Principles of Programming

Section 9: Random Access File Storage

IBM® Personal Study Program

© 1961 by International Business Machines Corporation

Section 9: Random Access File Storage

9.1 Basic Concepts

So far in this text, we have concentrated on programs built around files on punched cards or magnetic tape. These file storage media have the advantage that they are relatively inexpensive. They have the partially offsetting disadvantage that a record within the file can be accessed only sequentially, since it is not possible to get to one record without passing over all of the records in front of it. This characteristic forces us to sequence all files and transactions according to the keys of the records, which leads to considerable amounts of time spent in sorting. It also forces us to batch the transactions to be processed against the file, since it is not economical to read the entire master file to process a few transactions.

In many applications, these requirements of sequenced files and batch processing are not serious handicaps; in fact, batch processing is in many cases a natural mode of operation. In other applications, however, it would be much more desirable to be able to process transactions immediately as they arise, rather than waiting for batches of them to accumulate, and without sorting. To do so requires a master file storage medium that permits any record in the file to be obtained about as quickly as any other. (This is most definitely not true of magnetic tape.) Such a device is called a *random access file storage medium.*

It must be realized that the time to obtain a record from the currently available random access file storage devices does depend somewhat on the location of the record relative to the location of the record most recently read. Thus the devices are not *truly* random. However, the maximum time to obtain a record is so much less than the time to obtain a randomly placed record from a reel of magnetic tape that we are justified in using the word "random" in comparison with tape file storage. (The only *completely* random access storage device widely used at present is magnetic core storage; the time to obtain a word is absolutely independent of where the previous word was located. In comparison with magnetic core storage, "random access" file storage media are decidedly *not* random, but this is not a relevant comparison, since the much more expensive core storage restricts its use to smaller sizes than are needed for file storage.)

With any of the various random access file storage devices, it is possible and desirable to organize programs in entirely different ways from those employed with magnetic tape file storage. No sorting of the transactions is ordinarily required; transactions can be processed as soon as they reach the data processing center, if desired; the program can be organized to refer to many small files if it is convenient to do so. Furthermore, certain types of applications become feasible that are simply not practical with magnetic tapes.

9.2 The IBM 1405 and 1301 Disk Storage Units

The random access file storage devices available for IBM computers are built around a set of rotating metal disks on which information is recorded. In the IBM 1405, pictured in Figure 1, information is written or read by one or more *access arms* that are able to move to the desired disk and to the desired position on a disk. A 1401 system including the 1405 is called an IBM RAMAC® 1401 System, where RAMAC stands for Random Access Method of Accounting and Control. In the IBM 1301, pictured in Figure 2, information is written and read by a complete set of access arms, one for each disk surface.

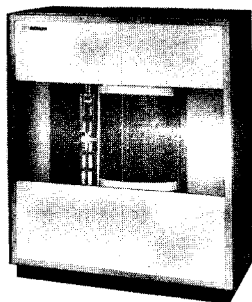

Figure 1. The IBM 1405 Disk Storage unit

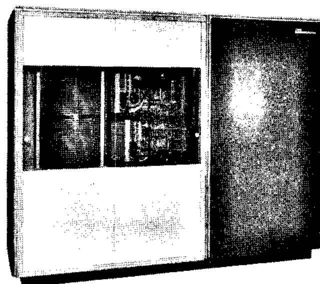

Figure 2. The IBM 1301 Disk Storage unit

The IBM 1405 Disk Storage unit can contain 25 disks (Model 1) or 50 disks (Model 2), storing either ten or twenty million characters. Each disk has 200 concentric *tracks* on which data can be recorded. A track has two sides, one on each surface of the disk. Each track is further divided into ten *sectors*, five on each side: the upper side of each track contains sectors 0 through 4, and the lower side sectors 5 through 9. A track sector, which contains 200 characters, is the smallest unit of disk information that can be addressed.

A 1405 unit normally has one access arm, with a second being available as an optional special feature. The fork-shaped arm has two read-write heads that read and record data on the disks. One read-write head is for the top disk face, and the other is for the bottom disk face. An access arm moves to the position specified by an instruction by moving vertically to the correct disk and horizontally to the specified track on that disk.

The disks rotate on a vertical shaft at the rate of 1,200 rpm. Data is read or recorded at the rate of 22,500 characters per second. The time required to access a record varies between 100 and 800 ms, depending on how far the arm has to move from its previous position.

As with magnetic tapes, the representation of a character on a disk consists of seven bits, including a parity bit. When information is read from the disk, parity is checked and an indicator is turned on if parity is not correct. An instruction is provided with which it is possible to determine whether the information recorded on the disk is actually the same as the information in core storage from which the disk data was written.

