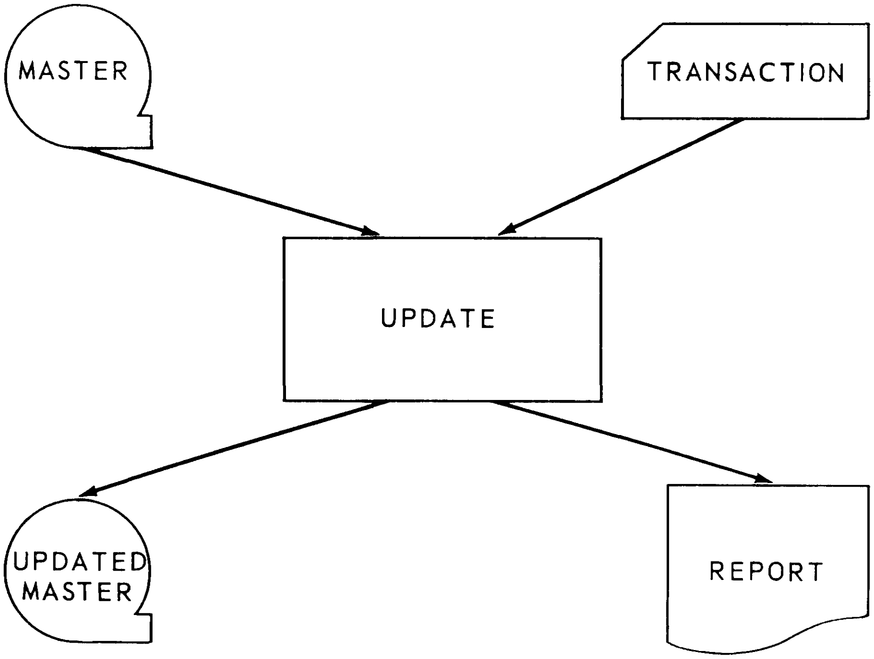


Figure 13-2. Process Chart for a Nonconcurrent Computer with Online Peripherals

Introduction of timing figures will demonstrate the superiority of the second approach over the first. Suppose the speed of a card reader and the volume of the transaction file set the time for reading the card deck at 15 minutes. Suppose the speed of the central processor, the speed of the magnetic tape units and the volume of the master file set the time for updating this file at 5 minutes. Finally, suppose the printer speed together with the volume of the report file set printing time at 20 minutes.

Running time for the process shown in Figure 13-1 is the sum of these times, 40 minutes. If the computer is buffered so that card reading, printing, magnetic tape reading and writing, and internal computer processing can occur simultaneously, running time for the second approach shown in Figure 13-2 is the largest of these times, 20 minutes. If the computer is partially or totally unbuffered, running time for the second approach is greater than 20 minutes but is always smaller than 40. Clearly, the technique exemplified in Figure 13-2 is the appropriate one for a nonconcurrent computer with online peripherals.

Such computers are characteristically medium scale with an average instruction execution time in the range of 200 microseconds. If the card reader of such a computer can read one card in 400 milliseconds, 2000 instructions can be executed in card read time. This is not an unreasonable number of instructions to perform the housekeeping operations associated with the control of the tape handlers, card reader, and printer, to do the processing associated with applying the transaction to the appropriate master item, and to edit the card image for this application and the one or more line images required to produce the prescribed information on a printer report. Hence, such a computer configuration functions as a well balanced system.

However, there are computers with online peripherals whose average instruction execution time is about 14 microseconds or less. The UNIVAC III System is such a computer. With a 400 millisecond per card reader, upwards of 28,000 instructions can be executed in card read time. It would be an unusual application that required this number of instructions to be executed per card. Generally, such a computer would be hopelessly peripheral limited.

Nevertheless, economy of hardware construction legislates for online peripherals. Consequently, to achieve the construction economies associated with online peripherals but to avoid the disutility of a seriously peripheral bound computing system, computers with high internal speeds generally have concurrent processing capability.

C. CONCURRENT PROCESSING COMPUTERS

Typically, concurrent processing computers have more than one program stored in memory, the control of the computer periodically switching from one program to another. To achieve this concurrent processing capability, the computer requires the following features.

■ EXECUTIVE SYSTEM

Some type of executive system is required. It may be hardware, software, or more typically, some combination of the two. This executive system performs several functions.

– The executive system determines which of the several programs in memory is to have computer control. There are two aspects to this control.

1. The executive system must be able to periodically switch control from the program currently being executed and pass this control on to another program. This feature prevents one program from dominating control of the hardware system. For example, this feature prevents a heavily computer bound program from retaining control to a point where the operation of input/output equipment is slowed. This aspect of executive system control may be tied in with the interrupt system, another necessity for concurrent processing which will be noted in further detail below.

2. The executive system must be able to accept control from some program and pass it on to another. This allows a program that is input or output limited to relinquish control when it has completed processing on the items currently in memory and is waiting for more items to be delivered or for the items in the output area to be recorded.

– The executive system generally determines where programs are to be loaded in memory and loads them there. This feature implies that programs are written in such a form that they are relocatable. This ability is generally required on a concurrent processing computer because, at the time of loading a program P into memory, there are usually other programs already loaded in memory. These previously loaded programs are typically not the same from one running of program P to another. Consequently, different portions of memory are occupied from one running of program P to another. Program P must have the ability to be loaded in that part of memory which is available at the time of its loading.

– Similarly, the executive system generally determines what logical peripheral and magnetic tape units are to be assigned to program P for each running. This, in turn, implies that program P must have the ability to address input/output units symbolically. The reason for this necessity is similar to that for relocatability. Program P contains the complement of input/output equipment required for its running, but it generally cannot predict which logical units will be available for assignment at running time. As will be pointed out later, this is not necessarily true. Arrangements can be worked out for fixed input/output assignment to programs.

■ INTERRUPT SYSTEM

A concurrent processing computer requires an interrupt system. An interrupt is a hardware feature that, as the result of the occurrence of some event, causes control to go to some fixed storage location. At the very least, there must be one interrupt that periodically returns control to the executive system so it can cycle control among concurrent programs. More typically, interrupt is supplied whenever an input/output unit completes a cycle.

1. Use of a Concurrent Processing Computer

One reason for not using a concurrent processing computer in the way shown in Figure 13-2 has already been stated. The reason is that such a processing scheme generally results in seriously peripheral-bound programs. On the off chance that a program calling for peripheral as well as tape input/output consumes more computer time than peripheral time, it is still not a good idea to mix tape units with the peripheral units in this program, since the peripheral units are then not operated at top effective speed.

The other reason for not mixing tape units with peripherals in one run has to do with scheduling. If any program can require any array of input/output equipment, it is difficult to achieve a constant program mix on the computer that makes full utilization of the peripheral units. Also, there will probably be many instances when a particular program will not be able to be loaded because one or more of the peripheral units it requires are already in use.

The alternative is to design systems in the manner exemplified in Figure 13-1. That is, application systems for a concurrent processing computer consist of tape to tape runs, programs utilizing only magnetic tape input and output, andof peripheral runs, programs with one magnetic tape input or output and one peripheral unit. Thus, peripheral runs are divided into two types, input peripheral runs andoutput peripheral runs. In an input peripheral run, information is read from some peripheral unit such as a card reader or paper tape reader and is written on magnetic tape. In an output peripheral run, information is read from a magnetic tape and is put out on some peripheral unit such as a printer or card punch.

The program mix on a concurrent processing computer then may consist of one tape to tape run and one or more peripheral runs. For example, the program mix at one point in time might consist of a tape to tape run from the payroll system, a card to tape run to convert transactions for the inventory system, and a tape to printer run to produce a report for the billing system. When the tape to tape program reaches completion, it calls in another tape to tape run as its successor. When the card to tape conversion winds up, its successor is another tape to card conversion. Another printer program succeeds the current one, and so on. In this manner, all peripherals are kept busy; and, with the possible exceptions of adequate memory space or adequate numbers of tape handlers, no program about to be loaded need be delayed because of adequate facilities being unavailable. Also, all peripherals are kept running at maximum speed. A schematic of such concurrent processing is shown in Figure 13-3.

2. Tape Unit Assignment on a Concurrent Processing Computer

That each program will find an adequate number of tape handlers available when it becomes time to be loaded can be guaranteed by installation convention. For example, suppose a configuration of input/output equipment consists of 14 tape handlers, a printer, a card reader, and a card punch. The installation is on single shift; and application demands require that the printer be run a full eight hours, the card reader be run four hours, and the card punch two hours. An instruction tape is required. Then one tape handler can be set aside for the instruction tape, one to drive the printer, and one to be used four hours by the card reader and two hours by the card punch. The installation can then set up as a system design requirement that no tape to tape run be designed to use more than 11 tape handlers. In this fashion, each program will, on loading, always find adequate tape handlers available for it.

Some installations may wish to go further and assign a particular logical tape unit to the instruction tape, to all printer runs, and to all card to tape and tape to card runs. This leaves a specific 11 logical tape handlers available for tape to tape runs. Programs can then address tape handlers directly, and the executive system function of allocating input/output equipment to programs as they are loaded becomes unnecessary.

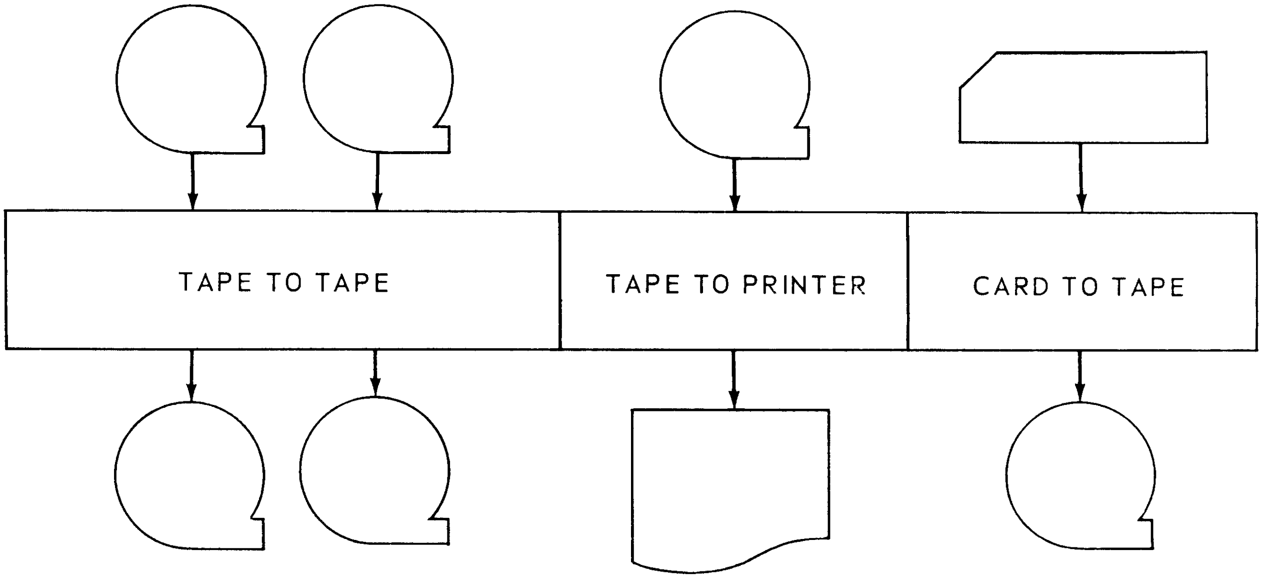


Figure 13-3. Schematic of Concurrent Processing

3. Store Assignment on a Concurrent Processing Computer

Adequate storage space for each program can also be assured by installation convention. In this case, each run type is assigned a maximum amount of store within which it must be designed. For example, if the computer's memory consists of 32,000 locations, 3500 of which are required by the executive system, 2000 storage locations might be assigned to each peripheral run that must run concurrently. This would be 2000 for the printer run and 2000 for the card to tape or tape to card run. Twenty four and a half thousand locations then remain as the upper limit within which all tape to tape runs must be designed.

To make most effective use of store, a minimum amount of store is usually assigned to each peripheral program. This leaves a maximum amount available for tape to tape runs. Such an approach dictates that peripheral runs be limited to little more than straight conversion, all editing being handled by the tape to tape runs. This also provides further assurance that all peripherals will operate at maximum speed.

A characteristic of peripheral runs is that they frequently require restarting. Paper in the printer tears, or a card jam occurs on the card reader. In such instances, the process must be backed up some number of items and restarted. Restarting procedures are simplified if peripheral runs are limited to conversion. This simplification leads to smoother and more standard computer center operating procedures, a desirable system design goal.

Once the store has been partitioned by convention, it becomes possible, if desired, to assign fixed storage locations to each run type. For example, presuming that the executive system pre-empts storage locations 0000-03499, locations 03500-27999 can be assigned to tape to tape runs, 28000-29999 to printer runs, and 30000-31999 to card to tape and tape to card runs. Programs can then be written with fixed storage locations and the program relocatability function of the executive system is eliminated.

4. Instruction Tape Handling on a Concurrent Processing Computer

   Because it usually cannot be predicted which program currently in the computer will end first, some rocking of the instruction tape to locate successor runs seems inevitable. One way to minimize this rocking is to use "wired in" peripheral programs. For example, an installation may have a sole printer program that requires input in a specified format. All tape to tape runs producing tapes for this printer program must produce these tapes in the specified format. The same can be true of the card to tape program and the tape to card program. Then, at the beginning of the shift, the printer and card to tape programs can be read into storage from the front of the instruction tape. These remain in memory and service all printer and card to tape operations. The tape to tape programs can be arranged on the instruction tape in the order in which they are to be run. As a consequence, if the schedule is adhered to, no instruction tape rocking is necessary until the card to tape program is replaced by the tape to card program. After this has been done, the only other event which can cause instruction tape rocking is a change in schedule.

5. Multiple Use of Peripherals on a Concurrent Processing Computer

   On a concurrent processing computer, there is a temptation to use more than one peripheral on a peripheral program. For example, if a computer has two printers, there is the possibility of using both simultaneously to print different reports from information on a single input tape. This temptation should be avoided.

   Peripheral equipment is electromechanical and is subject to more frequent breakdown than electronic equipment. The more units of peripheral equipment that must be up concurrently before a run can be executed, the greater the possibility that the run will be delayed because of equipment failure.

   Moreover, such a run design requires that all peripheral gear involved be free before the run can begin to operate. In the above example, both printers must be free and set up before the run can begin. Both printers will not generally become free at the same time. One printer will have to remain idle until the other becomes free and set up before the run can begin.

   A third reason for avoiding such run design is pertinent when the volumes that the peripheral gear are to handle are disparate. For example, in the two printer run described above, if one printer is to produce a detail report of the input tape while the other is to produce a summary report, the speed of the run is limited by the printer producing the detail report, and the utilization of the other printer during the run is limited.

6. Mixing of Tape Limited and Computer Limited Runs on a Concurrent Processing Computer

   It is sometimes proposed that computer to tape balance can be achieved on a concurrent processing computer by running a tape-limited tape to tape program concurrently with a computer-limited program. While theoretically possible, there are several considerations that militate against such an approach.

   The proposed approach presumes that two or more tape to tape programs of different characteristics are to be run concurrently. This generally requires an increase in the number of tape handlers and in the storage size of the computer required. To justify such an acquisition, the installation must have sufficient computer load to keep several tape to tape runs in the computer concurrently for the greater part of a shift. Even in such cases it may be less expensive to settle for a more modest configuration, only run one tape to tape program at a time, and run into overtime.

The proposed approach also presumes that, whenever a compute limited run is scheduled, there is a tape limited run which can be scheduled concurrently, and vice versa. Such a program mix is exceptional rather than common. Moreover, even if such programs existed, the approach presumes a timing fineness, predictability, and static content in scheduling that is not generally realizable.

7. Use of Concurrent Processing Computers as Conversion Equipment

Suppose an installation with a concurrent processing computer, a number of tape handlers, a printer, a card reader, and some other peripheral equipment. Suppose the installation is on a one shift basis but that the application demand on the printer and card reader is four hours a day each. During the other four hours of the eight hour shift, the printer and card reader can be used in a peripheral run to read cards and print the information read. Such utilization lowers the installation's need for punched card tabulators. Other combinations of peripheral gear are possible: paper tape reading to paper tape or card punching, paper tape to printing, and so on. It should be emphasized that such use of a concurrent processing computer is economical only when there is idle time on the peripherals involved, The use of the computer as a card reader to printer device is usually particularly well balanced, since the document speeds of card readers and printers tend to be similar. Such planned use of the computer still suffers from design deficiencies mentioned earlier and should be approached cautiously. Simultaneity of availability of the peripherals involved is required. This is not only a scheduling problem, but also one of peripheral equipment reliability.

D. CONCURRENT PROCESSING ON THE UNIVAC III

The executive routine in the UTMOST system is EXEC. It pre-empts approximately the first 3500 storage locations for storage and is in store whenever programs are being executed.

The Processor is equipped with an *input/output interrupt.* This interrupt stores the contents of the control counter in storage location 20 and transfers control to the instruction stored in location 21. Roughly speaking, an input/output interrupt occurs every time a block is read from or written on tape, everytime a line is printed on the printer, every time a card is read or punched on the card reader or punch, and, in general, every time an input/output operation occurs on a piece of input/output equipment. Thus, input/output interrupt retrieves control from whatever program is being executed when an input/output operation occurs and gives control to EXEC. By means of a set of indicators, EXEC can determine what input/output operation caused the interrupt and can, if necessary, route control to the proper program for servicing the input/output operation that has occurred.

In accordance with good concurrent processing practice, EXEC allows one tape to tape program to be run concurrently with one or more peripheral programs. In the UTMOST system, peripheral programs are called *symbionts.* Unless a SEG control card is included that directs SUCO to do otherwise, tape to tape programs are loaded in low order storage contiguous to EXEC. Symbionts are loaded into high order storage.

Termination of a tape to tape program can cause initiation of a successor program by means of a NEXT control card. Symbionts are both initiated and terminated by manual action on the console. Thus, a symbiont may be loaded into store and left there to do as many jobs as is required. For example, a tape to printer symbiont may be loaded into store, used to print a tape, and then allowed to remain dormant until another tape is ready to be printed, at which point the symbiont can be reactivated from the console. As pointed out previously, such an approach minimizes instruction tape rocking.

When a symbiont is terminated, the executive routine relocates the remaining symbionts in high order storage. Thus, at all times the maximum amount of store is kept available for storing tape to tape programs between EXEC in low order storage and the symbionts in high order storage. If an installation determines the maximum number of symbionts it plans to run concurrently and the space each such symbiont requires in storage, it can then determine the maximum amount of storage a tape to tape program can require and never run into loading problems because of inadequate store being available for the loading of the program.

In relocating symbionts when a symbiont is terminated, the executive routine makes use of the tape to tape program's DUMP tape. This is why every tape to tape program must specify a DUMP tape even if rerun memory dumps are not to be made. Executive routine use of the DUMP tape in this fashion in no way prejudices the information being written on this tape by the tape to tape program.

As has been demonstrated by the use of tape assignment cards, allocation of UNISERVO tape units in the UTMOST system is essentially a fixed allocation. That is, UNISERVO tape units are addressed logically rather than symbolically. Therefore, it is to the installation's advantage to permanently allocate the number of UNISERVO tape units required to symbionts. EXEC pre-empts UNISERVO tape unit 0 for instruction tape handling. The UNISERVO tape units not used by EXEC or the symbionts can be allocated for use by tape to tape programs.

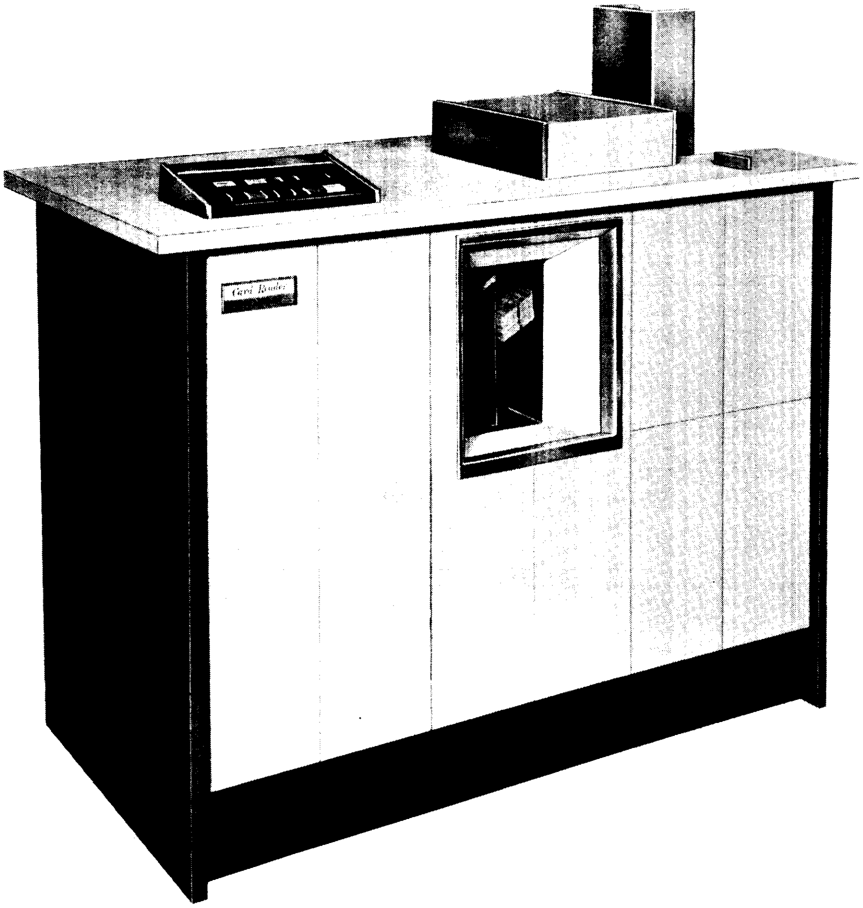
Also in accordance with good concurrent processing principles, the UTMOST systems provides a standard set of symbionts, a tape to card symbiont, a card to tape symbiont, a tape to printer symbiont, and so on. These symbionts minimize editing operations and confine themselves to conversion operations. As a consequence, each input symbiont produces a tape in a specified format. Similarly, each output symbiont expects as input a tape of a specified format. As a consequence, an installation may design and program only tape to tape runs and use the standard symbionts for peripheral operations. When a tape to tape run expects input produced from an input symbiont, say the card to tape symbiont, the format in which the tape is read is the fixed, specified format. Similarly, when a tape to tape run produces output to be used as input to an output symbiont, for example, the tape to printer symbiont, the format in which the tape is to be written is the fixed, specified format. All editing – editing of tapes from input symbionts to increase processing efficiency and editing of tapes for output symbionts – is done in the tape to tape program.