Each sector on a disk has a seven-digit address. The first digit specifies the disk storage unit; for a 1401 system, this is always zero. The next four digits specify the track. A Model 1 unit contains 5,000 tracks (0000-4999) ; a Model 2 unit contains 10,000 tracks (0000-9999). The outermost track of the bottom disk has the address 0000 and the innermost track of the bottom disk has the address 0199. The outermost track of the second disk has the address 0200 and the innermost track of the second disk has the address 0399. The tracks on the twenty-fifth disk have addresses running from 4800 at the outside to 4999 at the inside. The tracks on the fiftieth disk have addresses running from 9800 at the outside to 9999 at the inside. The sixth digit specifies the sector. The seventh digit of a disk storage address must always be zero. When a sector is addressed from the computer, it is preceded by a digit specifying the desired access arm; this digit is either zero or one. The arrangement and addressing of disk information is shown schematically in Figure 3. It may be noted from this figure that each record actually contains its address indelibly recorded in seven additional positions at the beginning of the record.

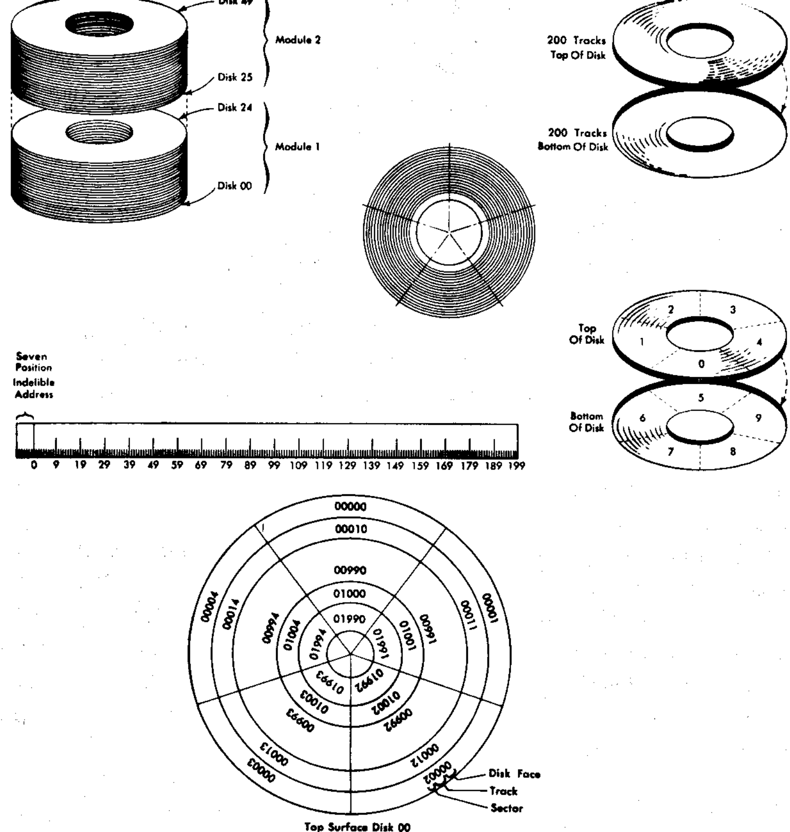

Figure 3. Information arrangement and addressing in the IBM 1405 Disk Storage unit

The IBM 1301

The IBM 1301 Disk Storage unit is similar to the 1405, but in several particulars it is much more powerful. The major change is that instead of having one arm that travels to the specified disk, it has a complete set of arms, one for each disk, arranged in a comb-like array. This means that once the arms have been positioned to a track location, that track on *all* disks is available with no further delay. It is convenient to think of all the tracks accessed from one setting of the arms as composing a *cylinder.*

Record length in the 1301 is flexible, instead of being restricted to 200 characters. This obviously simplifies working with records that are too large or too small to fit conveniently and efficiently in 200 characters.

Besides the radically different arm arrangement and the flexible record length, the 1301 also has greater speed and capacity. The disks turn at 1,800 rpm instead of 1,200, which shortens all delays based on

disk rotation by a third. This increase in speed of rotation, coupled with a greater character density on the disks, raises the transfer rate from 22,500 characters per second to 75,000. Finally, the capacity of a 1301 module is 25 million characters instead of the maximum of 20 million in a 1405 Model 2.

The unique powers of the 1301 lead to program organization and processing techniques that are sometimes markedly different from those used with the 1405. Since we shall not have space to cover the application of both systems, we limit the following discussions to the 1405, which is more widely distributed, at least at present.

Review Questions

1. *How many characters can be stored on one 1405 disk?*
2. *Which disk in a 1405 unit contains sector 33862? How many disks must the access arm go past in moving from sector 08330 to sector 41045?*
3. *True or false: in the 1405, the greatest distance (both horizontally and vertically) that an arm can move is from track 00000 to track 99999.*

9.3 Disk Storage Programming for the IBM 1405

There are five instructions in the 1401 that are used in working with disk storage.

The first of these is the Seek Disk instruction. Before any reading or writing can occur, the access arm must be moved to the desired track. This movement is initiated with a Seek Disk instruction. After the correct track has been located, a separate Read or Write instruction is used to move data.

As with the tape instructions, these instructions have an operation code of M, which is the same as that for Move Characters to a Word Mark. The fact that this is a disk instruction is specified by the first two characters of the A-address, which must be %F. The nature of the disk instruction is specified by the low-order character of the A-address and by the d-character. In a Seek Disk instruction, the A-address is %F0 and the d-character is R. The B-address specifies the high-order position in core storage of the address of the desired sector.

The use of Autocoder considerably simplifies the writing of disk instructions, as it does tape instructions. The augmented operation codes make it unnecessary to write any A-address or the d-character. When the Autocoder processor translates the operation code SD for Seek Disk, for instance, it fills in the %F0 automatically.

Seek Disk

FORMAT

Mnemonic	Op Code	A-address
MCW	M	%F0

B-address	d-character
xxx	R

AUTOCODER FORMAT SD Address

FUNCTION The A-address specifies that a seek operation is to be performed by the access arm. The B-address specifies the high-order position in core storage of the disk record address.
The selected access arm seeks the disk and track specified in the disk record address. Processing can continue while the access arm is in motion.

WORD MARKS Not affected.

TIMING $T = .0115\ (L_I + 9)$ ms + access time.

NOTE If the access arm is already at the disk track (not necessarily at the correct sector) that is to be used, a Seek Disk instruction is not needed.

It should be emphasized that the Seek Disk instruction does not move any data; it simply positions the access arm at the correct disk and correct track. The computer may continue processing with other instructions while the arm is in motion. If a disk Read or Write instruction is encountered before the arm motion is completed, the read or write operation is delayed until the access arm is in correct position.

Once the access arm is positioned at the correct track, a Read or Write Disk instruction can be used to transfer data. Reading is initiated with an instruction that once again has an operation code of M. The A-address must be either %F1 or %F2 and the d-character must be R for read. If the A-address is %F1, then a single 200-character record will be read. If the A-address is %F2, then we have specified what is called a full track read—that is, the specified record and the four following on the same side of the track will be read. The B-address specifies the high-order position in core storage of the disk record address, which might be thought to be redundant since the disk position has already been established by the Seek instruction. However, the presence of the address in the Read instruction allows the accuracy of the machine's functioning to be checked, since at the beginning of each sector is recorded in nonerasable form the address of the sector. Furthermore, the Read instruction is not able to "remember" from the Seek instruction which sector was specified.

Read Disk Single-Record
Read Disk Full-Track

FORMAT

Mnemonic	Op Code	A-address
MCW	M	%Fx

B-address	d-character
xxx	R

AUTOCODER FORMAT
Single-Record: RD Address
Full-Track: RDF Address

FUNCTION This instruction causes data to be read from disk storage into core storage. The digit 1 in the A-address (%F1) specifies that a single record is to be read. The reading of the disk is stopped by a group mark with a word mark in core storage and the end of the sector. If the digit 2 is present in the A-address (%F2) a full-track read occurs. That is, five 200-character records are read from disk storage into core storage. Reading stops at the end of the fifth sector.
The B-address specifies the high-order position in core storage of the disk-record address and is followed by the area in storage reserved for the data read from the disk.
The R in the d-character position signifies that this is a read operation.

WORD MARKS A group mark with a word mark must appear one position to the right of the last position reserved in core storage for the disk record. If a group mark with a word mark is detected before reading of the record is completed, the wrong-length record indicator turns on and reading stops.

TIMING $T = .0115\ (L_I + 9) + 10$ ms + disk rotation.
60.196 ms is maximum time for single-record read.
10.196 ms is minimum time for single-record read.

The data from disk storage is read into core storage beginning immediately after the disk address, the position of which is specified by the B-address. In other words, the B-address specifies where in core storage the disk address is located, and the record is read into core storage immediately after the disk address. The transfer of information stops at the end of the sector or upon encountering a group mark with a word mark in core storage. (Under normal conditions the program is organized so that the two occur at the same time.) The group mark should be one position to the right of the space reserved for the infor-

mation from the disk. If it is encountered earlier than that, the transmission stops and an indicator called Wrong-Length Record is turned on.

Writing of information from core storage to disk storage is carried out with a Write Disk instruction, which is very similar to Read Disk except that the d-character is W.

Write Disk Single-Record
Write Disk Full-Track

FORMAT

Mnemonic	Op Code	A-address
MCW	M	%Fx

B-address	d-character
xxx	W

AUTOCODER FORMAT

Single-Record: WD Address
Full-Track: WDF Address

FUNCTION

This instruction causes a single record (or full-track) in core storage to be written on a disk record. The digit 1 in the A-address (%F1) specifies that a single record is to be written. If a 2 is present in the A address (%F2), five 200-character records are written on a disk track. Writing stops at the end of the fifth sector.