In the remainder of this chapter, the standard card to tape symbiont and the standard tape to printer symbiont are discussed as examples of standard input and output symbionts.

E. THE CARD TO TAPE SYMBIONT

The High Speed Reader is available as either an 80 column model or a 90 column model. Both may be included in one UNIVAC III System. The synchronizer and power supply for the reader are housed within the reader cabinet. Cards are read and checked automatically at the rate of 700 cards per minute. The 80 column reader processes standard Hollerith card code and translates it into the UNIVAC III character code; it can also process any other 80 column card code. The 90 column reader processes 90 column Remington Rand card code and translates it into the UNIVAC III character code; it can also process any other 90 column card code.

The card feed path of the reader (Figure 13-4) includes a card input magazine, four card stations, and three output stackers. In Figure 13-4, cards are shown at the start of an operating cycle. The card stations are numbered 1 through 4, and the stackers are designated 0, 1, and 2. Card station 1 is the first read station. Card station 2 is the second read station; card images from from this station are transferred to memory. Cards are transported by means of continuously moving rollers which advance cards from the input magazine, through the two read stations, to card station 3, card station 4, and finally to one of the three output stackers.



The cards to be read are moved into the card path by the picker knife, which is program controlled. At the first read station, the card is brush sensed, and a hole count is stored for checking purposes. After the card is read at the second read station, hole counts from the two read stations are compared. If an error is detected, the program testable data error indicator is set and automatic program interrupt occurs.

Cards to be read are stored in the input magazine, which holds 2000 cards. Excellent card feed reliability is achieved through the use of a vacuum which helps to position the card to be engaged by the picker knife. When the input magazine is empty or a misfeed occurs, the unit stops, the MISFEED light of the reader control panel lights, the program testable operator oversight indicator is set, and automatic program interrupt occurs.

Each of the three output stackers holds 1000 cards. When a stacker is full, the STACK FULL light on the reader control panel lights, the program testable operator oversight indicator is set, and automatic program interrupt occurs. All cards enter stacker 0 unless the program specifies stacker 1 or 2.

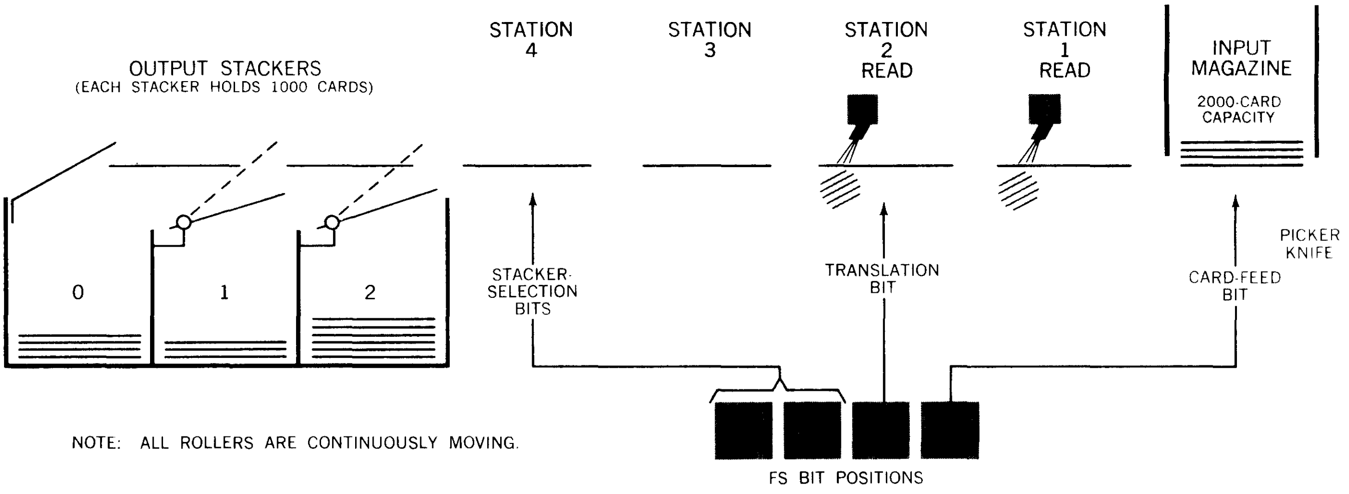


Figure 13-4. Card-Feed Path, High-Speed Reader

Cards may be read either with or without automatic translation from the card code to the UNIVAC III character code. Figures 13-5 and 13-6 illustrate the data flow for 80 column cards, and Figure 13-7 illustrates the data flow for 90 column cards.

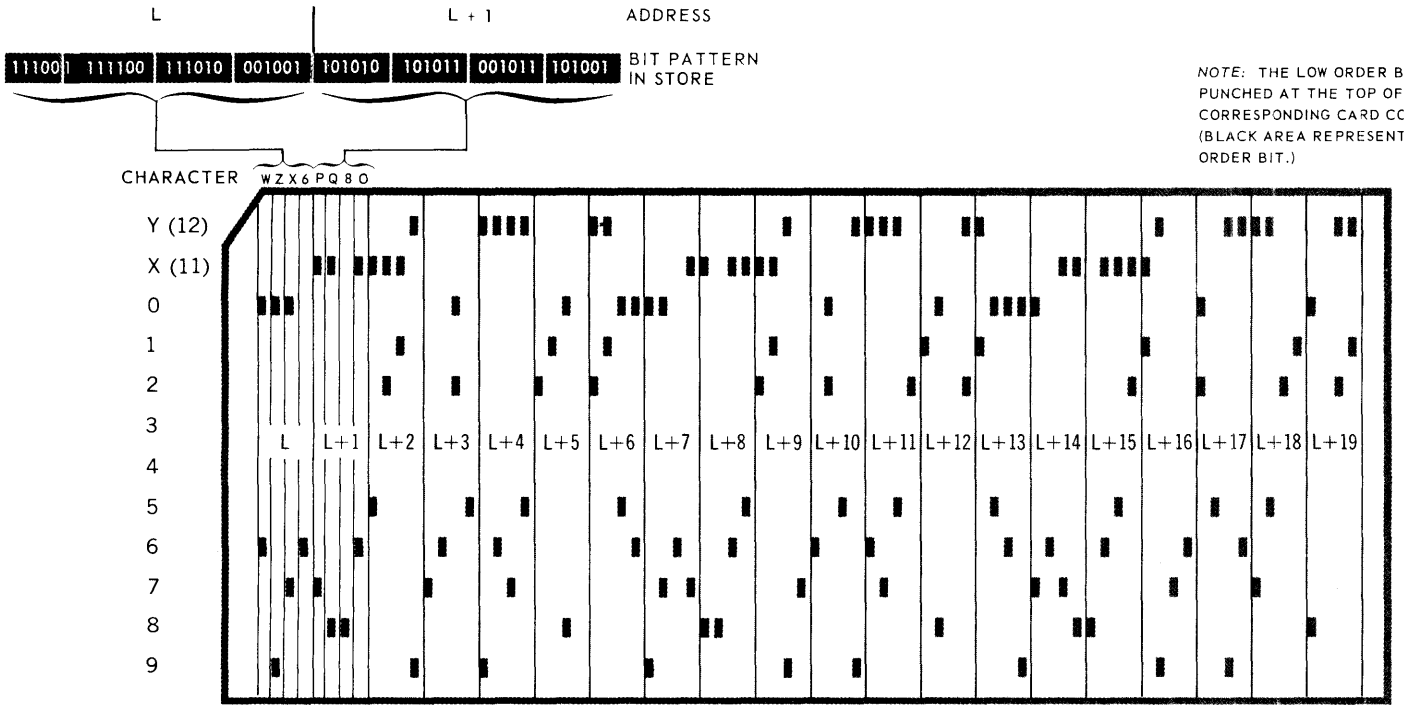
When an 80 column Hollerith code card is automatically translated, the card image occupies 20 UNIVAC III alphanumeric words in store, as indicated in Figure 13-5; an 80 column card image transferred without translation occupies 40 alphanumeric words. The first untranslated 24 bit word is represented by a card field in the upper left portion of the card, four columns wide by six rows deep, as shown in Figure 13-6. Because cards are read without translation, non-Hollerith codes may be used. For both Hollerith andnon Hollerith codes, a 0 bit is placed in each sign bit position in store when a card is read.

When a 90 column card image is transferred to store, either with or without translation, it occupies 24 alphanumeric words, as shown in Figure 13-7. Binary 0's are inserted in the three least significant character positions of the 12th word and of the 24th word of the card image in store. A 0 bit is placed in each sign bitposition in store when a card is read.

Hollerith and Remington Rand 90 column card codes are given in Figures 13-8 and 13-9, respectively.

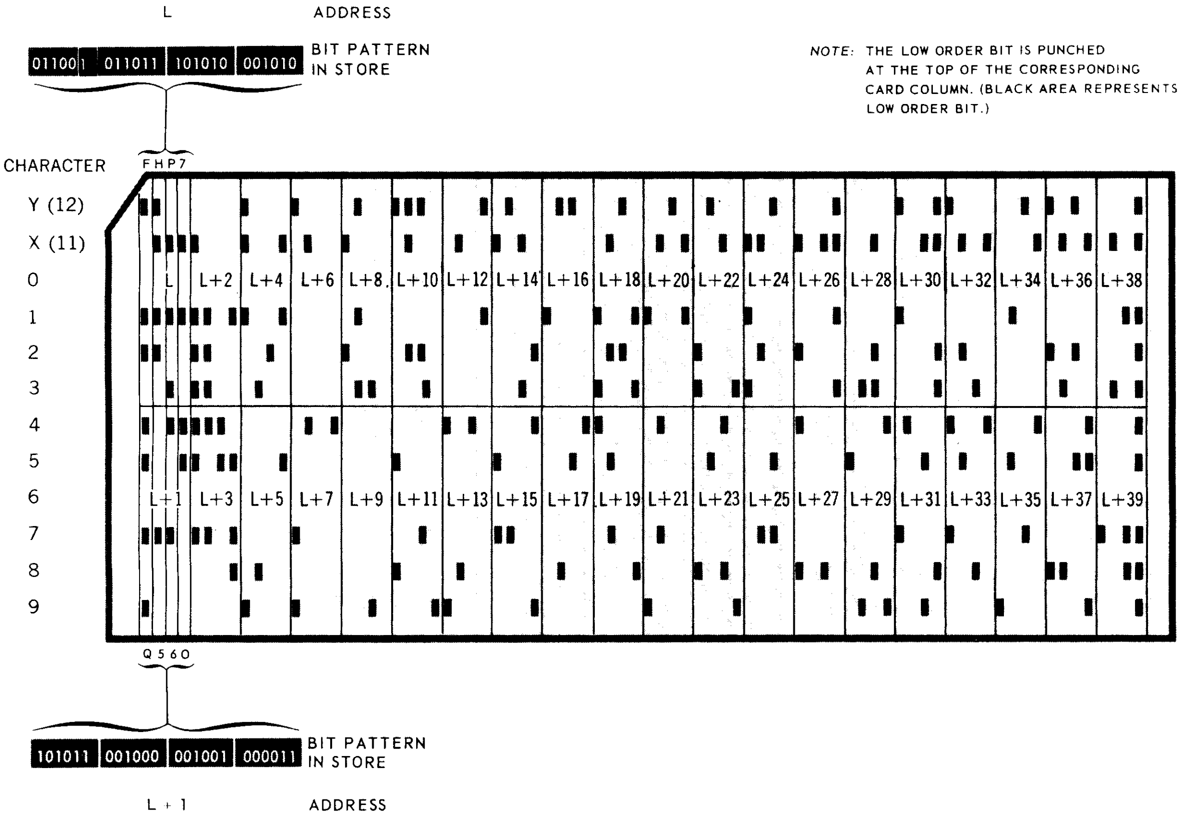
After the synchronizer has completed transferring the data from a card, the number of storage accesses is checked to verify that an entire card image has been transferred to store. Only the image from the second read station is transmitted to store. Thehole counts from the two read stations are compared. A modulo 3 check is made on each word that is transferred. If any of these checks detects an error, a program testable indicator is set, and an automatic program interrupt occurs.

The standard card to tape symbiont described below is for 80 column cards.



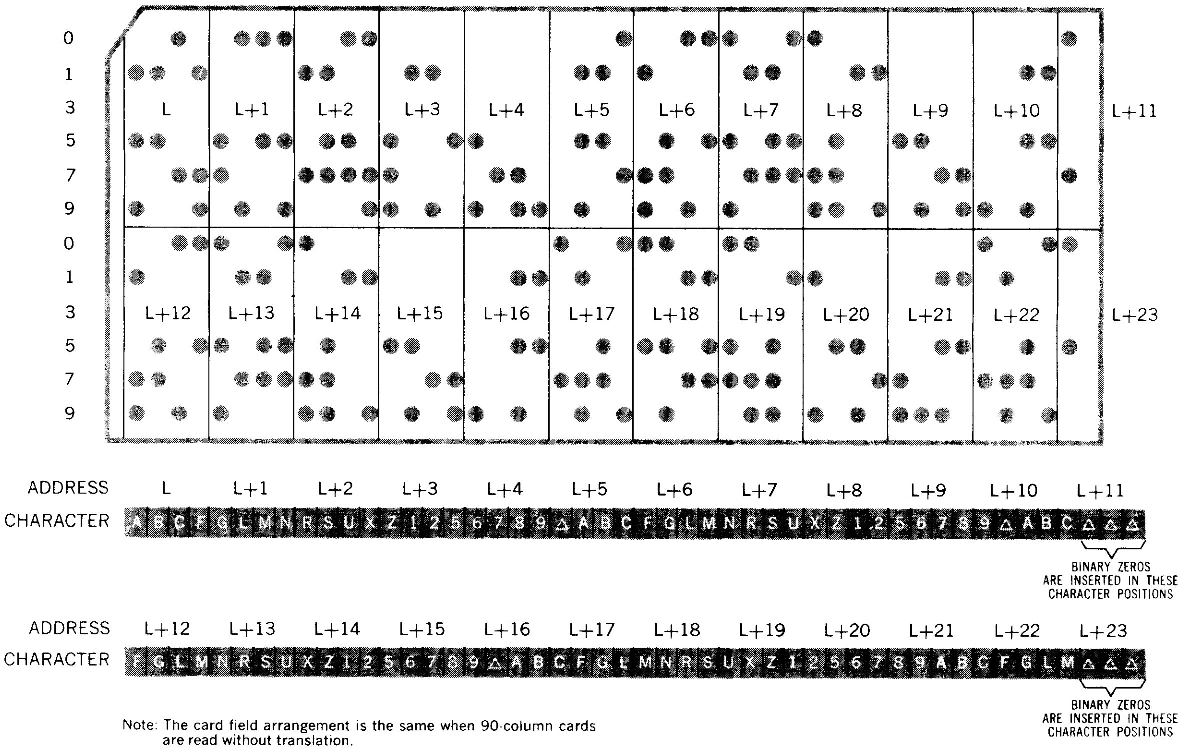
NOTE: THE LOW ORDER BIT IS PUNCHED AT THE TOP OF THE CORRESPONDING CARD COLUMN. (BLACK AREA REPRESENTS LOW ORDER BIT.)

Figure 13-5. Data Trasfer from Reader to Memory, with Translation, 80-Column Card


L
ADDRESS
011001 011011 101010 001010
BIT PATTERN IN STORE
CHARACTER
F H P 7
Y (12)
X (11)
0
1
2
3
4
5
6
7
8
9
L L+2 L+4 L+6 L+8 L+10 L+12 L+14 L+16 L+18 L+20 L+22 L+24 L+26 L+28 L+30 L+32 L+34 L+36 L+38
L+1 L+3 L+5 L+7 L+9 L+11 L+13 L+15 L+17 L+19 L+21 L+23 L+25 L+27 L+29 L+31 L+33 L+35 L+37 L+39
Q 5 6 0
101011 001000 001001 000011
BIT PATTERN IN STORE
L + 1
ADDRESS

NOTE: THE LOW ORDER BIT IS PUNCHED AT THE TOP OF THE CORRESPONDING CARD COLUMN. (BLACK AREA REPRESENTS LOW ORDER BIT.)

Figure 13-6. Data Transfer from Reader to Store, without Translation, 80-Column Card


0
1
3
5
7
9
0
1
3
5
7
9
L L+1 L+2 L+3 L+4 L+5 L+6 L+7 L+8 L+9 L+10 L+11
L+12 L+13 L+14 L+15 L+16 L+17 L+18 L+19 L+20 L+21 L+22 L+23
ADDRESS L L+1 L+2 L+3 L+4 L+5 L+6 L+7 L+8 L+9 L+10 L+11
CHARACTER A B C F G L M N R S U X Z 1 2 5 6 7 8 9 △ A B C F G L M N R S U X Z 1 2 5 6 7 8 9 △ A B C △ △ △
BINARY ZEROS ARE INSERTED IN THESE CHARACTER POSITIONS
ADDRESS L+12 L+13 L+14 L+15 L+16 L+17 L+18 L+19 L+20 L+21 L+22 L+23
CHARACTER F G L M N R S U X Z 1 2 5 6 7 8 9 △ A B C F G L M N R S U X Z 1 2 5 6 7 8 9 A B C F G L M △ △ △
BINARY ZEROS ARE INSERTED IN THESE CHARACTER POSITIONS

Note: The card field arrangement is the same when 90-column cards are read without translation.

Figure 13-7. Data Transfer from Reader to Store, with Translation, 90-Column Card

The upper entry represents the card punching positions.
The lower entry represents the corresponding High-Speed Printer character.
NP indicates a code which is not printed by the High-Speed Printer.
NS indicates nonstandard codes.

Numeric Bits	Zone Bits			
	00	01	10	11
0000	Blank Space	12 +	NP, NS	NP, NS
0001	1 4 8 ;	12 4 8)	11 4 8 *	0 4 8 (
0010	11 —	12 3 8 .	11 3 8 $	0 3 8 , (Comma)
0011	0 0	NP	NP	4 8 ' (Apostrophe)
0100	1 1	12 1 A	11 1 J	0 1 /
0101	2 2	12 2 B	11 2 K	0 2 S
0110	3 3	12 3 C	11 3 L	0 3 T
0111	4 4	12 4 D	11 4 M	0 4 U
1000	5 5	12 5 E	11 5 N	0 5 V
1001	6 6	12 6 F	11 6 O	0 6 W
1010	7 7	12 7 G	11 7 P	0 7 X
1011	8 8	12 8 H	11 8 Q	0 8 Y
1100	9 9	12 9 I	11 9 R	0 9 Z
1101	4 6 8 :	3 8 =	NP, NS	NP, NS
1110	4 5 8 <	NP, NS	NP	NP
1111	3 5 8 >	NP, NS	NP, NS	NP, NS

Figure 13-8. Hollerith Code, High-Speed Reader

The upper entry represents the card punching positions.
The lower entry represents the corresponding High-Speed Printer character.
NP indicates a code which is not printed by the High-Speed Printer.

Numeric Bits	Zone Bits			
	00	01	10	11
0000	Blank Space	0 1 3 5 7 +	0 1 5 7 9 NP	0 1 7 9 NP
0001	1 3 5 7 ;	1 3 7 9)	0 1 *	0 1 5 (
0010	0 3 5 7 —	1 3 5 9 .	0 1 3 5 9 $	0 3 5 9 , (Comma)
0011	0 0	0 1 3 NP	0 3 7 9 NP	1 5 7 9 ' (Apostrophe)
0100	1 1	1 5 9 A	1 3 5 J	3 5 7 9 /
0101	1 9 2	1 5 B	3 5 9 K	1 5 7 S
0110	3 3	0 7 C	0 9 L	3 7 9 T
0111	3 9 4	0 3 5 D	0 5 M	0 5 7 U
1000	5 5	0 3 E	0 5 9 N	0 3 9 V
1001	5 9 6	1 7 9 F	1 3 O	0 3 7 W
1010	7 7	5 7 G	1 3 7 P	0 7 9 X
1011	7 9 8	3 7 H	3 5 7 Q	1 3 9 Y
1100	9 9	3 5 I	1 7 R	5 7 9 Z
1101	0 1 3 7 9 :	0 1 5 7 =	0 1 9 NP	0 1 3 9 NP
1110	1 3 5 7 9 <	0 1 5 9 NP	0 1 3 7 NP	0 3 5 7 9 NP
1111	0 5 7 9 >	0 1 3 5 7 9 NP	0 1 7 NP	0 1 3 5 NP

Figure 13-9. 90-Column Card Code, High-Speed Reader

The card deck for the standard card to tape symbiont should be formed as follows. The first four cards should be blank cards. The fifth card should be a *label card,* which has the following format:

Column 1: A control punch of 12-0-2.

Column 2: Blank

Columns 3-7: LABEL

Column 8: Blank

Columns 9-16: In these columns should be punched the eight character file identification desired in the tape label block.

Column 17: Blank

Columns 18-23: In these columns should be punched the date of cycle desired in the tape label block.

Column 24: Blank

Columns 25, 26: In these columns should be punched the number of card images desired to make up one data block on tape. The maximum number of images that can make up one block is 25. If these columns are left blank, a block size of 25 images is used.

Column 27: Blank

Columns 28, 29: In these columns should be punched the reel number desired in the tape label block. If these columns are left blank, the reel number in the tape label block is set to 1.

Following the label card should be the data cards. Following the data cards should be an *end of file card.* An end of file card should have a 12-0-2 control punch in column 1 and ENDΔOFΔFILE in columns 2-12. Following the end of file card should be four blank cards.

Periodically through the card deck should appear *restart cards.* A restart card should have a 12-0-2 control punch in column 1, RESTART in columns 2-8, and columns 9-12 should be blank. The function of restart cards is described later in this section.

The card to tape symbiont is loaded into store by means of a manual operation at the console. Once loaded, the symbiont remains dormant until activated. The symbiont is activated by typing in ScΔSTART, where c is the number of the general purpose channel by which the card reader is connected to the Processor. (The Processor has eight general purpose channels. Any piece of peripheral equipment may be connected to the Processor through any of the eight general purpose channels.) Suppose for purposes of this section that the card reader uses general purpose channel 5. In this case, the form of the above typein is:

S5ΔSTART

Once the symbiont is activated, it begins reading cards and writing tape. The symbiont is designed to write tape on the UNISERVO whose number is specified to be file number 13. It writes a standard label block containing the information specified in the label card. It then writes data blocks of the size specified in the label card. Each card is read in a translated mode and is written as a 20 word item on tape. This item is a card image as exemplified in Figure 13-5.

The symbiont will stack cards in stackers 1 and 2. It stacks 500 cards in one stacker and then switches to the other.