The B-address specifies the high-order position of the disk-record address and is followed by the data to be written on the disk.

The W in the d-character position signifies that this is a write operation.

Before writing starts, an automatic check of the record address in storage, with the record address on the disk, is made. If they are not the same, the unequal-address compare indicator is turned on, and the data in storage is not written on the disk.

WORD MARKS

The writing of data stops when the end of a record is reached or when a group mark with a word mark is sensed in core storage. If the group mark with word mark is sensed before the end of a record, the remainder of the disk record is filled with blanks and the wrong-length record indicator is turned on.

TIMING

$T = .0115\ (L_I + 9) + 10\ \text{ms} + \text{rotation time.}$
60.196 ms is maximum time for a single-record write.
10.196 ms is minimum time for a single-record write.

NOTE

A Write Disk Check instruction must be performed following a write disk operation. No other disk storage operation can be performed until the check of data written on the disk is completed.

A Write Disk instruction must *always* be followed by a Write Disk Check instruction. This instruction causes a character-by-character comparison of data in core storage with the data just written on the disk, and turns on an error indicator if there are any differences. The actual machine instruction is just like a Write Disk instruction except that the A-address must be %F3 and the d-character is W.

Write Disk Check

FORMAT

Mnemonic	Op Code	A-address
MCW	M	%F3

B-address	d-character
xxx	W

AUTOCODER FORMAT

WDC Address

FUNCTION

This instruction causes a character-by-character comparison of the data in core storage with the data just recorded on the disk. The system automatically reads the disk record that was most recently addressed. This instruction *must* follow a Write Disk instruction.

The digit 3 in the A-address specifies that a checking operation is to be performed. Either a single record or a full track is checked, depending on how the data was recorded by the most recent Write Disk instruction.

The B-address specifies the area in core storage where the record address and the data recorded on the disk are located.

WORD MARKS

A group mark with a word mark must appear one position to the right of the disk data in core storage.

TIMING

$T = .0115\ (L_I + 9)\ \text{ms} + 50\ \text{ms}$

NOTE

If the disk address in core storage is not the same as the address in the record, the unequal-address compare indicator is turned on. If any of the characters in the disk record are not the same as the characters in core storage, the read-back check-error indicator is turned on.

In the instructions that we have described, word marks in core storage are not written on the disk and when disk data is read word marks are not affected in core storage. There is, as in the case of tapes, a separate instruction for converting core storage word marks into separate characters when the record is written on the disk. When such a record is read back with a suitable instruction, the special characters are again reconverted to word marks. Beyond this note of their existence we shall not consider instructions for reading and writing word marks.

The final type of disk instruction is simply a variation of one that we met much earlier: Branch If Indicator On. For use with disk storage, we have five additional indicators and corresponding d-characters. These are shown in the summary box for the instruction. "Any Disk Unit Error Condition" comes on if any one of the first three is on. This makes it possible to make one check for any disk errors and, if the indicator is off, to proceed with the normal program. If the indicator is on, only then is it necessary to test the other three indicators to find out which of the errors is present.

Branch If Indicator On

FORMAT

Mnemonic	Op Code	I-address	d-character
B	B	xxx	x

FUNCTION The d-character specifies the indicator tested. If the indicator is on, the next instruction is taken from the I-address. If the indicator is off, the next sequential instruction is taken. The valid d-characters and the indicators they test are as shown below.

d-character	Indicator
V	Read-or-Write Parity or Read-Back Check Error
W	Wrong-Length Record
X	Unequal-Address Compare
Y	Any Disk-Unit Error Condition
N	Access Inoperable

WORD MARKS Not affected.

TIMING $T = .0115\ (L_I + 1)$ ms

For a simple illustration as to how these instructions may be used, we may consider the inventory control application of Section 8.4. We shall assume as before that the transaction cards contain a part number in columns 1 to 5, a code in column 6 indicating an adjustment, a receipt or an issue, and a quantity in columns 7 to 10. We shall assume that we have a 1405 Model 2, which has exactly 100,000 sectors in it and that each master inventory record contains 200 characters. We shall assume that the part numbers are numerical and shall, therefore,

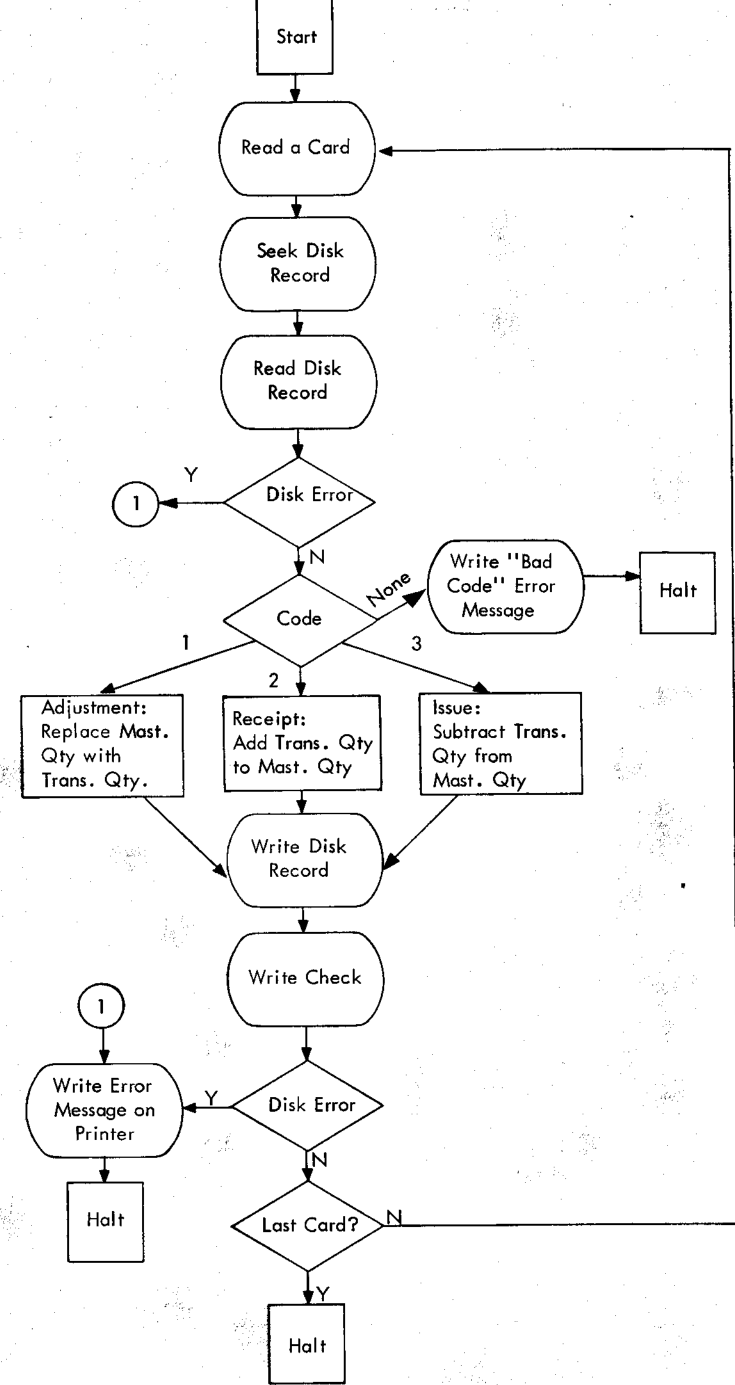

Figure 4. Block diagram of the disk storage approach to the inventory control application of Section 8.4

IBM

Program

Programmed by

Date

1401/1410 AUTOCODER CODING SHEET

FORM X24-1350-1
PRINTED IN U.S.A.

Identification

Page No. 1 of 2

Line	Label	Operation	OPERAND
0.1	START	R	READ A CARD
0.2		MCW	TRANPN, DSKADD-1 SET UP DISK RECØRD ADDRESS
0.3		SD	DSKADD-6 SEEK
0.4		RD	DSKADD-6 READ
0.5		BIN	ERRØR, Y BRANCH IF ANY ERRØR
0.6		BCE	RECØUN, CØDE, 1 DETERMINE
0.7		BCE	RECPT, CØDE, 2 TYPE ØF
0.8		BCE	ISSUE, CØDE, 3 TRANSACTIØN
0.9		LCA	MESSG1, 226 BAD CØDE –
1.0		W	WRITE ERRØR MESSAGE
1.1		H	*-3 AND HALT
1.2	RECØUN	MCW	TRANQY, MSTQY
1.3		B	WRITE
1.4	RECPT	A	TRANQY, MSTQY
1.5		B	WRITE
1.6	ISSUE	S	TRANQY, MSTQY
1.7	WRITE	WD	DSKADD-6 WRITE DISK RECØRD
1.8		WDC	DSKADD-6 CHECK WRITING
1.9		BIN	ERRØR, Y BRANCH IF ANY ERRØR
2.0		BLC	HALT LAST CARD Q
2.1		B	START NØ
2.2	ERRØR	LCA	MESSG2, 222
2.3		W	
2.4		H	*-3
2.5	HALT	H	START BEGIN AGAIN IF PUSH START