When the symbiont detects a restart card in the input deck, it writes whatever previously unrecorded card images it is holding in store on tape as a block, regardless of whether the block is full size. It then writes four *bypass sentinel blocks*. Normal card to tape conversion then resumes.

A bypass sentinel block is recognized as such by the input routine which will subsequently read the tape being produced by the symbiont. It will, in fact, "bypass" these blocks and not deliver them to the program for processing. Consequently, the existence of bypass sentinel blocks on a tape has no effect on the processing of the tape. What the symbiont uses the bypass sentinel blocks for is described later in this section.

When the symbiont detects an end of file card, it writes whatever card images remain in store on tape, writes end of file sentinel blocks, and rewinds the tape. The message RDR EOF is typed on the console, and the symbiont then becomes dormant. A new card deck can be placed in the reader, a new blank tape can be mounted on the output UNISERVO, and a new conversion can be initiated by an S5ΔSTART typein at any subsequent time.

If a fault or error occurs during conversion, a message to this effect is typed on the console, conversion ceases, any cards already committed to the reader but not properly read by the symbiont are selected into stacker zero, and the symbiont becomes dormant. All cards up to and including the last restart card should then be removed from the stacker and placed at the bottom of the deck in the input magazine. S5ΔGO is then typed in at the console. The symbiont repositions the output tape to the last bypass sentinel blocks written on the tape and recommences the conversion.

If, during restart, the symbiont does not find a restart card at the head of the deck in the input magazine, it types out NO RESTART and takes the same action as if a fault or error had occurred. The same recovery action as used for faults and errors should be instituted.

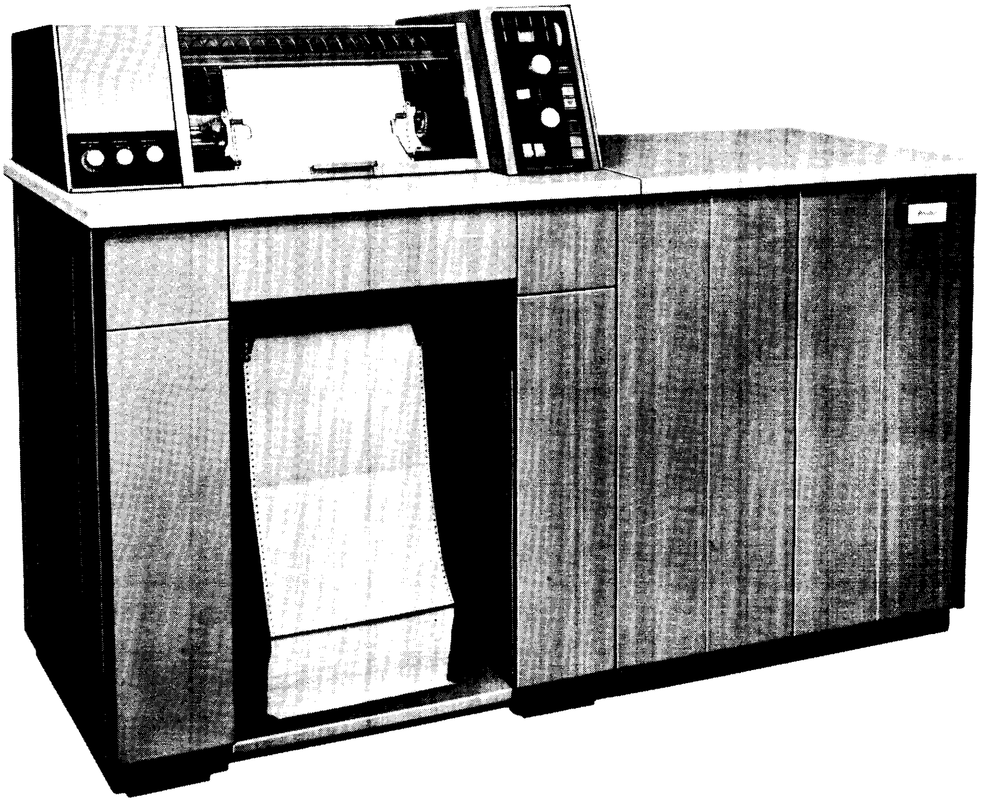
If, during conversion, the symbiont detects the end of tape warning window on the output tape, it writes end of reel sentinels after the last full data block, rewinds the tape, and becomes dormant. The output tape should be dismounted, a new blank tape mounted, and S5ΔGO typed in at the console. The symbiont then writes a label block on the new tape and continues conversion.

The standard card to tape symbiont requires a little more than 1500 storage locations.

F. THE TAPE TO PRINTER SYMBIONT

Under control of the central processor program, the High Speed Printer produces documents at the rate of 700 lines per minute for alphanumeric data and 922 lines per minute for numeric data. Lines are composed of 128 characters. In addition to the original, up to five carbon copies may be produced.

Data flows from the Central Processor to the printer synchronizer through one of the eight general purpose channels. The synchronizer controls printing of the data. To assure that the printer operates at full capacity without delaying the operation of the Central Processor, the automatic program interrupt feature is used.



The High Speed Printer consists of a printer cabinet and an adjoining synchronizer. The printer cabinet contains a continuously rotating type drum with 128 printing positions, 128 print hammers which correspond to the printing positions, a self reversing ribbon feed mechanism, and a paper drive mechanism. The synchronizer contains the circuitry that controls data transfers, paper advance, and printing.

Along the length of the type drum (Figure 13-10) are 128 bands of printing characters. Each band contains the 51 character print set around the circumference of the drum.

Characters are arranged on the drum in a checkerboard pattern so that they are separated from characters on adjacent bands by approximately ⅛ inch; this space reduces the possibility of smudging by characters on a band adjacent to the one being printed.

Two sets of sprocketed tractors – an upper set and a lower set – advance the continuous paper through the printer under program control. While the paper is being printed, the two sets of tractors maintain paper tension.

Each of the four tractors is equipped with a tractor locking lever. These levers are pushed in before the tractors are adjusted; after tractor adjustment, the locking levers are pulled out, thus locking the tractors to prevent any further lateral motion.

Blank or preprinted paper from 4 to 22 inches wide and up to 22 inches long between folds can be used with the printer. The original document and up to five carbon copies may be printed, using paper between approximately 11 and 13.5 pounds in weight, up to a pack thickness of approximately 15.5 mils; this includes card stock. Vertical spacing, which may be set at the control panel by the operator, is either 6 or 8 lines per inch; horizontal spacing is 10 characters per inch. When only 2½ inches of paper remains below the print hammers, a signal from the High Speed Printer alerts the Central Processor and automatic program interrupt occurs. The Central Processor also is alerted and paper movement stops if paper has advanced continuously for more than 1.5 seconds.

Thirty-two words from consecutive storage locations are transferred to the printer synchronizer and modulo 3 checked. They remain in the synchronizer until they are printed according to the printable character code; sign bits are ignored. The program being executed controls paper advance and printing only. If editing of the final printed page is desired, it is accomplished within the 32 consecutive storage locations by the internal program before the order for printing is given.

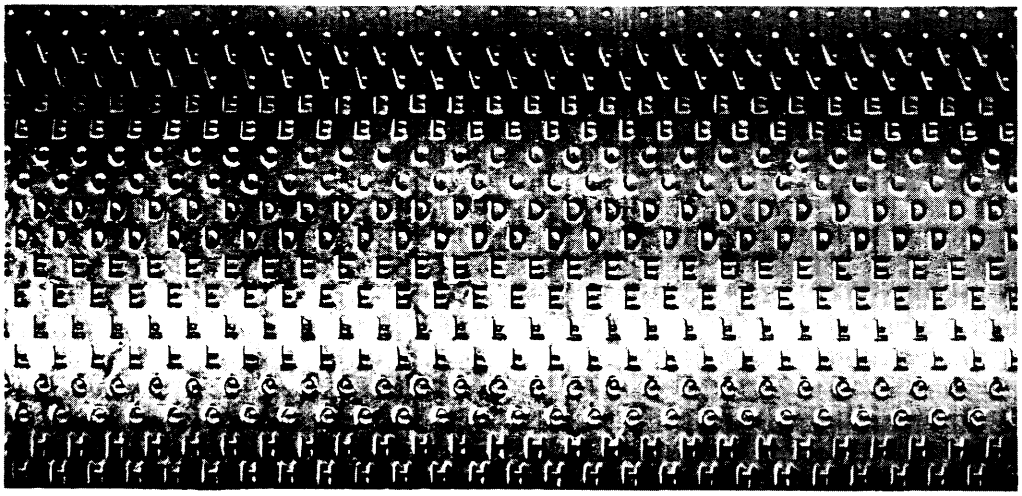


Figure 13-10. Type Drum, High-Speed Printer, Front View

After the 32 words are transferred to the synchronizer, each character in the 32 words is printed in a sequence governed by the order of the characters on the type drum. The determination of which characters are to be printed next is made in the following way:

1. *As the type drum rotates, the printer synchronizer keeps track of which row of characters is in printing position.*

2. *The character in printing position is compared with the characters stored in the synchronizer.*

3. *When the character on the drum in printing position matches the characters in the synchronizer, the appropriate print hammers are actuated to drive the paper and ribbon against the type drum, thereby printing the desired characters onto the paper.*

Single spaced numeric information can be printed at the rate of 922 lines per minute; single spaced alphanumeric information can be printed at the rate of 700 lines per minute. Timing of the paper advance depends on the number of lines advanced and on whether the line spacing is 6 or 8 lines per inch. For spacing of either 6 or 8 lines per inch, 10 milliseconds are required to advance the first line of paper; each additional line requires 8.3 milliseconds if the spacing is 6 lines per inch or 6.25 milliseconds if the spacing is 8 lines per inch. After the paper is advanced, 10 milliseconds are required to stabilize the paper before actual printing begins. When all the characters stored in the synchronizer have been printed, the line is complete and interrupt occurs if it was specified. The printed line per minute rate depends on the required paper advance and on the group of characters to be printed.

The printer can be acquired with either of two sets of printable characters. The COBOL – FORTRAN set is shown in Figure 13-11, the UNIVAC III Standard Set in Figure 13-12.

NP indicates a code which is not printed by the High-Speed Printer.

Numeric Bits	Zone Bits			
	00	01	10	11
0000	Space	+	NP	NP
0001	;	)	*	(
0010	–	.	$	Comma ,
0011	0	NP	NP	' Apostrophe
0100	1	A	J	/
0101	2	B	K	S
0110	3	C	L	T
0111	4	D	M	U
1000	5	E	N	V
1001	6	F	O	W
1010	7	G	P	X
1011	8	H	Q	Y
1100	9	I	R	Z
1101	:	=	NP	NP
1110	<	NP	NP	NP
1111	>	NP	NP	NP

Figure 13-11. COBOL - FORTRAN Set

NP indicates a code which is not printed by the High-Speed Printer.

Numeric Bits	Zone Bits			
	00	01	10	11
0000	△	&	NP	NP
0001	)	:	*	%
0010	–	.	$	Apostrophe
0011	0	NP	NP	+
0100	1	A	J	/
0101	2	B	K	S
0110	3	C	L	T
0111	4	D	M	U
1000	5	E	N	V
1001	6	F	O	W
1010	7	G	P	X
1011	8	H	Q	Y
1100	9	I	R	Z
1101	Comma	#	NP	NP
1110	;	NP	NP	NP
1111	(	NP	NP	NP

Figure 13-12. UNIVAC III Standard Set

The standard tape to printer symbiont accepts as input a tape with a standard label block and is terminated by either end of reel or end of file sentinels. Data blocks on the tape may be of any size up to and including 254 data words. Item size on the tape may be from 2 through 33 words, but all items on the tape must be the same size. The tape to printer symbiont is designed to read this tape from the UNISERVO tape unit whose number is specified by a typein.

The symbiont assumes that it is printing on forms capable of having 66 lines printed on one form and that it will leave a blank six line *heading* and a blank six line *footing* on each form. These assumptions can be modified by means to be described later in this section.

Assume a tape of 33 word items. Also assume that the symbiont is ready to print a line on a form at some point between the heading and footing. The current 33 word item contains the information to be printed. The zero word of the item is a control word that tells the symbiont how the line is to be printed. Bit positions 9-15 and 25 of the control word must contain a zero. Bits 1-7 of the control word specify to the symbiont how many lines the symbiont is to advance the form in the printer before the item is printed. The specification is made in binary. The last 32 words of the item make up a *line image*, which is printed after the form advance. Thus, the first word of the item is printed in print positions 1-4, the second word of the item in print positions 5-8, the third word in print positions 9-12, and so on.

Bit 16-21 of the control word specify in binary the length of the line image in words. Thus, for a 33 word item, bits 16-21 of the control word contain a binary 32. If the line image is smaller than 32 words (amdconsequently, the item is smaller than 33 words), the line image will be *left justified* in printing. Thus, the first word of the item will be printed in print positions 1-4, and so on. Print positions to the right, not accounted for in the line image, will have spaces printed in them automatically by the symbiont.

If form advance causes advance into the footing of a form, the form advance specified in the control word is ignored, and the line image is printed on the *head line* of the next form, which is the line immediately below the heading of the form.

To obtain normal form advance, bit position eight of the control word must contain a zero. If it contains a one, the paper in the printer is advanced to the next form. The line image associated with such a control word is printed on the line of the new form whose number is specified in bits 1-7 of the control word. Lines on a form are numbered from 1 and are counted starting with the head line.

Bits 22-24 of a control word are immaterial.

If it is desired to change the symbiont's assumptions concerning form, heading, and footing length, the first item on tape should have a control word of the following format:

Bit 25:	One
Bits 22-24:	Immaterial
Bits 15-21:	Number of lines in heading expressed in binary.
Bits 8-14:	Number of lines in footing expressed in binary.
Bits 1-7:	Number of lines on form expressed in binary.

The contents of the rest of the item is immaterial. Once the Symbiont's assumptions concerning heading, footing, and form length have been changed by the above means, these revised assumptions are retained by the symbiont until either another change is made or the symbiont is reloaded from the instruction tape.

Punch and print images may be intermixed on the tape. The tape to printer symbiont will bypass all items constituting punch images.

The symbiont is loaded into store by means of a manual operation at the console. Once loaded, the symbiont remains dormant until activated. The symbiont may be activated and directed to print any one of a number of "stacked files" on the input tape by typing in

ScΔSTARTΔAAAAAAAA,

where c is the channel number of the general purpose channel by which the printer is connected to the Processor, and AAAAAAAA is the label ID of the file to be printed. Suppose for purposes of this section that the high speed printer uses general purpose channel 6. In this case, the form of the typein is

S6ΔSTARTΔAAAAAAAA.

If the file label ID is omitted from the above typein, the symbiont will print the next file physically on tape.

The typein S6ΔSTARTΔNNNN acts as an S6ΔSTART typein, except that it will not print the first NNNN pages.

None of the above typeins rewinds the tape, but rewind can be initiated by typing in

S6ΔRW.

When the symbiont is activated, it prints the file ID, date, and reel number of the label (if present) on the console typewriter, and it prints the first line on the printer. The symbiont then becomes dormant. The single line of printing (which may be a test pattern) can be used to position the paper in the printer. This line can be reprinted as many times as is desired by typing in S6ΔTEST.

Once the paper has been positioned, normal printing can be started by typing in

S6ΔGO.

If a form is spoiled because of a tearing of the form or carbon or because of a printer malfunction, such as a blown fuse, the symbiont can be made dormant by typing in

S6ΔRELEASE

Then, by typing in

S6ΔBACKΔn

the symbiont can be reactivated. At this point the symbiont will read the input tape backward enough blocks to reprint approximately the number of forms specified in the typein by n ($1 \leq n \leq 9$). The symbiont will then print one line and release. If necessary, the paper can then be repositioned. Normal printing resumes as a result of a typein of

S6ΔGO

When the symbiont detects end of reel or end of file sentinels on the input tape, it types out END PRINTING and becomes dormant. A different file, either on the current tape or on a newly mounted tape, can then be printed by means of an activating typein.

The tape to printer symbiont requires approximately 1200 storage locations.

UNIVAC III UTMOST
SECTION 17-F, UP-3853

FASTRAND

SUBSYSTEM

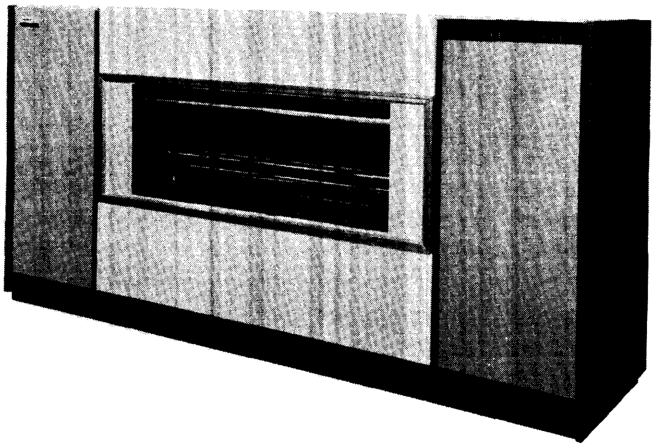


Figure 17-F-1. UNIVAC III FASTRAND Mass Storage Unit

1. UNIVAC III FASTRAND MASS STORAGE SUBSYSTEM

The UNIVAC FASTRAND* Mass Storage Subsystem *(Figure 17-F-1)* provides the UNIVAC III Data Processing System with random access external storage capability. Each subsystem is composed of from one to eight storage units linked to the central processor through a control unit and synchronizer. The control unit and synchronizer are housed in a single cabinet. Each Mass Storage (drum) Unit requires its own storage cabinet and power supply. The storage capacity of a full size subsystem is 528,482,304 six-bit alphanumeric characters. The capacity of a subsystem with one Mass Storage (drum) Unit attached is 66,060,288 characters.

Data is recorded around the surface of the drum cylinders in a bit serial format. The basic unit of drum storage is the sector which contains either 42 or 37 UNIVAC III words depending upon the recording mode used. 64 sectors per track are accessed serially as the drum rotates. There are 64 read/write heads located along the length of the drum, allowing the accessing of 64 tracks without changing the position of the read/write heads. The read/write heads are moveable and can be positioned over any one of 96 separate tracks. (See *Figure 17-F-2* for conceptual presentation).

Each FASTRAND Mass Storage Drum Unit contains two cylinders which revolve at the rate of 880 RPM. From the programmer's point of view, these two cylinders can be regarded as a single drum. The UNIVAC III programmer deals solely with the logical relationship of the various tracks and sectors, without regard to their physical location. Operation of the unit is entirely under program control.

The functions of the FASTRAND subsystem, like all other input/output components of the UNIVAC III Computer, are performed under its own control. The central processor action is merely to initiate the FASTRAND function request, after which it is free to perform operations or calculations as directed by the operating programs. The FASTRAND control unit controls the execution of all requested functions, automatically interrupting the central processor when a requested function has been completed. The capacity of a single drum unit and the various access factors are summarized in *Table 17-F-1.*

* *Trademark of Sperry Rand Corporation*

	PER UNIT	UNIVAC III WORDS	ALPHANUMERIC CHARACTERS	DECIMAL DIGITS	OCTAL DIGITS
DATA CAPACITY – COMPRESSED MODE	DRUM	16,515,072	66,060,288	99,090,432	132,120,576
	POSITION	172,032	688,128	1,032,192	1,376,256
	TRACK	2,688	10,752	16,128	21,504
	SECTOR	42	168	252	336
DATA CAPACITY – NORMAL MODE	DRUM	14,548,992	58,195,968	87,293,952	116,391,936
	POSITION	151,552	606,208	909,312	1,212,416
	TRACK	2,368	9,472	14,208	18,944
	SECTOR	37	148	222	296

	FUNCTION	MAXIMUM	MINIMUM	MEAN
ACCESS FACTORS (excluding programming requirements)	POSITION HEAD BAR	86 milliseconds	30 milliseconds	57 milliseconds
	SWITCH HEAD	20 microseconds	20 microseconds	20 microseconds
	LOCATE SECTOR	70 milliseconds	0 milliseconds	35 milliseconds
	PROCESS SECTOR	1.09 milliseconds	1.09 milliseconds	1.09 milliseconds

Table 17-F-1. FASTRAND Capacity and Access Time Chart

The FASTRAND subsystem is accessible to all programs sharing the computer. Access to the FASTRAND subsystem is controlled by the Executive Routine which monitors the execution of the various requests. FASTRAND functions are performed in the sequence in which they are forwarded to the Executive Routine. A successful completion signal is available for interrogation by the requesting program to determine that its request has been completed. This feature enables the FASTRAND subsystem to transfer information between memory and the Mass Storage (drum) Unit at full speed in parallel with the operation of the central processor and other input/output equipment.

Normally, a system shall contain one UNIVAC III FASTRAND synchronizer control unit. It is always attached to General Purpose Channel One. FASTRAND functions are executed serially in the order in which they are received. One order, the positioning of the read/write bar, once initiated for one of the drum units, can be executed concurrently with the execution of an instruction affecting another drum.

All words transferred between the central processor and the synchronizer are checked for Mod-3 errors. Data read from the drum to the synchronizer in normal mode receives a Mod-3 check. The transfer of data in the compressed mode is not Mod-3 checked between the drum and the synchronizer, but Mod-3 parity is formed by the synchronizer before data is transferred to memory. A Read Check instruction can be employed following write instructions when it is desired to insure the accuracy of recording. Data is not transferred to the central processor during read checking.

a. Drum Unit Characteristics

The drum units of the FASTRAND subsystem are each housed in their own cabinet. Each drum cabinet contains two 24 inch diameter cylinders mounted one above the other. The 64 read/write heads associated with the drum are attached to a metal bar so that 32 heads service each cylinder. The bar moves laterally to bring the heads over one of 96 possible positions. The heads are fixed to the bar and are simultaneously shifted to the same relative track positions within their 96 track range.

Each track is subdivided into 64 addressable sectors. The capacity of each of these sectors is 37 or 42 UNIVAC III words, depending upon the read/write mode specified. The rotation of the drum cylinders is 880 RPM bringing each sector under the read/write head once every 70 ms. The average access (latency) time for any sector of a track over which a head has been positioned is 35 ms.

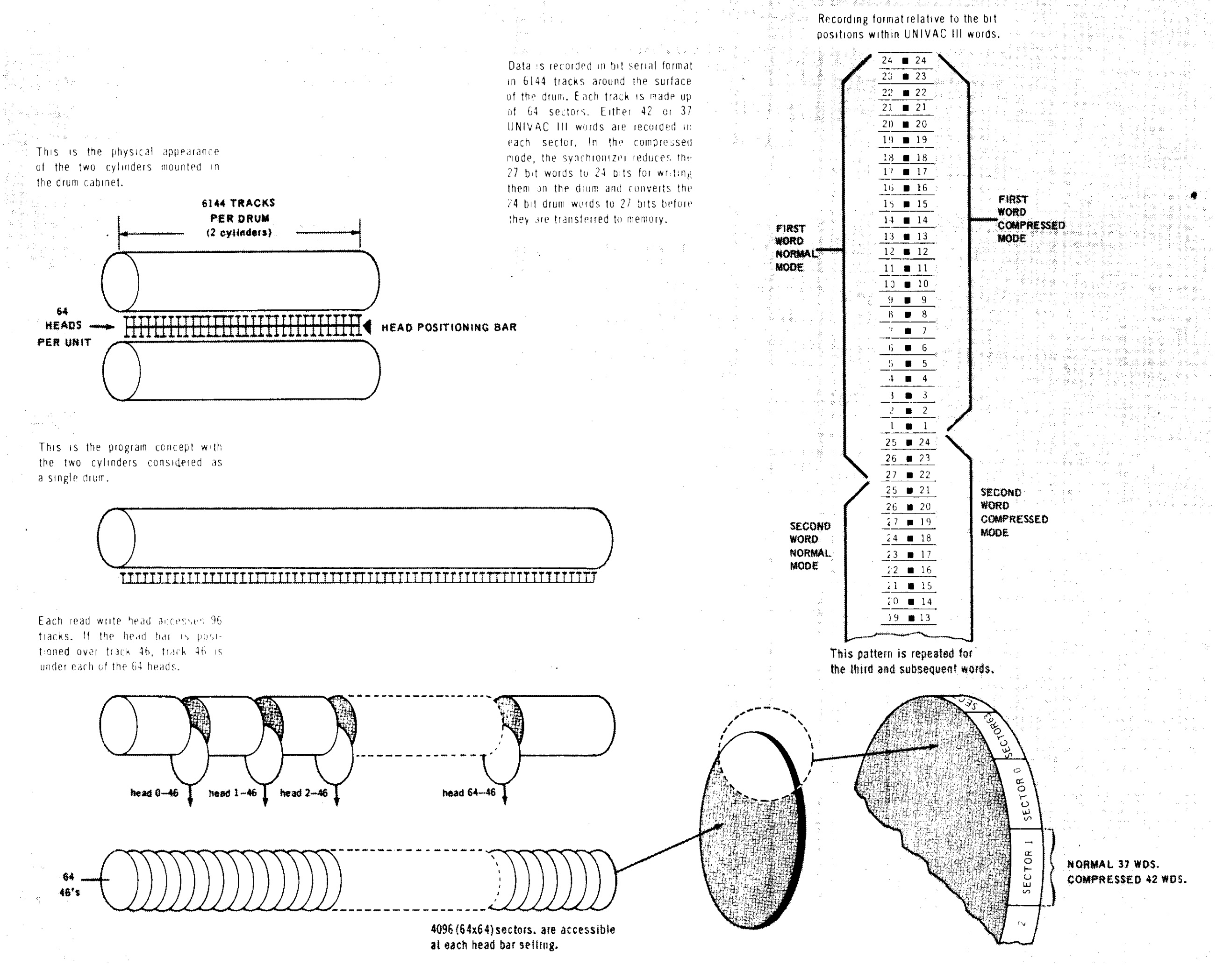
Information is written on the surface of the drum at a density of 1000 PPI. Each sector contains 1170 bit positions; some of which are used for sentinels, hardware control, and parity check patterns. Data is transferred between the central processor and the drum at the rate of 1.09 ms. per sector. Up to 128 contiguous sectors may be read or written in a single operation; up to 64 sectors may be read following a successful search comparison.

The full address of each sector consisting of the drum unit, track, head, and sector is recorded within the sector itself. The address is placed in the sectors prior to delivery of the subsystem to the customer. This area of the sector is not accessible to the user programmer. The instruction address is checked with this pre-recorded address when a read/write instruction is executed to verify that the proper drum address is being accessed.

b. Drum Storage

Figure 17-F-2 shows how data is recorded in specific locations around the drum surface. Each track is divided into 64 sectors. Each sector holds up to 168 alphanumeric characters, or 252 decimal characters in addition to control and parity check information. Data records may be extended over many sectors, or packed within a single sector.

The sector addresses start with zero and run through 63 for any given track. Reading or writing can be performed over a contiguous drum area ranging from one to 128 sectors. When sector 63 is encountered during the execution of a FASTRAND function, the head address is automatically incremented. As a result, sector zero of the track at the next higher head address is processed following sector 63. A continuous read/write instruction should not attempt to process sectors beyond the 64th head. If an instruction attempts to read beyond sector 63 of head 63, the operation will be terminated after sector 63 has been processed and a "head overflow" error will be signalled. A single read/write function must be limited to the range of one head bar setting; it cannot exceed 128 sectors.



Data is recorded in bit serial format in 6144 tracks around the surface of the drum. Each track is made up of 64 sectors. Either 42 or 37 UNIVAC III words are recorded in each sector. In the compressed mode, the synchronizer reduces the 27 bit words to 24 bits for writing them on the drum and converts the 24 bit drum words to 27 bits before they are transferred to memory.

This is the physical appearance of the two cylinders mounted in the drum cabinet.

This is the program concept with the two cylinders considered as a single drum.

Each read write head accesses 96 tracks. If the head bar is positioned over track 46, track 46 is under each of the 64 heads.

4096 (64x64) sectors, are accessible at each head bar setting.

Recording format relative to the bit positions within UNIVAC III words.

24	■	24
23	■	23
22	■	22
21	■	21
20	■	20
19	■	19
18	■	18
17	■	17
16	■	16
15	■	15
14	■	14
13	■	13
12	■	12
11	■	11
10	■	10
9	■	9
8	■	8
7	■	7
6	■	6
5	■	5
4	■	4
3	■	3
2	■	2
1	■	1
25	■	24
26	■	23
27	■	22
25	■	21
26	■	20
27	■	19
24	■	18
23	■	17
22	■	16
21	■	15
20	■	14
19	■	13

This pattern is repeated for the third and subsequent words.

Figure 17-F-2. FASTRAND Data Storage Concept

c. File Organization

The concurrent processing ability of the UNIVAC III greatly reduces the need for optimizing file organization. Efficient random access will improve the total elapsed time for a given program, however, the time saved will be that involved in functions executed under control of the FASTRAND synchronizer. If the scheduling of program operation is such that the central processor can be kept busy processing other files and programs, the central processor time requirement for execution of the FASTRAND programs will be, essentially, the same regardless of file organization.

When it is desired to organize a FASTRAND application to minimize its elapsed running time, the physical location of the records must be considered. It will be helpful to regard each of the drum units as a series of 96 separate drums. The 64 read/write heads are fixed to the positioning bar which must be set at one of 96 different positions. (See *Figure 17-F-2).* The information under each of these heads can be randomly addressed without changing the bar position.

Continuous read/write operations can be performed through the sectors on the tracks serviced by the head at the next higher address. For example, the head bar might be set to position 47. All of the data on tracks 47 under each of the 64 read/write heads are available for reading or writing operations subject only to the latency of rotation. Thus, with this single access movement, 4096 sectors become available. These sectors can then be regarded as constituting an individual cylinder of information with a capacity of 684,128 alphanumeric characters (See *Figure 17-F-3).* The positions to either side of the original setting can be considered as adjacent cylinders of equal capacity. It can then be seen that an additional 1,368,256 characters can be accessed by the minimum lateral movement of the head bar.

d. Recording Modes

Data is recorded on the drum in two modes. The treatment of data differs when transferred between the central processor and the drum according to the recording mode used.

(1) Normal Mode

The entire 27 bits of each UNIVAC III word are stored on the drum when data is written in this mode. The 25 addressable bits and the two Mod-3 check bits are preserved by the synchronizer, and are transferred intact (See *Figure 17-F-4).* A total of 37 UNIVAC III words, plus nine additional bits are stored for each sector written in normal mode. The nine addition bits are in the form of:

000scc000

where: **s** is the sign of the 37th word

cc are the two Mod-3 check bits of the 37th word

000...000 are binary zeros.

In multiple sector operations, the 38th word in memory corresponds to the first word of the next sector, etc.

Note: Data written in the normal mode can be read in the compressed mode without error indication. Data written in the compressed mode should not be read in the normal mode because of the likelihood of Mod-3 error indication.

(2) Compressed Mode

This mode provides for the transferring of bit positions 1–24 of each UNIVAC III word between the central processor and the drum. The sign and Mod-3 check bit positions are deleted by the synchronizer before the information is recorded. When data is read back to central processor in this mode, a zero (plus) sign and appropriate Mod-3 check bits (See *Figure 17-F-4)* are manufactured by the synchronizer, and are jammed into each word after 24 bits have been read. Recording in this mode permits 42 UNIVAC III words to be stored per sector. See *Figure 17-F-3* for the relationship of UNIVAC III content to data stored on the drum. In multiple sector operations, the 43rd word corresponds to the first word of the next sector, etc.

e. FASTRAND Functions

All functions of the FASTRAND are executed under program control. The Executive Routine executes a LOAD CHANNEL (**LC**) instruction which sets the standby indicator and furnishes the address of the function specification to be executed. In most cases the FASTRAND function specifications must be supplemented by control words. The address of the first control word is furnished by the function specification. The memory area for use in conjunction with any particular function must follow directly behind the associated control words (in successively higher addresses).

(1) Control Words

One of a possible set of control words, the Drum Address control word, furnishes the FASTRAND synchronizer with the drum area at which the function is to start. Another control word furnishes the range over which the instruction is to be executed. This Drum Range control word appears in one of two forms, depending upon the specific function with which it is associated. A third control word furnishes the search read instructions with the key for which they are to search. Specific control word requirements will be found in the detailed description of each function specification.

(2) Control Word Formats

The Drum Address control word may be fabricated for use in a FASTRAND function request by the source coding. An alternate approach is to establish a table containing a separate control word for each sector at which a FASTRAND function is to start. If the table technique is to be used, the programmer must provide the table. The control word must be present at the address specified by bit positions 1–15 of the associated function specification before the initiate input/output function instruction is executed (usually by the Executive Routine).

(a) Drum Address Control Word

The Drum Address control word has the following format:

FIELD NAME	Must be Zero	Drum Unit Address	Head Bar Position	Head Address	Sector Address
BIT POSITION	25 24	23 21	20 13	12 7	6 1

Control Word Content:

BITS

1-6 Contain a binary number in the range of 000000 through 111111 indicating one of 64 sectors with which the specified function is to start; the lowest order sector is zero.

7-12 Contain a binary number in the range of 000000 through 111111 indicating the address of one of the 64 heads. This field specifies the track containing the sector with which the function is to start; the lowest order head is zero.

13-20 Contain a binary number in the range of 00000000 through 01011111 indicating one of the 96 track positions to which the head bar must be set to access the sector with which the function is to start; the lowest order track position is zero.

21-23 Contain a binary number in the range of 000 to 111 indicating one of the eight possible drum units which contains the sector with which the function is to start; the lowest order drum unit address is zero.

24-25 Must contain zeros.

(b) Drum Range Control Word

A Drum Range control word, when required, must appear in memory immediately following the Drum Address control word. This word must be fabricated by the worker program. The Drum Range control word must be delivered to the required location before the associated **LC** instruction is executed. The Drum Range control word has two formats.

For other than search functions the following format is used:

FIELD NAME	Must be Zeros		Sector Count	
BIT POSITION	25	8	7	1

Control Word Content:

BITS

1-7 Contain a binary number in the range of 0000000 through 1111111 indicating the number of successive sectors over which the function is to be performed. The sector processed after 63 will be sector 00 on the track under the read/ write head at the next higher address. If the original number is zero, or if an attempt is made to process beyond sector 63 of head 63, an error signal will be given and processing of the function which caused the signal will be discontinued. When 0000000 appears, the function will be performed over 128 sectors; 0000001 indicates a single sector.

8-25 Must be zero.

For Search 1 and Search 2 instructions, the Drum Range control word will have the following format:

FIELD NAME	Must be Zero	Head Count	Sector Count
BIT POSITION	25 11	10 7	6 1

Control Word Content:

BITS

1-6 Contain a binary number in the range of 000000 through 111111 indicating the number of sectors to be processed after the key has been located. The number 000000 will be interpreted to mean that 64 sectors are to be processed. The sector processed after 63 will be sector 00 on the track under the head at the next higher address. If an attempt is made to process beyond sector 63 of head 63, an error signal will be given, and processing of of the function which caused the signal will be discontinued.

7-10 Contain a binary number in the range of 0000 through 1111 indicating the number of tracks over which the search is to be made. Zeros in this field will be interpreted as 16 heads; 0001 indicates a single head.

11-25 Must contain zeros.

(c) Drum Search Control Word

A Drum Search control word, when required, must appear in memory immediately following the associated Drum Range control word. This word must be an exact replica of the word for which the search is to be made. If the data is being searched in compressed mode, bits 1-24 of the Drum Search control word will be used; if the data is being searched in normal mode, bits 1-25 will be used.

(3) Positioning the Head Bar

The positioning of the head bar can be omitted from an instruction if it is known that the bar is already in position. Use of this technique will serve to reduce access time by five ms. One programming technique which may be utilized when processing a FASTRAND file is to let the FASTRAND synchronizer determine when head bar positioning is required. Request for FASTRAND functions can be made without specifying head bar positioning on the assumption that it will not be required in the usual case. When the synchronizer attempts to execute the requested function, and cannot find the specified address on the track under the designated head, a drum address error is signalled. The Executive Routine will automatically resubmit the request with head bar positioning specified.

(4) Writing on the Drum

Four write instructions are available for recording data on a FASTRAND drum when a write instruction is executed; information stored in a number of consecutive memory locations can be written on the drum in 37 or 42 word increments depending upon the choice of recording mode. Up to 128 increments can be written in adjacent drum storage areas during the execution of a single instruction.

The drum write instructions (function specifications) must be used in combination with Drum Address and Drum Range control words. The memory area from which the information is written immediately follows the control words. A write function specification contains the m' (15 bit)* address of the Drum Address control word which, in turn, indicates the area on the drum at which recording is to start. The Drum Range control word specifies the number of drum sectors to be written during this operation. All writing functions are performed after the heads have been positioned over the designated track position, and the appropriate sector has been encountered by the designated head.

(5) Reading from the Drum

Eighteen different instructions are available for reading data recorded on FASTRAND drums. These instructions fall into several categories, each of which are discussed below. All reading functions are performed after the heads have been positioned over the designated track, and after the appropriate sector has been encountered by the specified head.

(a) Reading to Memory

Four instructions are available for reading data from a drum to memory. When a read instruction is executed, the information stored in the specified drum address (sector, track, position, and unit) is transferred to memory under control of the synchronizer. The data is read from the drum serially in either 37 or 42 word increments depending upon the reading mode specified. Up to 128 increments (sectors) can be read from the drum into contiguous memory words. During a continuous read, the information is transferred to memory in ascending order from consecutively higher numbered sectors until sector 63; data in sector zero of the track under the next higher address head follows sector 63 in memory. (No additional time is required for head switching).

The Drum Read instructions (function specifications) must be used in combination with Drum Address and Drum Range control words. The memory area directly following these control words will be used as the read input area by the associated function. The Read Function specification word contains the address of the Drum Address control word. The Drum Address control word indicates the area of the drum from which the reading is to start. The Drum Range control word must be in the next higher order memory location from the Drum Address control word; it specifies the number of drum sectors to be read to memory.

*m' in this case represents a 15-bit address; the address furnished by a function specification is not indexed.

(b) Read Check

Four instructions are available for verification of data after it has been written on a drum. This operation does not result in the transfer of data to the central processor. The normal mode Read Check instructions perform a Mod-3 check (See *Figure 17-F-4)* of data appearing on the drum. With the exception of transferring data to memory, Read Check instructions are executed in the same fashion as that described for the Read to memory instructions. The sequence of reading, the drum range, and control word requirements are the same. The same recording mode should be used for Read Check as was used for writing the data.

(c) Search Reading

There are eight Search Read instructions for use in reading data from the drum when it is not convenient to furnish the specific location of the data. An additional control word (the Search Key control word) provides the FASTRAND synchronizer with a one word key. The synchronizer searches through from 1 to 16 tracks for the key word and reads only the information following that word to the central processor.

The Search instructions (function specifications) must be used in combination with Drum Address and Drum Range control words. The Search Key control word must immediately follow the Drum Range control word in memory, and it in turn is followed by the input area into which the drum datais read (See *Figure 17-F-3).*

Two types of Search instructions are available depending upon the system used for assignment of keys and item layouts. The search can be limited to only the first word of each sector or the data content of entire sectors can be searched. In either case, the key word is not actually transferred to memory, but the transfer of data starts with the word immediately following it. The number of sectors to be read after the key is located is designated by the Drum Range control word.

(d) Contingency Read

Two Contingency Read instructions are available for the recovery of information from the drum under extenuating circumstances. These instructions can be used to force reading of information to memory after a persistent error condition has blocked the transfer of data during one of the usual read instructions. These contingency read instructions have been provided to ensure against even the remote possibility of losing data from a FASTRAND file.

(6) Store Status Word and Terminal Drum Address

This instruction has been provided to furnish the programmer with full information concerning the transfer of information between the drum and the central processor. The FASTRAND synchronizer can be instructed to store two words in memory for programmed analysis. One of these words (called the status word) supplements the information provided by the program-testable indicators. It provides the means for determining the exact nature of any abnormal condition causing the unsuccessful completion indicator to be set (See *Subsection 17-F-1h).*

	m'	m'+1	m'+2	m'+3	m'+39	m'+40	m'+41	m'+42	m'+43	m'+44	m'+45
READ NORMAL	DA CONTROL WORD	DR CONTROL WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	NEXT SECTOR					
READ COMPRESSED	DA CONTROL WORD	DR CONTROL WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	DATA WORD 38	DATA WORD 39	DATA WORD 40	DATA WORD 41	DATA WORD 42	NEXT SECTOR
WRITE NORMAL	DA CONTROL WORD	DR CONTROL WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	NEXT SECTOR					
WRITE COMPRESSED	DA CONTROL WORD	DR CONTROL WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	DATA WORD 38	DATA WORD 39	DATA WORD 40	DATA WORD 41	DATA WORD 42	NEXT SECTOR
READ CHECK NORMAL	DA CONTROL WORD	DR CONTROL WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	NEXT SECTOR					
READ CHECK COMPRESSED	DA CONTROL WORD	DR CONTROL WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	DATA WORD 38	DATA WORD 39	DATA WORD 40	DATA WORD 41	DATA WORD 42	NEXT SECTOR
1st WORD SEARCH NORMAL	DA CONTROL WORD	DR CONTROL WORD	DS CONTROL WORD	DATA WORD * 2	DATA WORD 37	NEXT SECTOR					
1st WORD SEARCH COMPRESSED	DA CONTROL WORD	DR CONTROL WORD	DS CONTROL WORD	DATA WORD * 2	DATA WORD 37	DATA WORD 38	DATA WORD 39	DATA WORD 40	DATA WORD 41	DATA WORD 42	NEXT SECTOR
SEARCH AND READ NORMAL	DA CONTROL WORD	DR CONTROL WORD	DS CONTROL WORD	DATA WORD * 2	DATA WORD 37-n	← Only the words following the key word are read to memory NEXT SECTOR					
SEARCH AND READ COMPRESSED	DA CONTROL WORD	DR CONTROL WORD	DS CONTROL WORD	DATA WORD * 2	DATA WORD 37-n	Only the words following the key word are read to memory 38-n	39-n	40-n	41-n	42-n	NEXT SECTOR
STORE ADDRESS AND STATUS	STATUS WORD	TERMINAL DRUM ADDRESS									
CONTINGENCY READ	DA CONTROL WORD	DUMMY WORD	DATA WORD 1	DATA WORD 2	DATA WORD 37	DATA WORD 38	DATA WORD 39	DATA WORD 40	DATA WORD 41	DATA WORD 42	PARITY WORD 43 ↑ Special parity character word
POSITION HEAD BAR	DA CONTROL WORD										
NO OPERATION	NO AREA NEEDED										
TEST DRUM ADDRESS	DA CONTROL WORD										
	m'	m'+1	m'+2	m'+3	m'+39	m'+40	m'+41	m'+42	m'+43	m'+44	m'+45

Figure 17-F-3. Memory Work Area Requirement for Execution of FASTRAND Functions

* *Data word one is the key word which is not read into memory.*

NORMAL MODE								COMPRESSED MODE						
WORD NO.	BIT POSITIONS IN RECORDING SEQUENCE[1] →						FRAME NUMBER	BIT POSITIONS IN RECORDING SEQUENCE[1] →						WORD NO.
1	24	23	22	21	20	19	1	24	23	22	21	20	19	1
	18	17	16	15	14	13	2	18	17	16	15	14	13	
	12	11	10	9	8	7	3	12	11	10	9	8	7	
	6	5	4	3	2	1	4	6	5	4	3	2	1	
	25	26	27	25	26	27	5	24	23	22	21	20	19	2
2	24	23	22	21	20	19	6	18	17	16	15	14	13	
	18	17	16	15	14	13	7	12	11	10	9	8	7	
	12	11	10	9	8	7	8	6	5	4	3	2	1	
	6	5	4	3	2	1	9	24	27	22	21	20	19	3
	24	23	22	21	20	19	10	18	17	16	15	14	13	
							11 ↓ 162							
37	24	23	22	21	20	19	163	12	11	10	9	8	7	41
	18	17	16	15	14	13	164	6	5	4	3	2	1	
	12	11	10	9	8	7	165	24	23	22	21	20	19	42
	6	5	4	3	2	1	166	18	17	16	15	14	13	
	25	26	27	0	② 0	0	167	12	11	10	9	8	7	
EXTRA FRAME	(WORD 37) 25	26	27	0	② 0	0	168	6	5	4	3	2	1	
SHIFT PATTERN	0	0	1	1	0	③ 0	169	0	0	1	1	0	0 ③	SHIFT PATTERN
PARITY CK. CHAR.	P	P	P	P	P	③ P	170	P	P	P	P	P	P ③	PARITY CK. CHAR.

[1] *The data is read from the drum sectors in the same sequence in which it was recorded.*

[2] *Binary zeros.*

[3] *Six binary bits.*

NOTE: The bit positions of the various words are shown in the column above the parity check character position with which they are associated.

Figure 17-F-4. Derivation of Parity Check Character Positions.

The other word brought into memory by this instruction furnishes the Terminal Drum Address. It indicates the last drum sector accessed by the most recently[1] executed drum function. Both words are explained in detail in *Subsection 17-F-1f(2)-(v)*. The Executive Routine executes this instruction when it detects an abnormal condition and makes the information available to the operating program via a communication packet.