Figure 5. Autocoder program of the procedure diagrammed in Figure 4

Page No. 2 of 2

Line	Label	Operation	OPERAND
0.1	DSKADD	DCW	0000000
0.2	FILE	DA	1X200, G
0.3	MSTPN		1, 5
0.4	MSTQY		6, 10
0.5	CARD 0001	DA	1X80
0.6	TRANPN		1, 5
0.7	CØDE		6, 6
0.8	TRANQY		7, 10
0.9	MESSG1	DCW	@BAD CLASS CØDE JØB HALTED@
1.0	MESSG2	DCW	@FILE ERRØR JØB HALTED@
1.1		END	START
1.2			

Figure 5 Continued

be able to use the part number directly as the address of the corresponding master record. The master record will be assumed to contain in positions 1 to 5 the part number and in positions 6 to 10 a quantity.

We shall suppose that transactions are entered in the card reader as small groups of them accumulate. If the job were large enough to occupy a 1401 disk system fully, small groups might be entered almost continuously. If this application were only one of many things being done with the computer, the groups would be entered occasionally, and between times the system could be used for other work.

Notice that it is completely unnecessary to sort the transactions into sequence on any key—since the master file is not in any such sequence, which in turn is possible because it is not necessary to access the file sequentially. For the same reason it is not necessary to batch the transactions, although this might be done as a matter of convenience if the transactions are such as not to need immediate action.

The procedure is now so simple that the block diagram in Figure 4 is hardly needed. As each transaction card is read, the corresponding master record is obtained from disk storage, updated and replaced in disk storage. This simple process is repeated for as many cards as there are.

The program shown in Figure 5 presents no special problems. We begin by reading a card and setting up the disk address, which latter is done by moving the part number into a constant that will have a zero as the first and last digit. Then we seek the disk record having this address and when it is found read it into storage. This a followed by a test for reading errors. Notice in the constants in this program, that immediately following the disk address we have defined an area of 200 characters to hold the record. In this Define Area instruction, notice the G; this will cause a group mark with a word mark to be entered following the 200-character area. Next we inspect the classification code in the card record to determine whether it is a recount, receipt or issue, just as in the magnetic tape version. Once again, if it is none of these three, we write an error message. Next, the transaction quantity is used to update the master record and the master record is written back in disk storage. Notice that there is no Seek instruction here: it is not necessary to seek the correct track if we are already positioned in it. Immediately following the Write, there is the Write Check and a branch to an error routine if there is any file error. Finally, we test the last-card switch and go back to the beginning if there are more cards.

Notice in the constants for this program that the alphabetic constants are entered in Autocoder preceded and followed by the character @.

Review Questions

1. *Must a new Seek instruction be executed if a different sector in the same track is to be read or written?*
2. *Is there any circumstance in which it is not necessary to follow a Write Disk instruction with a Write Disk Check instruction?*

9.4 Disk Organization and Addressing

It may be well to consider precisely how the preceding example is not typical, in order to develop a few of the standard programming techniques in using disk storage.

First of all, we assumed that the entire disk storage was taken up with the master file. This is seldom the case. Usually, space is reserved for programs and for the files of other applications.

The most unrealistic thing about the preceding example is the assumption that the part numbers run from zero to 99999 in an unbroken numerical sequence. Such a simple correspondence between the key of the records and the record addresses is extremely uncommon. For one thing, part numbers are often not purely numerical; they often have letters and symbols in them. Second, even if they are numerical, they are often longer than five digits. Third, whether they are numerical or alphabetic, there are usually many unused numbers in the sequence, so that if we organize disk storage as in the preceding example, a large part of it would never be used, which is obviously uneconomical. Our task for the rest of this subsection is to consider some of the commonly used ways of deriving from the key of a record the disk storage address of that record.

The simplest method is based on deriving the address from the key by simple arithmetic. Suppose, for example, that the keys are purely numerical and seven digits long. Assume, further, that 20,000 disk storage records have been assigned to this file, with record addresses from 50,000 to 69,999. What sort of a scheme could we set up to obtain from such a key an address in the specified range? One way is to proceed as follows: Multiply the seven-digit number by 2, drop the last three digits of the product and add 5 to the high-order position of the remaining digits. A little experimentation will show that for any seven-digit number this yields an address between 50,000 and 69,999.

This method does create a new problem, however: It is very likely that in some cases several keys will convert to the same address. For instance, the keys 1234567 and 1234568 both convert to disk address 52469. This situation is handled by a technique that is known as *chaining*—which has no relation to 1401 address chaining. To understand this technique, we must discuss how disk storage is initially loaded and how the records are obtained when disk storage is later read.