(7) No Operation

The NO OPERATION instruction can be transferred to the FASTRAND synchronizer, and if the transfer is successful, the successful completion indicator will be set. This instruction can be used for drum functions in the same capacity as the NO OP instruction in central processor repertoire. It can be used as a program switch, or to reserve space for another function which is to be substituted for it during processing.

(8) Drum Test

The DRUM TEST instruction is similar to the NO OPERATION instruction in that its execution does not result in the transfer of data between the drum and central processor. It is used to verify that a specific drum address is available to the program before an instruction to transfer data is given.

(9) Operation of the Synchronizer

The FASTRAND synchronizer/control unit, like other synchronizers used for input/output media to the UNIVAC III Data Processing System, makes use of the input/output interrupt feature. When the FASTRAND subsystem is available to perform a function, its synchronizer accesses the memory word reserved for use by its particular channel (channel 1). The function specification is transferred to the synchronizer where it is checked for Mod-3 parity and decoded for execution.

If the function specification is one that is to be supplemented by control words, these words are then transferred to the synchronizer. The control words are also checked for Mod-3 parity, and if found to be correct, the execution of the function will begin. When the execution of the function has been successfully initiated, the standby interlock indicator will be reset, clearing the way for the execution of a new **LC** instruction. If a search function has been requested, the Drum Search control word is read into the synchronizer from memory before the standby interlock indicator is reset.

The initiated function will continue under the control of the synchronizer until terminated by either a successful completion signal, or an error indication. An error signal terminates the transfer of information between the synchronizer and the central processor.

When bit position 16 of the function specification contains a one, the program testable indicator (bit 2) is set when a function has been completed successfully. If an error signal has been encountered, indicator (bit 7) is set, and the setting of the successful completion indicator (bit 2) is inhibited. The synchronizer manufactures a special status word which particularizes error conditions.

f. Drum Instruction Format

The FASTRAND function specifications are submitted to the Executive Routine in combination with other words to form a function request packet. The instruction words discussed below are the specific formats that must be present for execution by the synchronizer at object time. The source coding to produce the object code and the associated instruction packets are discussed in *Subsection 17-F-2*.

[1] *This address is automatically incremented by one when equality is reached with a given sector's "dog tag address" (See Table 17-F-3). Thus, this word will usually contain a value equal to the address of the sector following the last sector processed.*

(1) Load Channel Instruction (**LC**)

The program controls the execution of FASTRAND drum operations through an **LC** instruction supplemented by an input/output function specification and one or more control words. The **LC** instruction accesses the FASTRAND synchronizer and specifies the location of a specific input/output function specification which is to be executed.

LOAD CHANNEL

OP Codes: Alphanumeric – **LC**
Octal – **70**

FUNCTION: Transfer the function specification from the indexed memory location to the fixed standby location in memory associated with channel one (designated in bit positions 11-14); set the respective standby location indicator.

OPERATION FLOW: (m′) → Channel standby location for channel 1; set channel standby interlock indicator.

EXECUTION PERIOD: 3 cycles

INSTRUCTION WORD:

The **LC** object code word has the following format:

FIELD NAME	$^{I}A/F_{S}$	INDEX REGISTER	OPERATION CODE	CHANNEL	ADDRESS (Unindexed)
BIT POSITION	25	24 21	20 15	14 11	10 1

SPECIFICATION:

Indirect Address/Field Select – Indirect addressing is allowed.

Index Register – Four bit binary value designating index register whose contents are to be used during the indexing cycle.

Operation Code – 70 (Octal)

Channel – 0101 (binary) for the FASTRAND Subsystem

Address – Ten-bit value to furnish the unindexed address of the associated function specification.

NOTES:

- The **LC** instruction places the FASTRAND function specification in the standby memory location associated with channel one. The program testable standby interlock indicator associated with channel 1 is set, thereby alterting the FASTRAND synchronizer that a function specification is available for execution.

- When the channel is free to perform a drum operation, the function specification is transferred to the synchronizer for execution. The interlock indicator will not be reset to zero until the necessary number of control words have also been transferred to the synchronizer.

- The binary value of the channel designation is the address of the standby location associated with the channel; it is **0101** for channel 1.

- Indirect addressing may be used; field selection may not.

(2) Function Specifications

The function specifications which are loaded into the standby locations by the **LC** instruction are explained below. *Table 17-F-2* is a compendium of the function specifications applicable to the subsystem when connected to a UNIVAC III Computer. For brevity of presentation, the drum instructions are described in pairs where the operation flow, instruction format, and notes apply equally to both instructions. The instruction names, their function codes and function descriptions are shown individually followed by the common information.

Deviation from the word patterns indicated is not permitted. If the FASTRAND synchronizer receives a function specification with any but the prescribed patterns while the operating controls are not properly set, a portion of the data stored on the drum could be destroyed.

HEAD BAR	FUNCTION	MODE	CONTROL WORDS			MAXIMUM SECTOR RANGE	OCTAL CODE	SALT XPAK	UTMOST FORM
POS.	READ	NORM.	DA	DR	-	128	005	, FPR, ADD.OF DA	005, i, ADD.OF DA,
FIXED	READ	NORM.	DA	DR	–	128	001	, FR , ADD.OF DA,	001, i, ADD.OF DA,
POS.	READ	COMP.	DA	DR	–	128	105	4, FPR, ADD.OF DA,	105, i, ADD.OF DA,
FIXED	READ	COMP.	DA	DR	--	128	101	4, FR, ADD.OF DA,	101, i, ADD.OF DA,
POS.	WRITE	NORM.	DA	DR	–	128	006	, FPW, ADD.OF DA,	006, i, ADD.OF DA,
FIXED	WRITE	NORM.	DA	DR	–	128	002	, FW, ADD.OF DA,	002, i, ADD.OF DA,
POS.	WRITE	COMP.	DA	DR	–	128	106	4, FPW, ADD.OF DA,	106, i, ADD.OF DA,
FIXED	WRITE	COMP.	DA	DR	-	128	102	4, FW, ADD.OF DA,	102, i, ADD.OF DA,
POS.	RD. CHK.	NORM.	DA	DR		128	007	, FPRC, ADD.OF DA,	007, i, ADD.OF DA,
FIXED	RD. CHK.	NORM.	DA	DR	–	128	003	, FRC, ADD.OF DA.	003. i, ADD.OF DA,
POS.	RD. CHK.	COMP.	DA	DR	–	128	107	4, FPRC, ADD.OF DA	107, i, ADD.OF DA,
FIXED	RD. CHK.	COMP.	DA	DR	-	128	103	4, FRC, ADD.OF DA.	103, i, ADD.OF DA,
POS.	1st. WD.SER.	NORM.	DA	DR	DS	S = 1024 R 64	015	. FPS1, ADD.OF DA,	015, i, ADD.OF DA,
FIXED	1st. WD. SER.	NORM.	DA	DR	DS	S 1024 R 64	011	. FS1, ADD.OF DA,	011, i, ADD.OF DA,
POS.	1st. WD. SER.	COMP.	DA	DR	DS	S 1024 R = 64	115	4,FPS1, ADD.OF DA,	115, i, ADD.OF DA,
FIXED	1st. WD. SER.	COMP.	DA	DR	DS	S = 1024 R = 64	111	4. FS1, ADD.OF DA,	111, i, ADD.OF DA,
POS.	SER. & RD.	NORM.	DA	DR	DS	S = 1024 R = 64	016	, FPS2, ADD.OF DA,	016, i, ADD.OF DA,
FIXED	SER. & RD.	NORM.	DA	DR	DS	S = 1024 R = 64	012	, FS2, ADD.OF DA,	012, i, ADD.OF DA,
POS.	SER. & RD.	COMP.	DA	DR	DS	S = 1024 R = 64	116	4, FPS2, ADD.OF DA,	116, i, ADD.OF DA,
FIXED	SER. & RD.	COMP	DA	DR	DS	S = 1024 R 64	112	4, FS2, ADD.OF DA,	112, i, ADD.OF DA,
POS.	HD. BAR	–	–	–	–	–	004	, FPHB, ADD.OF DA,	004, i, ADD.OF DA,
–	STAT. & A	–	–		–	–	014	,STSW, WK.AREA ADD,	014, i, WK AREA ADD,
POS.	CONT. RD.	COMP.	DA	–	–	1	017	4, FPCR, ADD. OF DA,	017, i, ADD.OF DA,
FIXED	CONT. RD.	COMP.	DA	–	–	1	013	4, FCR, ADD. OF DA,	013, i, ADD.OF DA,
–	NO OP	–	–	–	–	–	000	, FNOP , ,	000, i, ,
–	TEST ADD	–	DA	–	–	–	200	–	200, i, ADD.OF DA,

Table 17-F-2. FASTRAND Function Specifications

(1) **POSITION READ-NORMAL**

OP Codes: Alphanumeric – **FPR**
Octal – **005**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sector on the track under the designated head; read that sector to memory starting at $m' + 2$ of the address furnished in this instruction word. Continue reading through successively higher address sectors until the number of sectors specified by the Drum Range control word have been read.

(b) **FIXED READ-NORMAL**

OP Codes: Alphanumeric – **FR**
Octal – **001**

FUNCTION: Locate the specified sector on the track under the designated head; read that sector to memory starting at $m' + 2$ of the address furnished in this instruction word. Continue reading through successively higher address sectors for the number of sectors specified by the Drum Range control word.

OPERATION FLOW: Read to memory the specified number of sectors from the drum.
$[S \ldots S_n] \rightarrow m' + 2 \ldots m' + 2 + 37n$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 0 1 0 1	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must be one of the octal values **001** or **005.**

NOTES: ■ The address of the sector from which reading is to start is specified by the Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.

- The number of sectors to be read during the execution of this function is specified by the Drum Range control word. It must be present at $m' + 1$, relative to the address furnished in bit positions 1-15 of this instruction word.

- If sector 63 is reached on any track before the full number of sectors have been read, the operation will continue through the track under the head at the next higher address. The first sector processed on that track will be sector zero.

- An attempt to read beyond sector 63 of head 63 will result in an interrupt signal and the read operation will be terminated.

- If so specified, successful completion interrupt will occur when the function is completed without encountering an error signal.

- The program testable indicator (bit 7) will be set if reading is terminated due to an error or if the read/write heads are not properly positioned. If an error is signaled, the status word will be made available in the synchronizer. Reading will not start if the heads are not properly positioned.

- 37 UNIVAC III words are transferred from the drum for each sector read. Information is read in the form of serially recorded bits along the track, with each 27 bits resulting in a word in memory. In multiple sector operation, the first word of the sector at the next higher address results in the 38th word of data in memory, etc.

- A Mod-3 parity check is performed on data transferred between the drum and the synchronizer.

(c) POSITION READ-COMPRESSED

OP Codes: Alphanumeric – **4FPR**
Octal – **105**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sector on the track under the designated head; read that sector to memory starting at $m' + 2$ of the address furnished in this instruction word. Continue reading through successively higher address sectors for the number of sectors specified by the Drum Range control word.

(d) FIXED READ-COMPRESSED

OP Codes: Alphanumeric – **4FR**
Octal – **101**

FUNCTION: Locate the specified sector on the track under the designated head; read that sector to memory starting at $m' + 2$ of the address furnished in this instruction word. Continue reading through successively higher address sectors for the number of sectors specified by the Drum Range control word.

OPERATION FLOW: Read to memory the specified number of sectors from the drum
$[S \ldots Sn] \longrightarrow m' + 2 \ldots m' + 2 + 42n$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 1 0 0 0 0 0 1	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must be one of the octal values **101** or **105**.

NOTES: ■ The address of the sector from which reading is to start is specified by the Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.

- The number of sectors to be read during the execution of this function is specified by the Drum Range control word. It must be present at m′ + 1, relative to the address furnished by this instruction word.

- If sector 63 is reached on any track before the full number of sectors have been read, the operation will continue through the track under the head at the next higher address. The first sector processed on that track will be sector zero. An attempt to read beyond sector 63 of head 63 will result in an error signal and the read operation will be terminated.

- If so specified, successful completion interrupt will occur when the function is completed without encountering an error signal.

- The program testable indicator (bit 7) will be set if reading is terminated due to an error or if the read/write heads are not properly positioned. If an error is signaled, the status word will be made available in the synchronizer. Reading will not start if the heads are not properly positioned.

- 42 UNIVAC III words are transferred from the drum for each sector read. Information is read in the form of serially recorded bits along the track, with each 24 bits resulting in the low order bits of the word in memory. Bit positions 25-27 are filled in by the synchronizer. A zero bit is placed in position 25; the proper Mod-3 check bits are placed in bit positions 26-27. In multiple sector operation, the first word of the sector at the next higher address results in the 43rd word of data in memory, etc.

(e) **POSITION WRITE-NORMAL**

OP Codes: Alphanumeric – **FPW**
Octal – **006**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sector on the track under the designated head. Write on the located drum sector, the contents of memory starting at address $m' + 2$ relative to that specified in bits 1-15 of this instruction. Write continuously from successively higher sector addresses over the number of sectors specified by the Drum Range control word.

(f) **FIXED WRITE-NORMAL**

OP Codes: Alphanumeric – **FW**
Octal – **002**

FUNCTION: Locate the specified sector on the track under the designated head. Write on the located drum sector, the contents of memory starting at address $m' + 2$ relative to that specified in bits 1-15 of this instruction. Write continuously from successively higher memory locations to successively higher sector addresses over the number of sectors specified by the Drum Range control word.

OPERATION FLOW: Write from memory, through the specified number of sectors $[m'+2\ldots m'+2+37n] \rightarrow S\ldots Sn$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 0 0 1 0	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 ■ Must be one of the octal values **002** or **006**.

NOTES: ■ The address of the sector at which writing is to start is specified by the associated Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.

- The number of sectors to be written during the execution of this function is specified by the Drum Range (DR) control word. The DR control word must be present at $m' + 1$ relative to the address furnished by this instruction word.

- If sector 63 is reached on any track before the full number of sectors have been written, the operation continues through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to write beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- If so specified, successful completion interrupt will occur when the function is completed without encountering an error signal.

- The program testable indicator (bit 7) will be set if writing is terminated due to the detection of an error or if the read/write heads are not properly positioned. Writing will not be started if the heads are not properly positioned.

- If an error is signaled the status word will be made available in the synchronizer.

- 37 UNIVAC III words are transferred from memory for each sector written. Information is recorded serially on the specified track, with each word resulting in 27 bits on the drum. In multiple sector operations the 38th word in memory corresponds to the first word of the sector at the next higher address, etc.

(g) POSITION WRITE-COMPRESSED

OP Codes: Alphanumeric – **4FPM**
Octal – **106**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sector on the track under the designated head. Write in compressed mode on the located drum sector, the contents of memory starting at address $m' + 2$ relative to that specified in bits 1-15 of this instruction. Write continuously from successively higher memory locations to successively higher sector addresses over the number of sectors specified by the Drum Range control word.

(h) FIXED WRITE-COMPRESSED

OP Codes: Alphanumeric – **4FW**
Octal – **102**

FUNCTION: Locate the specified sector on the track under the designated head. Write in compressed mode on the located drum sector, the contents of memory starting at address $m' + 2$ relative to that specified in bits 1-15 of this instruction. Write continuously from successively higher memory locations to successively higher sector addresses over the number of sectors specified by the Drum Range control word.

OPERATION FLOW: Write from memory, through the specified number of sectors
$[m' + 2 \ldots m' + 2 + 42n] \longrightarrow S \ldots Sn$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 1 0 0 0 0 1 0	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must be one of the octal values **102** or **106**.

NOTES: ■ The address of the sector at which writing is to start is specified by the associated Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.

- The number of sectors to be written during the execution of this function is specified by the Drum Range (DR) control word. The DR control word must be present at $m' + 1$ relative to the address furnished by this instruction word.

- If sector 63 is reached on any track before the full number of sectors have been written, the operation continues through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to write beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- If so specified, successful completion interrupt will occur when the function is completed without encountering an error signal.

- The program testable indicator (bit 7) will be set if writing is terminated due to the detection of an error or if the read/write heads are not properly positioned. Writing will not start if the heads are not in the proper position.

- If an error is signaled, the status word will be made available in the synchronizer.

- 42 UNIVAC III words are transferred from memory for each sector written. Information is recorded serially on the track, with each word resulting in 24 bits on the drum; bit positions 25-27 are not written. In multiple sector operations the 43rd word in memory corresponds to the first word of the sector at the next higher address, etc.

(i) **POSITION READ CHECK-NORMAL**

OP Codes: Alphanumeric – **FPRC**
Octal – **007**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sector on the track under the designated head. Read the located sector into the synchronizer for parity and phase shift check, but do not transfer the data to memory. Read continuously through the number of sectors specified by the associated Drum Range control word.

(j) **FIXED READ CHECK-NORMAL**

OP Codes: Alphanumeric – **FRC**
Octal – **003**

FUNCTION: Locate the specified sectors on the track under the designated head. Read the located sector into the synchronizer but do not transfer the data to memory. Read continuously through the number of sectors specified by the associated Drum Range control word.

OPERATION FLOW: Read from the drum to the synchronizer through the specified number of sectors [S ... Sn]→synchronizer

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 0 0 1 1	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must be one of the octal values **003** or **007**.

NOTES: ■ The address of the sector at which read checking is to start is specified by the associated Drum Address control word. The address of this control word is given in bit positions 1-15 of this instruction word.

- The number of sectors to be checked during the execution of this function is specified by the Drum Range (DR) control words. The DR control word must be present in the m′ + 1 address relative to that furnished in bit positions 1-15 of this instruction word.

- If sector 63 is reached on any track before the full number of sectors have been checked, the operation will continue through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to check beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- If so specified, successful completion interrupt will occur when the function is completed without encountering an error signal.

- The program testable indicator (bit 7) will be set if writing is terminated due to the detection of an error or if the read/write heads are not properly positioned.

- If an error is signaled, the status word will be made available in the synchronizer.

- Drum information is transferred to the synchronizer where it is checked for phase modulation, longitudinal and Mod-3 parity. It is not read into memory.

- The memory address register which normally controls the transfer of data between the synchronizer and the central processor will be incremented as though data were being transferred.

k. **POSITION READ CHECK-COMPRESSED** OP Codes: Alphanumeric – **4FPRC**
Octal – **107**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sectors on the track under the designated head. Read the located sector into the synchronizer in compressed mode, but do not transfer the data to memory. Read continuously through the number of sectors specified by the associated Drum Range control word.

l. **FIXED READ CHECK-COMPRESSED** OP Codes: Alphanumeric – **4FPC**
Octal – **103**

FUNCTION: Locate the specified sector on the track under the designated head. Read the located sector into the synchronizer in compressed mode, but do not transfer the data to memory. Read continuously through the number of sectors specified by the associated Drum Range control word.

OPERATION FLOW: Read from drum to the synchronizer through the specified number of sectors [S ... Sn]→synchronizer

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 1 0 0 0 0 1 1	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must be one of the octal values **103** or **107**.

NOTES: ■ The address of the sector at which read checking is to start is specified by the associated Drum Address control word. The address of this control word is given in bit positions 1-15 of this instruction word.

- The number of sectors to be checked during the execution of this function is specified by the Drum Range (DR) control words. The DR control word must be present in the m′ + 1 address relative to that furnished in bit positions 1-15 of this instruction word.
- If sector 63 is reached on any track before the full number of sectors have been checked, the operation will continue through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to write beyondsector 63 of head 63 will result in an error signal and the termination of the operation.
- If so specified, successful completion interrupt will occur when the function is completed without encountering an error signal.
- The program testable indicator (bit 7) will be set if writing is terminated due to the detection of an error or if the read/write heads are not properly positioned.
- If an error is signaled, the status word will be made available in the synchronizer.
- Drum information is transferred to the synchronizer where it is checked for phase modulation and longitudinal parity. It is not read into memory.
- The memory address register which normally controls the transfer of data between the synchronizer and the central processor will be incremented as though data were being transferred.

(m) **POSITION FIRST WORD SEARCH AND READ-NORMAL** OP Codes: Alphanumeric – **FPS1**
Octal – **015**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been positioned, locate the specified sector on the track under the designated head. Read the first word of each sector into the synchronizer and compare this word with the Drum Search control word until an equal condition is detected. Upon finding the key word, read the information from the second and succeeding words of the sector into the central processor starting at memory location m′ + 3 relative to the address specified in bits 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

(n) **FIXED FIRST WORD SEARCH AND READ-NORMAL** OP Codes: Alphanumeric – **FS1**
Octal – **011**

FUNCTION: Locate the specified sector on the track under the designated head. Read the first word of each sector into the synchronizer and compare this word with the Drum Search control word until an equal condition is detected. Upon finding the key word, read the information from the second and succeeding words of the sector into the central processor starting at memory location m′ + 3 relative to the address specified in bits 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

OPERATION FLOW: Search the first word of each sector on up to 16 tracks. Read to memory the specified number of sectors upon encountering the specified key.
$[S \ldots S_n] \longrightarrow m' + 3 \ldots m' + 39 + 37(n-1)$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE									INT.	DA CONTROL WORD ADDRESS														
WORD CONTENT	0	0	0	0	0	1	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
BIT POSITION	25								17	16	15														1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 The operating program is interrupted automatically upon completion of all search functions; bit 16 can be either zero or one.

17-25 Must be one of the octal values **011** or **015**.

NOTES:

- The address of the sector at which the search is to begin is specified by the associated Drum Address (DA) control word. The address of this control word is given in bit positions 1-15 of this instruction word.

- The number of tracks over which the search is to be conducted is given in bit positions 7-10 of the Drum Range (DR) control word. 0000 establishes a search range of 16 tracks.

- The number of sectors which are to be read to memory during the execution of this function, is specified in bits 1-6 of the DR control word. The DR control word must be present at address m′ + 1 relative to that furnished in bit positions 1-15 of this instruction word.

- If sector 63 is reached on any track before the full number of sectors have been searched and/or read, the operation continues through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to search and/or read beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- The word at address m′ + 2 relative to that given in bit positions 1-15 of this instruction word is the Drum Search control word. Before the search is begun, it is transferred to the synchronizer as if a write instruction was being executed.

- The program testable indicator (bit 7) is set either before or after completion of this operation. The status word must be brought into memory from the synchronizer and examined by the program to determine the success of the operation.

- When sector 63 of the track at the upper limit of the search range is encountered before the key word is located, the search is terminated. The program testable indicator (bit 7) is set.

- Bit positions 1-25 of the key word are compared with bit positions 1-25 of the first word of each sector.

- Except for the sector containing the key, 37 UNIVAC III words are brought into memory of 27 bit increments for each sector read. In multiple sector reading, the 38th word in memory (starting with the key word) comes from the first word of the sector at the next higher address, etc.

- The second and succeeding words of the sector containing the key are read to the central processor. These words occupy contiguous memory locations immediately following the Search Key control word, thus, the resultant input area appears as if the key word has been read to memory following the DR control word.

(o) POSITION FIRST WORD SEARCH AND READ-COMPRESSED

OP Codes: Alphanumeric – **4FPS1**
Octal – **115**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been positioned, locate the specified sector on the track under the designated head. Read the first word of each sector into the synchronizer in the compressed mode. Compare this word with the Drum Search control word until an equal condition is detected. Read in compressed mode the information from the second and succeeding words of that sector into memory starting at location $m' + 3$ relative to the address specified in bits 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

(p) FIXED FIRST WORD SEARCH AND READ-COMPRESSED

OP Codes: Alphanumeric – **4FS1**
Octal – **111**

FUNCTION: Locate the specified sector on the track under the designated head. Read the first word of each sector into the synchronizer in the compressed mode. Compare this word with the Drum Search control word until an equal condition is detected. Read (in compressed mode) information from the second and succeeding words of the sector into the central processor starting at memory location $m' + 3$ relative to that specified in bit positions 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

OPERATION Search the first word of each sector on up to 16 tracks. Read to memory the specified number of sectors upon encountering the specified key.
$[S \ldots Sn] \longrightarrow m' + 3 \ldots m' + 44 + 42\,(n-1)$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 1 0 0 1 0 0 1	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 The operating program is interrupted automatically upon completion of all search functions; bit 16 can be either zero or one.

17-25 Must be one of the octal values **111** or **115**.

NOTES:

- The address of the sector at which the search is to begin is specified by the associated Drum Address (DA) control word. The address of this control word is given in bit positions 1-15 of this instruction word.

- The number of tracks over which the search is to be conducted is given in bit positions 7-10 of the Drum Range (DR) control word. 0000 establishes a search range of 16 tracks.

- The number of sectors which are to be read to memory during the execution of this function is specified in bits 1-6 of the DR control word. The DR control word must be present at address m′+ 1 relative to that furnished in bit positions 1-15 of this instruction word. 000000 is interpreted to specify 64 sectors to be read.

- If sector 63 is reached on any track before the full number of sectors have been searched and/or read, the operation continues through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to search and/or read beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- The word at address m′ + 2 relative to that given in bit positions 1-15 of this instruction word is the Drum Search control word. Before the search is begun, it is transferred to the synchronizer as if a write instruction was being executed.

- The program testable indicator (bit 7) is set either before or after completion of this operation. The status word must be brought into memory from the synchronizer and examined by the program to determine the success of of the operation.

- When sector 63 of the track at the upper limit of the search range is encountered before the key word is located, the search is terminated. The program testable indicator (bit 7) is set.

- Bit positions 1-24 of the key word are compared with bit positions 1-24 of the first word of each sector.

- Except for the sector containing the key, 42 UNIVAC III words are read from each sector in 24 bit increments. The synchronizer inserts a zero in bit position 25 and proper Mod-3 check bits in positions 26 and 27. In multiple sector reading, the 43rd word in memory (relative to the DS control word) comes from the first word of the sector at the next higher address, etc.

- The second and succeeding words of the sector containing the key are read to the central processor. These words occupy contiguous memory locations immediately following the Drum Search control word, thus, the resultant input area has the same format as if the key word had been read to memory following the DR control word.

(q) POSITION SEARCH AND READ-NORMAL

OP Codes: Alphanumeric – **FPS2**
Octal – **016**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been positioned, locate the specified sector on the track under the designated head. Read all words of that and each succeeding sector into the synchronizer and compare each word with the Drum Search control word until an equal condition is detected. Read the words following the key word into the central processor starting with location m + 3 relative to the address specified in bit positions 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

(r) FIXED SEARCH AND READ-NORMAL

OP Codes: Alphanumeric – **FS2**
Octal – **012**

FUNCTION: Locate the specified sector on the track under the designated head. Read all words of that and each succeeding sector into the synchronizer and compare each word with the Drum Search control word until an equal condition is detected. Read the words following the key word into the central processor starting with memory location m′ + 3 relative to the address specified in bit positions 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

OPERATION FLOW: Search each sector on up to 16 tracks. Upon encountering the key word, read to memory the remainder of that sector plus additional sectors if so specified. [Key wd + 1...Sn] ⟶ m′+3...m′+(39 minus the no. of wds. preceeding the key in key sector) + 37 (n−1).

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE									INT.	DA CONTROL WORD ADDRESS														
WORD CONTENT	0	0	0	0	0	1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
BIT POSITION	25								17	16	15														1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 The operating program is interrupted automatically upon completion of all search functions; bit 16 can be either zero or one.

17-25 Must be one of the octal values **012** or **016**.

NOTES:

- The address of the sector at which the search is to begin is specified by the associated Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.

- The number of tracks over which the search is to be conducted is given in bit positions 7-10 of the Drum Range control word. 0000 in this field designates the range to be 16 tracks.

- The number of sectors to be read to memory after the key word is located is specified in bits 1-6 of the DR control word. The DR control word must be present at address m′ + 1 relative to that furnished in bit positions 1-15 of this instruction. 000000 in this field designates 64 sectors to be transferred after the key is located.

- The Drum Search control word at m′ + 2 is read from memory to the synchronizer in a similar fashion to the execution of a write instruction.

- The words immediately following the located key word are read from the synchronizer to the central processor. This can result in the transfer of less than a full sector. Drum data will be read to memory starting with address m′ + 3 relative to that specified in bit positions 1-15 of this instruction word.

- When data is read to memory, it occupies contiguous memory locations immediately following the Drum Search control word. Thus, the resultant input area has the same format as if the key word had been read to memory.

- Except for the sector containing the key, 37 UNIVAC III words are brought from the drum to memory. In multiple sector reading, the 38th word in memory, adjusted by the number of words read before the key was encountered, comes from the first word of the sector at the next higher address, etc.

- If sector 63 is reached on any track before the full specified number of sectors have been searched and/or read, the operation continues through the track under the head at the next higher address. The first section processed on that track is sector zero. An attempt to search and/or read beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- The program testable indicator (bit7) is set either before or after completion of the operation. The status word must be examined to determine the success of the operation.

- If the key is found to be the last word in the sector, and only one sector has been specified for reading, no data will be read to memory.

(s) **POSITION SEARCH AND READ-COMPRESSED**

OP Codes: Alphanumeric – **4FPS2**
Octal – **116**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been positioned, locate the specified sector on the track under the designated head. Read all words of that and each succeeding sector into the synchronizer and compare each word with the Drum Search control word until an equal condition is detected. Read the words following the key word into the central processor starting with memory location m' + 3 relative to the address specified in bit positions 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

(t) **FIXED SEARCH AND READ-COMPRESSED**

OP Codes: Alphanumeric – **4FSE**
Octal – **112**

FUNCTION: Locate the specified sector on the track under the designated head. Read all words of that and each succeeding sector into the synchronizer and compare each word with the Drum Search control word until an equal condition is detected. Read the words following the key word into the central processor starting with memory location m + 3 relative to the address specified in bit positions 1-15 of this instruction. Read continuously from successively higher address sectors to successively higher memory locations until the number of sectors specified by bit positions 1-6 of the Drum Range control word have been read.

OPERATION FLOW: Search each sector on up to 16 tracks. Upon encountering the key word, read to memory the remainder of that sector plus additional sectors if so specified. [Key wd + 1...Sn] ⟶ m' + 3...m'+(41 minus the no. of wds preceding key in key sector) + 42 (n−1).

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 1 0 0 1 0 1 0	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 The operating program is interrupted automatically upon completion of all search functions; bit 16 can be either zero or one.

17-25 Must be one of the octal values **112** or **116**.

NOTES:

- The address of the sector at which the search is to begin is specified by the associated Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.

- The number of tracks over which the search is to be conducted is given in bit positions 7-10 of the Drum Range (DR) control word. 0000 in this field designates the range to be 16 tracks.

- The number of sectors to be read to memory after the key word is located is specified in bits 1-6 of the DR control word. The DR control word must be present at address $m' + 1$ relative to that furnished in bit positions 1-15 of this instruction. 000000 is interpreted to specify 64 sectors.

- Bit positions 1-24 of the Drum Search control word are compared with bit positions 1-24 of all the words in the searched sectors. The Drum Search control word must be present at address $m' + 2$ relative to that specified in bits 1-15 of this instruction.

- The Drum Search control word at $m' + 2$ is read from memory to the synchronizer in a similar fashion to the execution of a write instruction.

- The words immediately following the key word are transferred from the synchronizer to the central processor. This can result in the transfer of less than a full sector. Drum Data will be read to memory starting with address $m' + 3$ relative to that specified in bit positions 1-15 of this instruction word.

- When data is read to memory, it occupies contiguous memory locations immediately following the Drum Search control word. Thus, the resultant input area has the same format as if the key word had been read to memory.

- Except for the sector containing the key, 42 UNIVAC III words are brought from the drum to the synchronizer in 24 bit increments per word. Bit 25 is set to zero and bits 26 and 27 are set to the proper Mod-3 condition by the synchronizer before each word is brought into memory. In multiple sector reading, the 43rd word in memory (adjusted by the position of the key word within its sector) comes from the first word of the sector at the next higher address, etc.

- If sector 63 is reached on any track before the full specified number of sectors have been searched and/or read, the operation continues through the track under the head at the next higher address. The first sector processed on that track is sector zero. An attempt to search and/or read beyond sector 63 of head 63 will result in an error signal and the termination of the operation.

- The program testable indicator (bit 7) is set either before or after completion of the operation. The status word must be examined to determine the success of the operation.

- If the key word is the last word in its sector and only one sector has been specified for reading in the DR control word, no data will be read to memory.

(u) POSITION HEAD BAR

OP Codes: Alphanumeric – **FPHB**
Octal – **004**

FUNCTION: Initiate positioning of the head bar of the specified drum unit to bring the read/write heads over the specified track addresses.

OPERATION FLOW: Set head bar to specified track. No transfer of data takes place.

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 0 1 0 0	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must always be the octal value **004**.

NOTES:

- The address of the drum unit and the position to which the head bar is to be set specified by the associated Drum Address (DA) control word. The address of the control word is given in bit positions 1-15 of this instruction word.
- The DA control word may contain either zeros or ones in bit positions 1-12.
- No Drum Range control word is required.
- If so specified, successful completion interrupt will occur when positioning has been initiated; the program testable indicator (bit 2) will be set.
- If an error is signaled, the status word will be made available in the synchronizer.
- If a request is received by the synchronizer to position the head bar of a unit which has not finished the execution of a prior positioning instruction, the unfinished operation is halted. The bar is then positioned according to the specification of the most recent instruction.

(v) STORE STATUS WORD AND TERMINAL DRUM ADDRESS

OP Codes: Alphanumeric – **STSW**
Octal – **014**

FUNCTION: Transfer two words from the synchronizer to the memory address specified by bit positions 1-15 of this instruction word. The first of the two words is the status word. The second word furnishes the drum unit, position, head, and sector address of the sector immediately following the last drum area processed.

OPERATION FLOW: $[\text{Synchronizer (two words)}] \rightarrow m'$ and $m' + 1$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE		DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 1 1 0 0	X	X X X X X X X X X X X X X X X
BIT POSITION	25 … 17	16	15 … 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the memory location in which the status word is to be stored.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function; otherwise it is zero.

17-25 Must always be the octal value **014**.

NOTES:

- The address word is stored at $m' + 1$ relative to the address specified in bit positions 1-15 of this instruction.
- The address word is identical in format to the DA control word.
- The address word indicates the sector address of the sector following the last sector processed. If the last sector processed was sector 63, sector zero of the track at the next higher head address is given. If the last sector is sector 63 under the head 63, the head address of the address word will be zero.
- No Control words are used.
- If so specified, successful completion interrupt will occur if the function is completed without the detection of an error signal.
- See *Subsection 17-F-1h* for explanation of status word content.

(w) POSITION CONTINGENCY READ-COMPRESSED

OP Code: Alphanumeric – **FPCR**
Octal – **017**

FUNCTION: Position the head bar of the specified drum unit to bring the read/write heads over the specified tracks. When the heads have been correctly positioned, locate the specified sector on the track under the designated head. Read the contents of this sector into memory in the compressed mode. The first word of the designated sector is transferred to $m' + 2$ relative to the address specified in bit positions 1-15 of this instruction; the suceeding words are read to successively higher memory locations.

(x) FIXED CONTINGENCY READ-COMPRESSED

OP Code: Alphanumeric – **FCR**
Octal – **013**

FUNCTION: Locate the specified sector on the track under the designated head. Read the contents of this sector into memory in the compressed mode. The first word of the designated sector is transferred to $m' + 2$ relative to the address specified in bit positions 1-15 of this instruction; the suceeding words are read to successively higher memory locations.

OPERATION FLOW: Read to memory $[S] \rightarrow m' + 2 \ldots m' + 45$

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT.	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 1 0 1 1	X	X X X X X X X X X X X X X X X
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Must be a one bit if the operating program is to be interrupted automatically upon completion of the function, otherwise it is zero.

17-25 Must always be the octal values **013** or **017**.

NOTES:

- One sector is transferred to memory each time this instruction is executed.
- The address of the sector to be read is furnished by the associated Drum Address (DA) control word. The address of the DA control word is given in bit positions 1-15 of this instruction word.
- The use of the compressed mode avoids the possibility of halting the transfer of data due to a Mod-3 parity error signal which may occur prior to the phase error signal.
- 43 words are transferred to memory. The data from the drum is placed in bit positions 1-24 of the first 42 words. The sign (always zero) and Mod-3 check bits of these words, as they appear in memory, are fabricated by the synchronizer.

  The 43rd word contains the six longitudinal parity check bits as they are read from the drum repeated three times and the phase shift sentinel character. It has the following format:

FIELD NAME	SIGN	PHASE SHIFT SENTINEL	LONGITUDINAL PARITY	LONGITUDINAL PARITY	LONGITUDINAL PARITY
WORD CONTENT	0	0 0 1 1 0 0	X X X X X X	X X X X X X	X X X X X X
BIT POSITION	25	24 19	18 13	12 7	6 1

Word Content:

BITS

1-6 Contain the six longitudinal check bits.

7-12 Contain the six longitudinal check bits repeated.

13-18 Contain the six longitudinal check bits repeated.

19-24 Contain the octal value 14(the shift sentinel). If it is other than 14, a phase shift would have been indicated.

25 Is always zero.

The longitudinal parity check character is explained in *Subsection 17-F-1j*. 168 data characters, the phase shift sentinel character, and the parity character itself are recorded on the drum in odd parity.

(y) **NO OPERATION**

OP Codes: Alphanumeric – **FNOP**
Octal – **00**

FUNCTION: This function specification is accessed by the synchronizer and decoded. It is checked for Mod-3 parity, but results in other action.

OPERATION FLOW: None.

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE	INT	DA CONTROL WORD ADDRESS
WORD CONTENT	0 0 0 0 0 0 0 0 0	X	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BIT POSITION	25 17	16	15 1

Instruction Word Content:

BITS

1-15 Will normally be zero, however, any value can be present.

16 Is a one bit if the operating program is to be interrupted upon successful completion; otherwise it is zero.

17-25 Must always contain the octal value **000**.

NOTES:

- No control words are required. They will be ignored if they are present at an address specified in this instruction.
- If so specified, successful completion interrupt will occur when the function is completed without the detection of an error signal.
- The program testable indicator (Bit 7) will be set if the function is terminated due to an error.
- If an error is signaled the status word is made available in the synchronizer.

(z) TEST DRUM ADDRESS

OP Codes: Alphanumeric – **FTAD**
Octal – **200**

FUNCTION: This function specification is accessed by the synchronizer and decoded. It is checked for Mod-3 parity. The drum address specified by the DA control word is accessed, but no data is transferred.

OPERATION FLOW: None.

INSTRUCTION WORD:

FIELD NAME	FUNCTION CODE									INT.	DA CONTROL WORD ADDRESS														
WORD CONTENT	0	1	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
BIT POSITION	25								17	16	15														1

Instruction Word Content:

BITS

1-15 Contain a binary value giving the location of the associated Drum Address control word.

16 Is a one bit if the operating program is to be interrupted upon successful completion; otherwise it is zero.

17-25 Must always contain the octal value **200**.

NOTES:

- A DA control word is required. A DR control word will be ignored if it is present.
- If so specified, successful completion interrupt will occur when the function is completed without the detection of an error signal.
- The program testable indicator (Bit 7) will be set if the function is terminated due to an error.
- If an error is signalled, the status word is made available in the synchronizer.

g. Interrupt Indicators

The interrupt indicators for the FASTRAND subsystem and the codes used in instructions for testing them are as follows:

INDICATOR INSTRUCTIONS	INDICATOR CODES FOR USE IN INPUT/OUTPUT INTERRUPT INSTRUCTIONS
Standby	1 in bit position 1
Successful Completion	1 in bit position 2
Status word examination necessary	1 in bit position 7

These indicators are tested by the UNIVAC III Executive Routine which furnishes the operating programs with the same information in a different form. The user, therefore, will not usually be required to analyze the indicator settings directly, but will use the information supplied by the Executive Routine. The information provided by the Executive Routine is described in *Subsection 17-F-2.*

(1) Standby Location Interlock Indicator

The standby location interlock indicator can be tested and reset to zero by the execution of appropriate central processor instructions. This indicator is set when the **LC** instruction has delivered a function specification to the memory area associated with channel one (the channel assigned to the FASTRAND subsystem). It is reset to zero when the function specification and the associated control words have been successfully transferred to the synchronizer. In the case of a write instruction, it is not reset until the first two data words are transferred from memory.

If the standby indicator (bit 1) is set at the time a status word indicator (bit 7) is set, the resetting of bit 7 alone will cause the synchronizer to attempt again the execution of the instruction in the standby memory location. This loop continues until bit 1 is reset by the program. The resetting of indicators 1 and 7 through a single instruction will avoid this processing loop.

(2) Successful Completion Indicator

The successful completion indicator (bit 2) is set when input/output interrupt is specified in bit 16 of the function specification word, providing no unusual circumstances are encountered during the execution of the specified function. When this indicator is found to be set, there is no need to examine the status word, although a STORE STATUS WORD AND TERMINAL DRUM ADDRESS may be executed at any time if the terminal drum address is of interest.

The setting of the bit 7 indicator inhibits the setting of the bit 2 indicator. If the synchronizer finds that bit 2 is already set from the execution of a previous function specification and has not yet been reset, it will stop executing further function specifications. When the program resets bit 2 under this condition, the synchronizer will automatically set bit 2 or 7 for the function completed at the time bit 2 was found to be already set.

(3) Status Word Indicator

The setting of this indicator (bit 7) results from either a successful search function or as the result of an error encountered during the execution of a function. The status word is described in *Subsection 17-F-1h.*

The specific condition which causes the status word indicator (bit 7) to be set can be determined by examining the content of a status word. This status word is manufactured by the synchronizer and is stored in memory upon the execution of the STORE STATUS WORD AND TERMINAL DRUM ADDRESS function specification. The cause of the abnormal condition is specified by one bits in specific bit positions within the status word.

h. Status Word

The status word is manufactured by the synchronizer and is stored in memory upon the execution of the STORE STATUS WORD AND TERMINAL DRUM ADDRESS function specification. This word supplements the setting of the status word indicator (bit 7) to establish the specific condition which caused the interrupt. The status word is a pattern code in which the setting of a specific bit position identifies conditions as described in *Table 17-F-3.*

PANEL LIGHT	BIT POSITION SET	CONDITION	EFFECT ON DATA TRANSFER
None	None	Search Read operation successfully completed.	All data transferred.
None	1	Search Read operation completed but the key word was not encountered.	No data transferred.
LOAD (Button)	2	Addressing error detected during read or a Mod-3 parity error detected on either a read or write function.	Transfer of data halted at the point the error is detected.
LOAD (Button)	3	Mod-3 parity error encountered during transfer of a function specification, a control word, or the first two data.words during a write operation.	No data transferred.

Table 17-F-3. Conditions Indicated through the Status Word

PANEL LIGHT	BIT POSITION SET	CONDITION	EFFECT ON DATA TRANSFER
LOAD (Button)	4	Mod-3 parity error on data read from the drum or in the transferof the third and succeeding words during a write function.	Incrementation of the Memory Address Counter is inhibited. If set by a read instruction, data transfer is halted at the point the error is detected. If a write instruction is in progress, the data word being accessed at the time the error was signalled is repeated until the sector is filled.
LOAD (Button)	5	The parity character generated by the synchronizer during a read function is not equal to that read from the drum.	Data transfer is halted before a new sector is read.
LOAD (Button)	6	Synchronizer overflow/ underflow due to interference from other I/O channels.	Incrementation of the Memory Address Counter is inhibited. If a write function is in progress, the data word being accessed at the time that the error was signalled is repeated until the sector is filled. The write function ceases at the end of the sector. If a read function is in process, transfer of data stops when the error is detected.
LOAD (Button)	7	Addressed sector can't be located.	No data transferred.
LOAD (Button)	8	Change in phase modulation detected during a read function (this error may have been preceded by a Mod-3 parity error).	If other than a contingency read function, transfer of data to memory is halted if not stopped by a prior Mod-3 check. If this is a Contingency Read, a special word will be transferred to memory (See Note 1 for the special word conditions).

Table 17-F-3. Conditions Indicated through the Status Word (continued)

PANEL LIGHT	BIT POSITION SET	CONDITION	EFFECT ON DATA TRANSFER
LOAD (Button)	9	A multiple phase shift has been detected.	If other than contingency read function, data transfer is halted at the time the first error is detected. On contingency read function, an entire sector is brought into memory including the phase shift sentinel and parity character.
LOAD (Button)	10	The unit, track and head addresses furnished by the DA control word are not in agreement with the "dog tag" addresses on the track under the specified head.	No data is transferred.
ABNORMAL CLEAR (Button)	11	The addressed drum unit was in not ready status at some point during the execution of the function.	Transfer of data is halted upon the detection of this error.
None	12	Not Used (always zero)	None
ABNORMAL CLEAR (Button)	13	Used for maintenance purposes only.	None
None	14	Not Used (always zero)	None
None	15	Not Used (always zero)	None
ABNORMAL CLEAR (Button)	16	An addressed drum unit is offline or not in the configuration.	No data is transferred.
ABNORMAL CLEAR (Button)	17	The printed circuit card is missing from the synchronizer.	No data is transferred.
None	18	The DR control word has specified a sector range beyond sector 63 of head 63.	The transfer of data is halted after sector 63 of head 63 is read, written, or searched.