Suppose we are loading storage with records whose addresses are derived by the simple computation described above. The section of storage that is to be loaded is initially cleared to blanks, using a utility program that will be described later. Then, as each record is about to be loaded into storage, the record address is computed from its key. Before storing the record at this address, however, we first check to make sure that the space really is free. If the space still contains blanks, we go ahead and load the record into the sector address as computed. If, however, the space already contains a record because some previous key had converted to the same address, then we store this record in an *overflow* location. This might be the next consecutive record, or it might be in a separate section of storage set up for overflows. Then in the record having the address computed from this key we place an overflow address that specifies where the second record having this same computed address is located. When two or more records have the same disk storage address, we speak of the one that is placed in the computed address location as the *home record* and all of the others as *overflow records*. Each record is said to be *chained* to the one following.

We naturally hope that the characteristics of the keys of the source records, together with the method of computing the addresses, will lead to a minimum of such overflows. This, in fact, is one of the primary considerations in choosing an address computation method.

When disk storage that has been loaded in this fashion is to be read, we go through the same address computation scheme on the key. We seek and read the record at this computed address and then check to see whether it contains the record that we desire, by comparing the key of the record that has been read with the key from which the address was computed. If the two are the same, we are able to proceed immediately with processing. If they are not the same, we must obtain the address of the first overflow record from the record that has been read. When it has been read into core storage, we can similarly inspect its key and find out whether it is the desired one. This process is continued until the proper record has been brought into core storage.

Under unfavorable circumstances, the address computation method suggested above could lead to very long chains of records. This, in turn, would lead to long processing times to search through the chains to find the desired record. For instance, suppose that in one range of the keys there was an unbroken sequence of part numbers, running from 1200000 to 1200499. Every one of these keys would convert to the same address, namely 52400. This would lead to a chain 500 records long, which would obviously be highly undesirable. It appears, therefore, that this method of arriving at a record address applies only if the keys are fairly uniformly distributed over the entire range of possible values. This is frequently not the case, and we therefore attempt to find address computation schemes that will create a fairly uniform distribution of the addresses even when the incoming keys are tightly bunched together in some regions. A good deal of effort has been put into finding such schemes, and the subject is still under development.

It is not possible to state any one method that will *always* lead to a sufficiently uniform pattern. Two methods that *often* work, however, are the following:

1. Split the key into sections of four or five digits and add them. Multiply by a compression factor that will bring the final product into the desired range of address, and add the base address.

Example. Given eight-digit keys that must be changed into sector addresses between 32000 and 36999, which is 5,000 sectors. Take the key 82145369 as a sample.

Add the first four digits to the second four, giving 13583. Multiply by 0.25, which is required to "compress" a number that could be as large as 19998 into a number no larger than 5000, giving 3395. Add the base address, giving the final result: 35395.

2. Split the key into two parts, multiply the two parts and extract the middle five digits. Multiply by a suitable compression factor and add the base address.

Example. Given nine-digit keys that must be changed into sector addresses between 24000 and 38999, which is 15,000 sectors. Take the key 298154726 as a sample.

Multiply the first five digits by the last four, giving 140905690. Extract the middle five digits, giving 09056. Multiply by the compression factor 0.15, giving 01358. Add the base address, giving the final result: 25358.

Another approach is to use some address computation scheme to reach a specified track, without going on to compute a sector within the track. Sector zero within this track is then used as an index to the other nine sectors. That is to say, sector zero contains the keys of all the records stored in that track, together with their addresses. Now, to obtain a record, we compute the track address, seek sector zero on that track, read the index into storage and from that obtain the address of the proper record location. Since the record will be in the same track as the index, only one Seek is required and the method is not too time-consuming. Overflow records become necessary only if the keys of more than nine records convert to the same track address. A disadvantage of this method is that it does require two Read Disk instructions to obtain a record.

The same general idea can be extended even further. An index to the entire file can be set up so that we first locate the proper disk, then go to the outside track on that disk to find an index to the desired track and go from there to the desired record. This has the advantage that little or no address computation is required and that if suitably set up there are no overflow records.

Review Questions

1. *Using the address computation scheme outlined at the beginning of this subsection, to what disk sector address does 0085692 convert? How about 8882450?*
2. *What is the basic idea of chaining?*
3. *What is the most important factor in choosing a randomizing formula for computing the address of the home record in a chained file?*

9.5 Disk Storage Utility Routines

A number of utility programs are available for simplifying work with files stored in random access storage. A brief description of some of these routines will also allow us to introduce a few more ideas about how disk storage may be used.

Clear Disk Storage. The clear disk storage program erases all data in areas of disk storage specified by the user, and fills these areas with blanks. The program can clear disk storage completely or only in selected areas.