Table 17-F-3. Conditions Indicated through the Status Word (continued)

PANEL LIGHT	BIT POSITION SET	CONDITION	EFFECT ON DATA TRANSFER
None	19	DR control word used with a non-search function has other than zeros in bit positions 8-10.	No data is transferred.
None	20	Not Used (always zero)	None
None	21	Not Used (always zero)	None
None	22	Not Used (always zero)	None
None	23	Not Used (always zero)	None
None	24	Not Used (always zero)	None
None	25	Not Used (always zero)	None

Table 17-F-3. Conditions Indicated through the Status Word (continued)

Note 1: A special word has been provided for use after the detection of a phase shift error following a compressed mode read operation. This word can be located by executing a STORE MEMORY-ADDRESS COUNTER instruction designating General Purpose Channel 1. The execution of this instruction produces the address of the special word which has the following format:

FIELD NAME		NOT USED	CHAR. POS.	ERROR CHARACTER
WORD CONTENT	1	X X X X X X X X X X X X X X X X	X X	X X X X X X
BIT POSITION	25	24 9	8 7	6 1

Word Content:

BITS

1-6 Contain the six bit pattern of one of the four characters within a UNIVAC III word.

7-8 Contain a two bit value to which designates the position of the error character within its word. The possible codes and the associated bit positions within the word are:

00 – bit positions 1–6
01 – bit positions 7–12
10 – bit positions 13–18
11 – bit positions 19–24

9-24 Are not significant.

25 Contains a binary 1; the word is always negative.

i. Modulo-3 Parity

The UNIVAC III word contains 27 bits of which the low order 25 are addressable. Bits 26 and 27 are used internally to check arithmetic operations and the parity of data transfers; these bits are not normally addressable. Bits 26 and 27 have greater significance to the FASTRAND programmer than for systems which do not include drum storage. When FASTRAND data is recorded in the normal mode these bits are written on the drum. If normal mode data is read back into the central processor through a CONTINGENCY READ instruction, the Mod-3 check bits will appear in program accessable word areas. Further, the longitudinal parity check character includes these check bits in its parity count. The programmer must be able to predict bit positions 26 and 27 in order to verify the accuracy of the longitudinal parity character.

Instructions, control words, and data are Mod-3 checked between the synchronizer and the central processor regardless of recording mode. This check is also performed on data transferred from the drum to the synchronizer in normal mode. If a Mod-3 error is detected during the transfer of data from the drum to the synchronizer the proper check bits will be jammed into the error word and the transfer of data will be halted. Data transferred in the compressed mode will have the appropriate Mod-3 check bits jammed into bit positions 26 and 27 by the synchronizer before the words are transferred to the central processor.

A simple procedure for determining the Mod-3 check bits for a given UNIVAC III word is as follows:

(a) Ascertain the binary content of the 25 addressable bit positions of the word.

(b) Count the one bits in the even numbered bit positions (2-24).

(c) Multiply this number by two.

(d) Count the one bits in the odd numbered bit positions (1-25).

(e) Add the two numbers (from steps c and d).

(f) Divide the sum by three. The check bits are determined by the remainder in this calculation. The check bits are the remainder expressed as a two position binary value.

IF THE REMAINDER IS	THEN THE CHECK BITS ARE	
	27	26
0	0	0
1	0	1
2	1	0

EXAMPLE: The 27 bits of a UNIVAC III word containing the four alphanumeric characters WORD would be:

		W	O	R	D
0 0	0	1 1 1 0 0 1	1 0 1 0 0 1	1 0 1 1 0 0	0 1 0 1 1 1
27 – 26	25	24 19	18 13	12 7	6 1

j. Parity Check

The FASTRAND synchronizer generates a longitudinal parity check character for each sector written. This six bit parity character is written on the drum following the 1008 bit data storage area (See *Table 17-F-4*). When data is read back into the synchronizer from the drum during either a read or search function, a new parity check character is generated for each sector processed. This new check character is compared to the character read from the drum for equality and if the two are not equal, an error is signalled. Data transfer is halted at the point the error is encountered.

The part of the sector over which this parity check is made is illustrated in *Table 17-F-4. Figure 17-F-4* illustrates the arrangement of the UNIVAC III words in the synchronizer and the method by which the contents of these words are subdivided for recording on the drum. It can be seen from this figure that the words are treated differently depending upon the recording mode employed.

Each of the parity character bit positions apply to specific bits within the UNIVAC III words as illustrated in *Figure 17-F-4*. Note that the Phase Shift Sentinel code is included in the parity check. To predict the parity bit applicable to each column in the figure, count the one bits in the bit positions according to the illustrated columns. If there are no one bits or if the count is an even number, a one bit is inserted in the associated parity word condition. If the one bit count for a given column is an odd number a zero should appear in the associated parity bit position.

(1) Normal Mode

Data recorded in the normal mode results in the division of two 27 bit words into nine six bit frames (See *Figure 17-F-4*). These frames are transferred to the buffer which controls the serial recording of the data. One hundred and seventy frames are required to record 37 words of data, one extra frame, the phase shift pattern, and the odd parity check characters. (The last three frames are fabricated by the synchronizer).

(2) Compressed Mode

Data recorded in the compressed mode results in the transfer of only the low order 24 bits of each word to the drum. Each word is divided into four six bit frames (See *Figure 17-F-4*). These frames are transferred to the buffer which controls the serial recording of the data. One hundred and seventy frames are required to record 42 words of data, the phase shift pattern, and the odd parity check character. (The last two frames are fabricated by the synchronizer).

	FIELD NAME	NO. OF BITS	CONTENTS
	Sector Start Pattern	54	1 1 1 1 1 1 1 1 1 - - - - - - - 1 1 1 1 1 1 1 1 1
	Sector Sentinel	6	0 0 1 1 0 0
	Recorded Address (dog tag)	24	u u u u t t t t t t t t h h h h h h s s s s s s
	Phase Check Sentinel A	6	0 0 1 1 0 0
	Second Start Pattern	18	1 1 1 1 1 1 - - - - - - - - - - 1 1 1 1 1 1 1 1
	Data Storage	1008 (148x6)	d d d d d d - - - - - - - - - - - - d d d d d d
*	Shift Check Pattern	6	0 0 1 1 0 0
	Longitudinal Parity Character	6	p p p p p p
	Phase Check Sentinel B	6	0 0 1 1 0 0
	Blank Space	(50-80 mils)	0 0 0 0 0 0
	Next Sector Start Pattern	54	1 1 1 1 1 1 1 1 1 - - - - - - 1 1 1 1 1 1 1 1 1

LEGEND:		
	uuuu	drum unit address (in binary)
	tttttttt	head bar position address (in binary)
	hhhhhh	head address (in binary)
	ssssss	sector address (in binary)
	dddddd	Data
	pppppp	Parity character

* *These fields included in longitudinal parity check*

Table 17-F-4. Sector Organization

UNIVAC III UTMOST
SECTION 17-F, UP-3853

FASTRAND

SUBSYSTEM

APPENDIX E. DATA FILE CONVENTIONS

This appendix describes the conventions and tape formats for UNISERVO IIIA data files.

A. LABELS

The first block on a tape reel and in a tape file must be a 12-word label block of the form shown in *Table E-1*.[1]

B. DATA BLOCKS

The first and last words of each data block must be data descriptor words, as shown in *Table E-1*. The maximum acceptable data block size is 4096, including data descriptor words.

C. END-OF-REEL SENTINELS

Each reel of a multireel file except the last, is terminated by two one-word end-of-reel sentinel blocks (refer to *Table E-1*), which immediately follow the last data block.

D. END-OF-FILE SENTINELS

The last data block of a file is followed by two one-word end-of-file sentinel blocks of the form shown in *Table E-1*.

E. BYPASS SENTINELS

When a file includes information that is not part of the data proper (for example, a rerun memory dump), the non-data blocks of the file must be preceded and followed by two one-word bypass sentinel blocks. (Refer to *Table E-1.*) The information to be bypassed may appear at any place within the file.

[1] *Tape Files used or created by FORTRAN programs do not contain a label block.*

WORD	SIGN	CONTENT	COMMENTS
LABEL BLOCK			
0	–	0---0	Minus indicates non-data block. Binary 0's indicate label block.
1	+	aaaa	First Four Characters of the Eight Character Alphanumeric file ID
2	+	Date of cycle	All reels of multireel file should contain same date.
3	+	000ddd	Decimal reel number.
4	±	x---x	Unused.
.			
.			
.			
9	±	x---x	Unused.
10	+	aaaa	Last Four Characters of the Eight Character Alphanumeric file ID
11	–	0---0	Minus indicates non-data block. Binary 0's indicate label block.

Table E-1. Data Tape Block Formats

WORD	SIGN	CONTENT	COMMENTS
DATA BLOCK			
0	+	bbbbbbbbbbbbcccccccccccc	Data descriptor word. b---b = Binary no. of items in block. c---c = Binary no. of words in block* Plus indicates data block.
1 . . c-2		DATA	
c-1	+	bbbbbbbbbbbbcccccccccccc	Data descriptor word, identical to word 0.
BYPASS SENTINEL BLOCK			
0	–	010---0	Minus indicates non-data block. Binary 01 indicates bypass sentinel.
END-OF-REEL SENTINEL BLOCK			
0	–	10b---b	Minus indicates non-data block. Binary 10 indicates end-of-reel sentinel. b---b indicates the total number of blocks recorded on this tape (in binary)
END-OF-FILE SENTINEL BLOCK			
0	–	11b---b	Minus indicates non-data block. Binary 11 indicates end-of-file sentinel. b---b indicates the total number of blocks recorded on this tape (in binary)

*Including data descriptor words.

Table E-1 Data Tape Block Formats (Continued)

UNIVAC III
UTMOST General Reference Manual, UP-3853

March 28, 1966

UPDATING PACKAGE "E"

The attached material represents an addition for the UNIVAC III General Reference Manual, UP-3853, and should be utilized in the following manner:

SECTION	FILE PAGES NUMBERED	PLACE AFTER TAB LABELED
Appendix F	i(Table of Contents) pp. 1 - 18	INPUT/OUTPUT (following Appendix E Information)

UTMOST UIII-418 HANDLER

The UNIVAC III-418 Handler coding in the executive routine provides a program and no more than two symbionts access to the UNIVAC 418 System. It controls a single UNIVAC III-UNIVAC 418 Computer Intercoupler attached to General Purpose Channel 7. A special version of the executive routine is employed by the UNIVAC III-418 Handler. This version precludes the utilization of the FASTRAND Handler. It will operate on UNIVAC III systems with 24,576 or 32,768 words of store.

To employ the UNIVAC III-UNIVAC 418 system, the user prepares program pairs. Each program pair contains a UNIVAC III program and a UNIVAC 418 program designed to communicate with each other through the UNIVAC III-418 Handler (of the UTMOST System) and the UNIVAC III Handler (of the ART System) respectively. Each transmission of data from one program to the other requires that one program make a specific request of the other and that the other respond. Either program of a program pair or both, may initiate requests as long as the other is designed to respond.

The UNIVAC III program defines the responses for which it is prepared by forming a response table in a prescribed format. The address of this table is given to the UNIVAC III-418 Handler as a part of the initiation procedure. The UNIVAC III program initiates a request by loading into the arithmetic registers information relating to the response table and transferring control to the UNIVAC III-418 Handler.

The information in the response table must be supplemented by storage work area for use by the handler in transferring data between the two computer systems. The UNIVAC III program is responsible for the allocation of such storage areas as well as for any and all processing of the data involved.

1. Execution Phases

When the UNIVAC III-418 Handler receives and accepts a request from a UNIVAC III program it holds the request until it is possible to initiate it. When any current UNIVAC III-UNIVAC 418 transmission is completed, the Handler transmits to the UNIVAC 418 a control message fabricated from the UNIVAC III program's response table. This three word message is decoded by the UNIVAC 418 which in turn tells the UNIVAC III Handler whether to proceed with its request or not. In the latter case the reason for the rejection is forwarded to the UNIVAC III program as status information to be acted upon by the program. If the UNIVAC 418 signaled the UNIVAC III Handler to proceed, the requested data is transmitted (in the requested direction) and status information is again made available to the UNIVAC III program.

Option is provided to transfer control to the UNIVAC III program immediately upon completion of the transmission to allow the program to interchange buffers, if necessary.

When a request is received from the UNIVAC 418, the UNIVAC III Handler either rejects the request or signals the UNIVAC 418 to proceed. In the latter case data is transmitted between the two computer systems. When the transmission is complete, status information is made available to the UNIVAC III program. The UNIVAC III program may analyze this information to determine its next action. As in the case of UNIVAC III requests, the option is provided to transfer control to the UNIVAC III program immediately upon completion of the transmission to facilitate buffer interchange.

2. Coding of the Response Table

The UNIVAC III program employing the UNIVAC III-418 Handler must contain a single response table prepared in the general format shown in Figure 1. The first five words of the response table are used for major control options and must always exist. These five words may be followed by an arbitrary number of three word response packets which are used to define planned data interchanges with the related UNIVAC 418 program. Following the last three word response packet an end-of-table sentinel is required.

a. Major Controls

1) Handler Use Key: The first word of the response table must initially contain binary zero. This word is modified only by the handler. The UNIVAC III program may test this word to determine whether or not the UNIVAC III-418 Handler is currently using the table. If the value of this word is not binary zero the table is being used by the handler and variation in the content of the table or operating mode as described in subsection 8 should be delayed.

2) Dispatcher Address: The second word of the table must contain the address of a line in the UNIVAC III-UNIVAC 418 dispatcher. In preparing the table the user must write '+U418 RETURN' in this location.

3) Run Identification: The third word of the response table must contain a two-character run identification in the low order character positions. While this identification need not be related to the program ID of the UNIVAC III program it is the program ID of the related UNIVAC 418 program. Hence, the first character of this field must be alphabetic. The second must be alphanumeric.

The run identification should be uniquely defined for each UNIVAC III-UNIVAC 418 program pair in order to avoid erroneous transmission between program pairs.

4) Request Status Key: The fourth word of the table must initially be set to binary zero. The value of this word is modified by the UNIVAC III-418 Handler only in servicing UNIVAC III program requests. It may therefore by ignored by a UNIVAC III program which has been designed only to respond to UNIVAC 418 requests.

When the UNIVAC III program makes requests of the UNIVAC III-418 Handler it may determine the status of the requests by examining this word. The various values of this word, which are fabricated by the handler, are summarized in Figure 3. They are discussed subsequently.

It should be noted that initiation of a second UNIVAC III program request will alter the value of the request status key. Hence, the user program should not initiate a subsequent request prior to determining status of the first.

5) User Deactivate Key: The fifth word of the table may be used to signal the Handler that UNIVAC 418 requests are not to be accepted for this program. This is necessary if variations in the table or change in operating mode is effected as described in subsection 8. If the deactivate key is greater than 0777, the Handler will use the table. If not greater than 0777, the handler will not initiate new actions for the table.

b. Response Packet

The UNIVAC III program must contain a single response packet for every type of transmission it expects. The types of transmission are distinguished by their direction, the size of the related storage area and the type of data to be transmitted. Thus a UNIVAC III program may require one or more response packets of the form shown in Figure 1.

1) Transmission Key: The first word of each response packet is a transmission key. This 12 bit non-zero field must be in the low order positions of the word. The key is used to define a type of transmission allowed by the program. The same key must appear in the UNIVAC 418 program because the UNIVAC III-418 Handler uses this key as the basis for making a request to the UNIVAC 418 or responding to a request from the UNIVAC 418.

Each response packet in the UNIVAC III program must contain a unique request key.

If the key field is negative, the handler will consider the packet to be unavailable. It will accordingly inform the 418 that a requested interchange cannot yet be instituted.

2) Island Address: The second word of each response packet contains a 15 bit island address, whose use is optional. If set to binary zero, completion testing for various transmissions must be done as described in subsections 5 and 7.

If non-zero, the island address is the address of a closed subroutine written by the user which will be entered by the Handler immediately upon completion of the transmission. This subroutine, if written in the form described in subsection 5b, will allow for rapid reaction such as buffer interchange where the user requires it.

3) Buffer Word: The third word of each response packet is composed of three separate fields.

a) Buffer Address

The address field b is the 15-bit address of a storage work area in the UNIVAC III program to be used for this type of transmission. Further information on this storage work area is given in subsection 3.

		LABEL	OPERATION	OPERAND
MAJOR CONTROLS	ORIGIN OF TABLE	UF418	FORM 10, 15	
			+ 0	. HANDLER USE KEY - SEE 2A1
			+ U418RETURN	. DISPATCHER ADDRESS - SEE 2A2
			+ Orrrr	. RUN IDENTIFICATION - SEE 2A3
			+ 0	. REQUEST STATUS KEY - SEE 2A4
			+ 01000	. USER DEACTIVATE KEY - SEE 2A5
		. ONE OR MORE RESPONSE . PACKETS MUST IMMEDIATELY . FOLLOW THE MAJOR CONTROL FIELDS		
RESPONSE PACKET	ORIGIN OF RESPONSE PACKET		+ Okkkk	. TRANSMISSION KEY - SEE 2B1
			+ ISLAND	. ISLAND ADDRESS - SEE 2B2
			UF418 OP,b	. BUFFER WORD
		. AN END SENTINEL MUST . FOLLOW IMMEDIATELY . AFTER LAST RESPONSE PACKET		- SEE 2C
END OF TABLE SENTINEL			-0	

UNIVAC III PROGRAM RESPONSE TABLE

Figure 1

b) Direction of Transmission

The direction field indicates the direction of the transmission. If set to 0 the UNIVAC III program is to send data to the UNIVAC 418. If set to 2 the UNIVAC III program is to receive data from the UNIVAC 418.

c) Buffer Status

The sign field is used to indicate the status of the associated buffer. If the sign is positive, the buffer is ready to send (if direction is 0) or receive (if the direction is 2).

The sign field is employed by the UNIVAC III program to signal the Handler as to what response packets may be used. Since in completing the transmission the buffer is, at least temporarily, not to be used, the Handler sets this word negative.

Thus the UNIVAC III program may determine if a transmission has been completed by testing the sign of this word. If negative, transmission has been completed and appropriate action should be taken.

When a new buffer is assigned to the request packet or when the completed buffer is again ready for use, the UNIVAC III program should set the sign positive.

c. End of Table Sentinel

The UNIVAC III program indicates the end of the response table by a word containing a binary value of -0.

3. Data Areas

The UNIVAC III-UNIVAC 418 Intercoupler transmits only twelve bits to or from the low order positions of the transmission storage buffer in the UNIVAC III program store. In transmitting to the UNIVAC 418, the 12 high order bits are ignored. The sign bit is used to signal the last word to be transmitted. In receiving from the UNIVAC 418 data is placed in positions one through twelve of each word. All remaining positions are set to zero.

The first word of each data buffer is reserved for use by the handler. Thus in preparing a data buffer for transmission to the UNIVAC 418 the user must unpack his data into a series of two character words starting with the second word of the area. The last word to be transmitted must be negative. Figure 2 shows the general form of an output buffer.

When data is received from the UNIVAC 418, the first word of the storage buffer will contain the address of the last word of data received. Thus the user may employ this word to determine the length of the message received. He may then pack the data as he requires for further utilization. Figure 3 shows the general format of an input buffer.

4. Initiation of a Program

Initiation of the UNIVAC III-418 Handler is required if transmission between the two computer systems is to be allowed. When a program is first started or when a rerun is initiated, it is necessary to supply the address of the response table to the handler by

- loading into AR8 the positive address of the response table
- executing a transfer of control to the handler; i.e.,

```
        SLJ     U418RQ

        +1      busy return
        +2      accepted return
```

A typical initiation subroutine is:

```
Label     Operation

             LA     8, (TABLE)
             SLJ    U418RQ
             J      $              . Busy Return - Not entered from Initiate.
                                   . Accepted Return
PROCESS
```

An initiation entry must be made both when a program is originated and restarted from a checkpoint. Therefore, the program must initiate the handler after each entrance to checkpoint.

5. Completion of Response Transmissions

Once the UNIVAC III-418 Handler has been initiated, requests to or from the UNIVAC 418 for transmissions indicated in the Response Table will be accepted. The UNIVAC III program is responsible for determining if any such transmissions have been completed. Two methods of determining completion are available. The first involves direct testing of the response table, the second employs the island code option.

a. Response Table Testing

When a transmission is requested by the UNIVAC 418, the UNIVAC III-418 Handler controls its operation until the transmission has been successfully completed. If it is not successfully completed, the UNIVAC 418 program is informed of the cause and is responsible for any corrective actions. Only if a response has been successfully completed is the UNIVAC III program involved.

When a transmission is completed, the last or buffer word of the associated response packet is set negative. Testing the last word of the relevant packets will isolate those transmissions which have been completed.

b. Island Code Manipulation

If he desires, the user may specify an island code address in the second word of a response packet. The specification of this option does not change the logic or the techniques described in section 5a. The closed subroutine whose address is specified is entered from the handler immediately upon completion of the relevant transmission. The

```
RESPONSE     +042                                                     +027
  PACKET     +0                                                       +0
        UF418    0, OUTPUT                                       UF418    2, INPUT
                                     UF418 FORM  10, 15
```

```
OUTPUT       +0     .RESERVED FOR HANDLER

            b̸b̸DA
              TA
             b̸T
             Ob̸
             BE
             b̸T
             RA
             NS
             MI
             TT
           -b̸b̸ED .SIGNALS LAST
                  .WORD TO BE
                  .TRANSMITTED


OUTPUT BUFFER LAYOUT
```

```
INPUT       INPUT + 7       .ADDRESS OF LAST WORD RECEIVED
                            .STORED BY HANDLER
            b̸b̸DA
              TA
             b̸R
             EC
             EI
             VE
            b̸b̸D.


  INPUT BUFFER LAYOUT
```

UNIVAC 418 DATA AREAS

Figure 2

user may therefore include in the closed subroutine coding to interchange data buffers and perform similar processing which requires quick reaction to the completion of a transmission.

Island coding may appear anywhere in the source program. It is entered via an SLJ to the specified address. At the time of entry, IR1 contains the address of the island code and IR2 contains the address of the response table entry associated with the completed transmission. The user is responsible for retention of the index register environment and must store and restore any additional index registers he may require.

When the associated UNIVAC III-418 request is completed, the UTMOST Executive Routine will execute the following instructions before control is relinquished.

Index Register 1 will be loaded with the address of the island code.

Index Register 2 will be loaded with the address of the response packet just completed which specified this island coding.

Control is transferred to the island coding by execution of a SLJ ISLAND.

The exit from island coding, with all index registers restored to their entry state, is accomplished by executing

```
        J     *ISLAND
```

As shown above, the last line of the island coding is an unconditional transfer to the origin of the island code. This returns program control to the Executive Routine allowing it to complete its function.

Since island coding is a closed subroutine executed as part of the Executive Routine, certain restrictions are imposed on it. These restrictions are listed below.

- Release of control is prohibited.
- All Index Registers must remain intact.
- Requests of the UNIVAC III-418 Handler are prohibited.

6. Initiating a UNIVAC III Request

Should an initiated UNIVAC III program desire to initiate a request, the user must first assure that a packet related to the desired transmission is within the response table. He must further assure that the user deactivate key contains a value greater than +0777; (i.e., that he has not deactivated the run) and that the packet is active, i.e., its first and third words are positive.

The request is then initiated by

- loading into AR8 the negative address of the response packet describing the requested transmission.

- and transferring control to the handler by executing

```
                SLJ        U418RQ

                 +1        . busy return
                 +2        . accepted return
```

a. Busy Return

Since only one request will be handled at a time, the user must provide for the possibility that the handler is busy and his request will not be accepted. If the handler is busy it will return control to the busy return. The UNIVAC III program is responsible for determining its next action and reinitiation, if desired, of the request.

b. Accepted Return

If a request is accepted by the handler, the user will receive control at the accepted return.

A sample initiation is

```
LABEL        OPERATION

               LAN    8, (RESPAK4)
               SLJ    U418RQ
               J      $-2                .Handler Busy - Try Again
                                         .Accepted Return
```

7. Completion of Requested Transmissions

The program which requests a transmission is responsible for overall control of the request. Status information beyond that described in section 5 is given to the UNIVAC III program making the request.

When, and only when, a requested transmission is successfully completed, the third, or Buffer word of the response packet is set negative as mentioned in section 5. It is possible that the Handler will terminate its processing of a request even though the transmission has not been completed. For this reason, the requesting program must contain code to analyze the status of its request. This status is maintained in the Request STATUS Key of the UNIVAC III Response Table.

Response Table Testing

When the handler accepts a request from a program, the Request Status Key is set to a negative non-zero value. When the handler terminates a request, it places a positive octal value in the key. The values of this key are summarized in Figure 3.

8. Turning Off the Handler

Through the use of the User Deactivate Key, the user may disengage his program from the UNIVAC III-418 Handler. By placing a value less than, or equal to, +0777 in the Deactivate Key he prevents the handler from initiating any further action. When the Handler Use Key returns to zero, the user then knows that no UNIVAC 418 activity can affect his program. He must assure that there is no activity should he desire to modify the size of the response table or obtain a conventional checkpoint.

9. Selection of Requesting Program

In most cases major systems considerations will dictate which of the UNIVAC III-UNIVAC 418 program pair should request and which should respond. As chart below indicates the number of input-output interrupts varies according to the choice. Since the number of interrupts provides a guide to the efficiency of the program pair, these figures might be useful in selecting the requesting program where this would otherwise be an arbitrary decision.

UNIVAC III Prog. Requests	UIII INTS.	U418 INTS.
- to send	3	6
- to receive	2	4
UNIVAC III Prog. Responds		
- to send	3	6
- to receive	4	8

10. Channel Assignment

Elements have been placed on the Relocatable Library to facilitate the users assignment of a channel to programs or symbionts.

10a. <u>Principal Programs</u>: The UNIVAC 418 channel is defined by the element MAINCHAN for principal programs. Including the control card

```
        Ƶ  SELECT MAINCHAN
    or
        Ƶ  LIBE BOSS III
```

in the DECO control cards for the program will cause the correct assignment to be made.

10b. <u>Symbionts</u>: A channel number not directly related to the channel to which the UNIVAC 418 is attached is used for symbionts. A different "fictitious" channel number must be assigned to each UNIVAC III symbiont which will concurrently use the UNIVAC III-UNIVAC 418 intercoupler.

Elements defining these "fictitious" channels are included on the Relocatable Library and should be selected when the symbiont is processed by DECO. Since the symbiont channel number is defined during DECO, the operator must only call the symbiont on the channel defined.

Chart below describes the elements on the Relocatable Library.

ELEMENT NAME	CHANNEL NUMBER	LETTER USED IN OPERATOR COMM	FOR USE IN
MAINCHAN	7	Not Applicable	Principal Prog.
U418U14	14	U	Symbiont
U418V15	15	V	Symbiont

11. Formation of Relocatable Library

The executive routine containing the UNIVAC III-UNIVAC 418 Handler will operate on either a 24,576 or 32,768 word store. In order to provide for the creation of the appropriate system, the user must select the appropriate elements from the program library when he is building his relocatable library. In addition, appropriate versions of U418U14 and U418V15 must be selected. The following table will aid in making the selection.

Store Size	Alias Name	Element Name
32,768	U418 Exec	U418EX32
	U418U14	U418U143
	U418V15	U418V153
24,576	U418 Exec	U418EX24
	U418U14	U418U142
	U418V15	U418V152

12. Symbiont Utilization of the Handler

If the user desires to prepare a symbiont program which utilizes the UNIVAC III-UNIVAC 418 Handler, he must also provide for release entries to the handler.

Control should be released to the handler when the completion testing described in sections 5 and 7 indicate the transmissions have been completed. In order to prevent possible loss of an interrupt, a prevent interrupt instruction should be performed before performing the appropriate tests. If no functions are found to be complete, execution of the following calling sequence will release control:

```
         SLJ          *$+1

      +1              +U418RL
      +2              +SAVE           . Address of SAVE area in symbiont
      +3              +COVER          . Symbiont COVER value
      +4              .....release return
```

When an interrupt is received on the channel, control is returned to the symbiont at the release return. Interrupt is not inhibited at the time. If the user wishes to have interrupt prevented on release return, he may execute the instruction

```
      SZ          U418RAI
```

in the symbiont initialization coding.

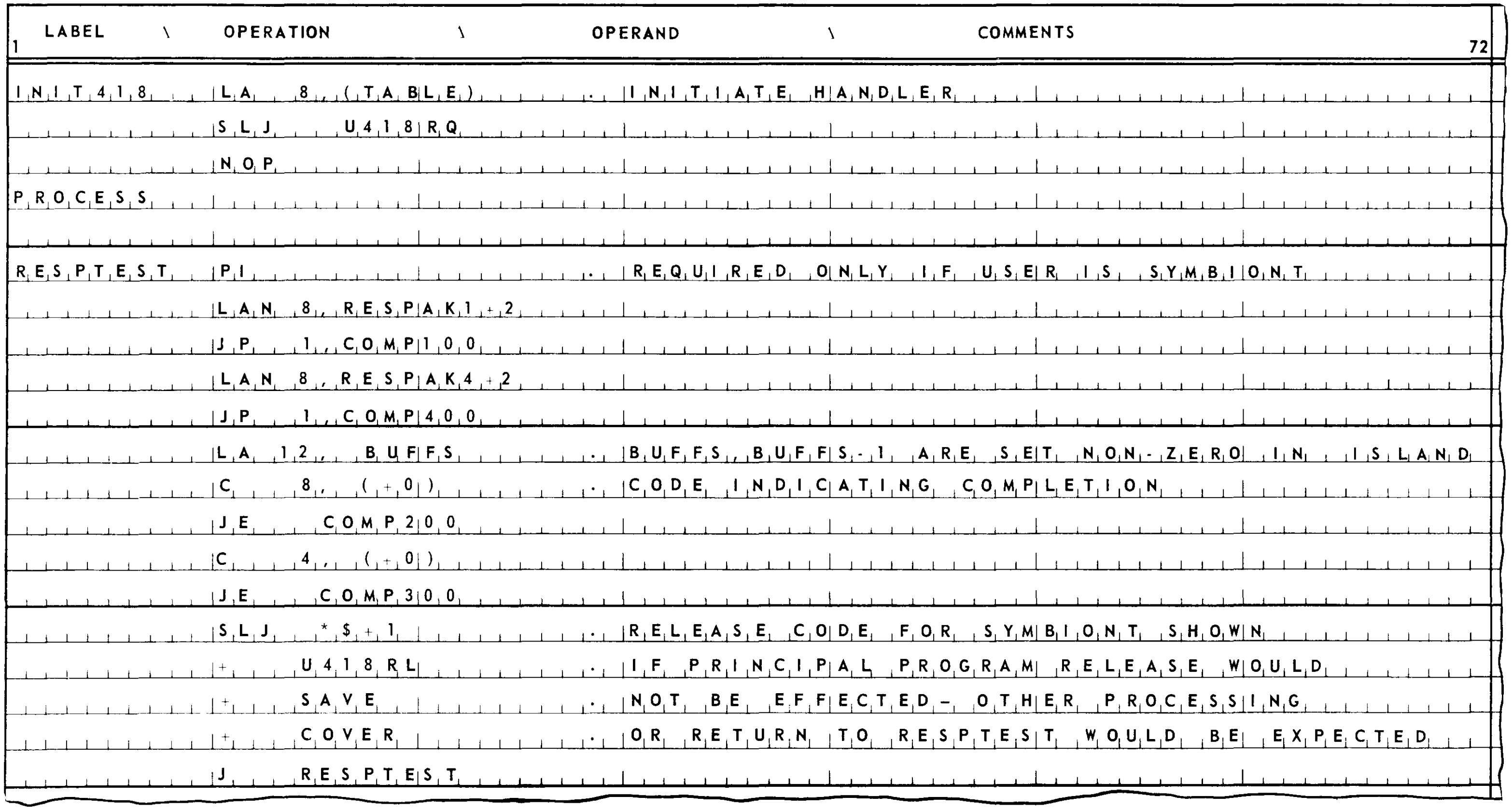
REQUEST STATUS KEY	MEANING	TYPICAL PROGRAM ACTION
<0	Request being serviced by handler.	Do other processing. Test periodically until key ≥ 0.
+000	Requested transmission completed successfully.	Process completed transmission as appropriate and continue.
+03	UNIVAC 418 Program did not have entry with same transmission key, or did not have required buffer ready.	Repeat the request by initiating again. If reply of 03 persists, former reason probably applies.
+04	The UNIVAC 418 program requested (see third word of response table) has not been loaded into UNIVAC 418	Inform operation of situation, then proceed with installation's procedure for this contingency.
+010	Parity error detected while processing request.	Initiate request again. If error persists, terminate program for analysis of cause.
+011	The request caused a transmission to the UNIVAC 418 to exceed the UNIVAC 418 Buffer.	Jettison program in order to analyze cause.

REQUEST STATUS KEY

Figure 3

```
1 LABEL       \  OPERATION        \        OPERAND          \            COMMENTS

UF418             FORM      10,15           . FORM DEFINITION FOR FUNCTION SPEC
TABLE             + 0                       . HANDLER USE KEY
                  + U418RET                 . ADDRESS IN U418 DISPATCHER
                  + Orrrr                   . RUN IDENTIFIER
                  + 0                       . REQUEST KEY
                  + 0                       . DEACTIVATE KEY
RESPAK1           + Okkkk                   . RESPONSE PACKET. MESSAGE KEY
                  + island                                        . ISLAND ADDRESS
                  UF418  op,b                                     . FUNCTION SPEC



RESPAKM           + Okkkk                   . Nth  RESPONSE PACKET
                  + island
                  UF418  op,b
                  - 0                       . TABLE END SENTINEL
```

User Response Table

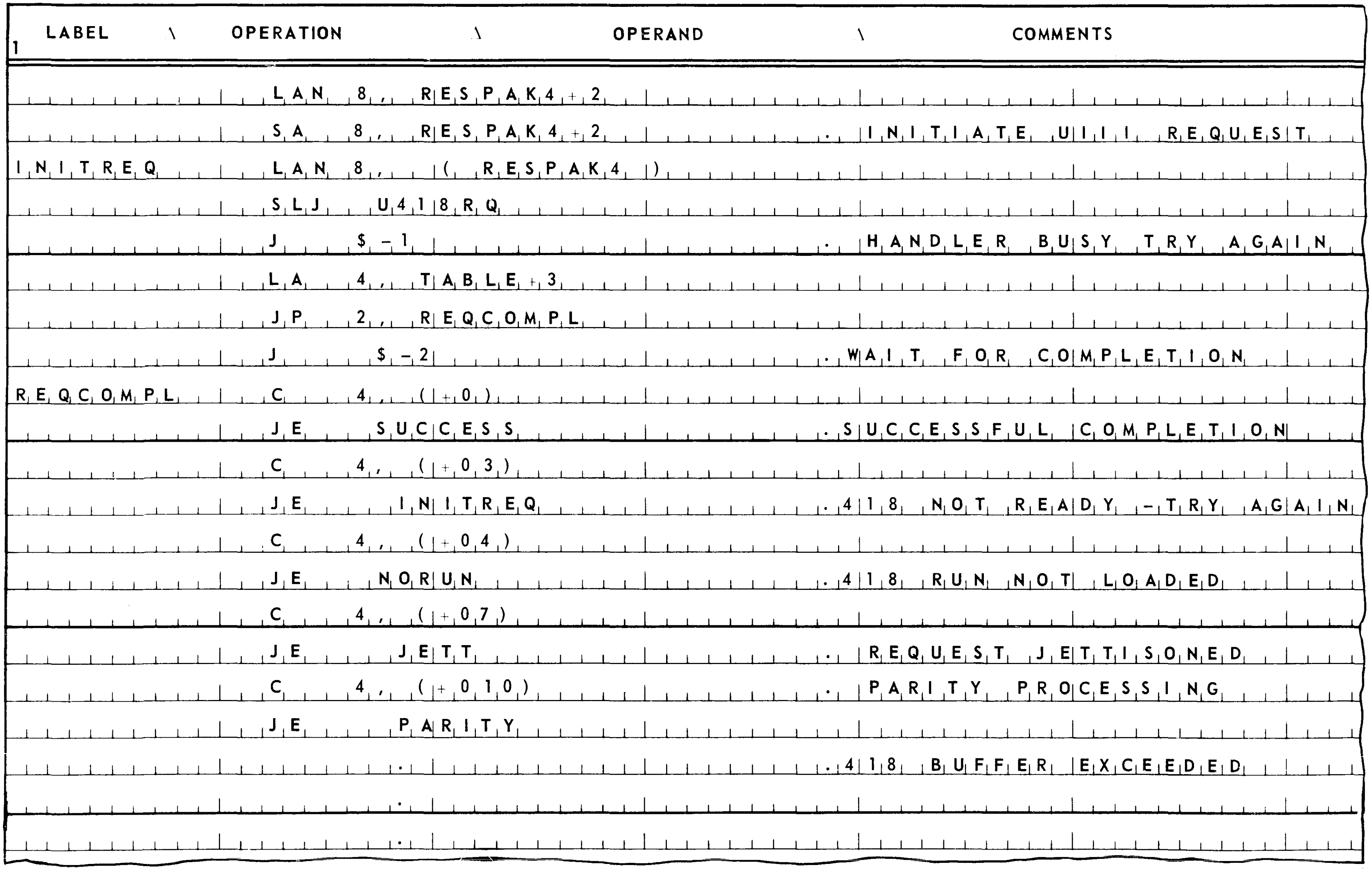


```
 LABEL     \  OPERATION        \        OPERAND          \        COMMENTS
1                                                                                72
INIT418      LA    8,(TABLE)           . INITIATE HANDLER
             SLJ     U418RQ
             NOP
PROCESS

RESPTEST     PI                        . REQUIRED ONLY IF USER IS SYMBIONT
             LAN   8,RESPAK1+2
             JP    1,COMP100
             LAN   8,RESPAK4+2
             JP    1,COMP400
             LA    12, BUFFS           . BUFFS,BUFFS-1 ARE SET NON-ZERO IN ISLAND
             C     8, (+0)             . CODE INDICATING COMPLETION
             JE      COMP200
             C     4, (+0)
             JE      COMP300
             SLJ     *$+1              . RELEASE CODE FOR SYMBIONT SHOWN
             +     U418RL              . IF PRINCIPAL PROGRAM RELEASE WOULD
             +     SAVE                . NOT BE EFFECTED- OTHER PROCESSING
             +     COVER               . OR RETURN TO RESPTEST WOULD BE EXPECTED
             J     RESPTEST
```

Sample User Code Elements

```
  LABEL        OPERATION              OPERAND                    COMMENTS
1
COMP100        LAN   8 ,RESPAK1+2      .  REACTIVATE RESPONSE PACKET
               SA    8, RESPAK1+2


MEMDUMP        SZ       TABLE+4           . DEACTIVATE TABLE
               LA    8, TABLE
               C     8, (+0)
               JG     8-1                 .  HANDLER BUSY


                 INITIATE CHECKPOINT


MDRETNRN       SL    TABLE+4           .  REACTIVATE TABLE
```

Sample User Code Elements

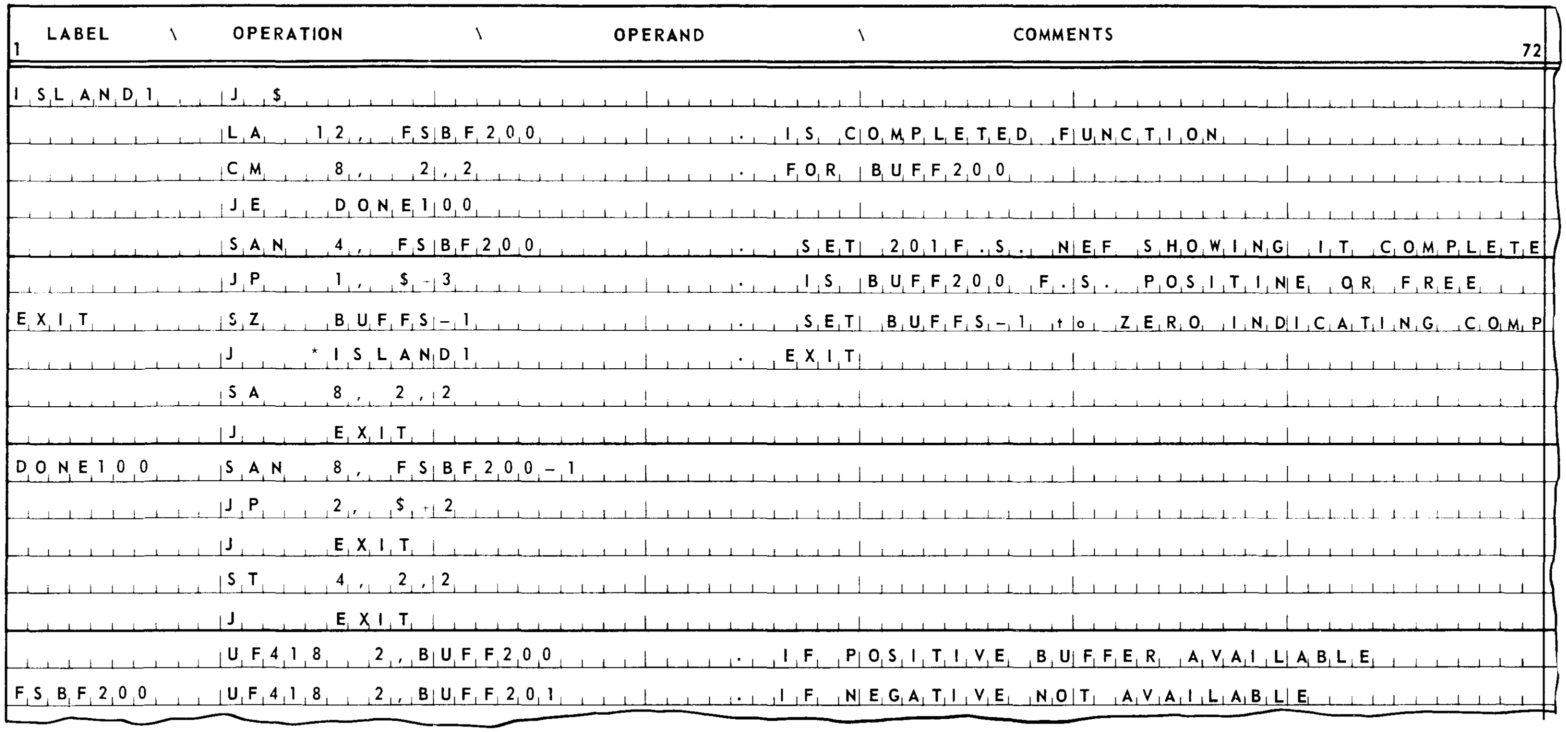


```
1 LABEL     \     OPERATION              \             OPERAND              \            COMMENTS

                  LAN    8,  RESPAK4+2
                  SA     8,  RESPAK4+2                       . INITIATE UIII REQUEST
INITREQ           LAN    8,    (  RESPAK4 )
                  SLJ     U418RQ
                  J        $-1                               . HANDLER BUSY TRY AGAIN
                  LA     4,  TABLE+3
                  JP     2,  REQCOMPL
                  J          $-2                             . WAIT FOR COMPLETION
REQCOMPL          C      4,   (+0)
                  JE       SUCCESS                           . SUCCESSFUL COMPLETION
                  C      4,   (+03)
                  JE         INITREQ                         . 418 NOT READY -TRY AGAIN
                  C      4,   (+04)
                  JE       NORUN                             . 418 RUN NOT LOADED
                  C      4,   (+07)
                  JE         JETT                            . REQUEST JETTISONED
                  C      4,   (+010)                         . PARITY PROCESSING
                  JE         PARITY
                             .                               . 418 BUFFER EXCEEDED
                             .
                             .
```

Sample User Code Elements

```
1 LABEL     \  OPERATION           \            OPERAND           \        CO

UF418          FORM      10,15
TABLE          +  0
               +0418RET
               +  '   A6'
               +  0
               +  01                        . TABLE SET ACTIVE
RESPAK1        +  0100
               +  0                         . NO ISLAND CODE
               UF418   2,BUFF100
RESPAK2        +  0200
               +  ISLAND 1
               UF418   2,BUFF200
RESPAK3        +  0300
               +  ISLAND2
               UFH18   0,BUFF300
               -  0400                      . PACKET INACTIVE
               +0
               UF418   0,REQBUF
               -  0
```

Sample User Code Elements



```
1 LABEL    \  OPERATION        \         OPERAND        \          COMMENTS                             72
ISLAND1       J     $
              LA    12,  FSBF200                    .  IS COMPLETED FUNCTION
              CM    8,   2,2                        .  FOR BUFF200
              JE    DONE100
              SAN   4,   FSBF200                    .  SET 201F.S. NEF SHOWING IT COMPLETE
              JP    1,   $-3                        .  IS BUFF200 F.S. POSITINE OR FREE
EXIT          SZ    BUFFS-1                         .  SET BUFFS-1 to ZERO INDICATING COMP
              J     *ISLAND1                        .  EXIT
              SA    8,  2,2
              J     EXIT
DONE100       SAN   8,  FSBF200-1
              JP    2,  $+2
              J     EXIT
              ST    4,  2,2
              J     EXIT
              UF418   2,BUFF200                     .  IF POSITIVE BUFFER AVAILABLE
FSBF200       UF418   2,BUFF201                     .  IF NEGATIVE NOT AVAILABLE
```

Sample User Code Elements

UTMOST

UNIVAC

DIVISION OF SPERRY RAND CORPORATION