Disk to Tape. Each time a disk storage transaction is processed, the previous contents of the master file are no longer available. This raises the possibility that through machine error, program error or improper data, parts of the file could be destroyed. This problem is not nearly so serious using magnetic tapes, because we can save the tapes from previous cycles and, if necessary, rerun the job. With disk storage, of course, we don't have the previous contents of the file unless steps are taken to make a copy of the file at periodic intervals. This capability is provided by the disk-to-tape routine, whereby the entire file or selected portions of it can be written on magnetic tape. This dumping of file storage can be done in a reasonably short time, and in most applications would be done periodically, perhaps weekly. Now, if through some sort of error the file contents are destroyed, it is necessary only to reload the file from the most recent tape and reprocess all of the transactions that have occurred since then. It is, of course, necessary to save the transactions for this purpose.

Tape to Disk. This routine is the exact analogy of the disk to tape, making it possible to reload the entire disk file or selected portions of it from magnetic tape.

Disk to Card. This program also makes it possible to preserve the contents of disk storage. It is normally used only in systems that do not include magnetic tapes, because the time required to punch cards is much greater than the time required to write on magnetic tape.

Card to Disk. This program is the exact opposite of the disk to card.

In all of these loading and unloading programs the smallest unit of information that can be moved is a single track (2,000 characters).

Chain Loading Program. This program simplifies the initial loading of a disk file when the file is being created. In order to use this system, the programmer must provide an address computation routine that can be used by the loading program. The program loads the master records into disk storage under control of this addressing routine and establishes chains for master records converting to the same disk storage address. Each record in a master chain is located as close to the preceding chain record as possible, thus minimizing access time during disk storage operations. Input records can be on cards or tape.

Chain Additions Program. This program adds new records to a chained file, once again under control of an addressing routine that must be provided by the programmer. The format of the added records must be consistent with that of the records already in the file.

Chain Maintenance Program. This program carries out a number of operations that are required in using a random access file storage system. For instance, when a record is to be deleted from the file the simplest thing to do is to *tag* it by placing a character somewhere in the record to indicate that it is to be deleted. Then the chain maintenance program can be used to actually remove the record from the file and make the record storage locations available for later additions, modifying chains as may be necessary.

The chain maintenance program makes it possible to take advantage of a characteristic of most files. Analysis of many typical files shows that a relatively small fraction of the items account for a relatively large fraction of the total activity. This is sometimes described approximately as the 80-20 rule: 80% of the activity comes from 20% of the records. This being the case, it obviously saves disk access time to place the records having the highest activity at the front of their chains. A simple way to accomplish this is to allow space in the records for a count of the number of times each record is referred to; this count is kept by the application program. The chain maintenance program can then inspect this count and reorganize the chains so that the records most frequently referred to appear early in the chains.

The chain maintenance program can be run periodically whenever time is available, since it keeps track of a portion of the file remaining to be processed.

In use, the chain loading program, the chain additions program and the chain maintenance program are all stored in one part of the disk storage unit so that they can be loaded into core storage through a simple calling procedure. A common plan is to allot some of the lowest-numbered tracks to these service routines.

9.6 Case Study: Wholesale Grocery

The following example based on the data processing requirements of the chain or wholesale grocery operation will serve several purposes. It will provide an example of how a 1401 RAMAC system can be used. At the same time, it will provide an example of how several related data processing activities are frequently combined into one program. Finally, it will illustrate how an ingenious programmer can take full advantage of the equipment available to him by tailoring the machine methods to fit both the equipment and the application. (In this last respect the case study is, in certain details, slightly untypical of disk file methods.) We shall describe the business situation in which this program would be applied, discuss the organization of the program itself, and show a block diagram of the processing. We shall not write a program.

A certain wholesale grocery distributor has an inventory of 5,000 merchandise items which he trucks to 30 stores. An order must be shipped not later than the next working day after he receives it. The order must be accompanied by an invoice.

The order as received shows the items in the sequence in which they appear in the catalog. They are arranged for convenience in making up the order, with similar items grouped together and with the broad classes arranged in the order in which they appear on the shelves in a typical store. There is no way to change this general scheme of catalog arrangement. In the warehouse, however, the merchandise is arranged for ease in making up the order, with the most active items located close to the loading dock, for instance. The invoice which is used by the warehouse to make up the shipment must show the merchandise items in the sequence in which they should be "picked." Thus, the order sequence must be transformed into the picking sequence for printing the invoice. Furthermore, each page of the invoice must show the store name and address.

It is necessary to maintain records of the shipments to each store for billing purposes. It is also necessary to maintain inventory records on all of the items in the warehouse and to print low-stock notices when the balance on hand falls below a minimum point.

The order is fed into the computer on a deck of cards that shows the desired merchandise in terms of page and line numbers in the catalog, along with the quantity desired, store number, and date. Each card refers to one catalog page and shows the quantity desired, for each of up to 50 items. The order will require as many cards as there are catalog pages from which merchandise is ordered. By a special method of card coding, it is possible to specify a quantity of up to 79 for each item.

The disk storage is divided into five sections for this application, as shown in Figure 6.

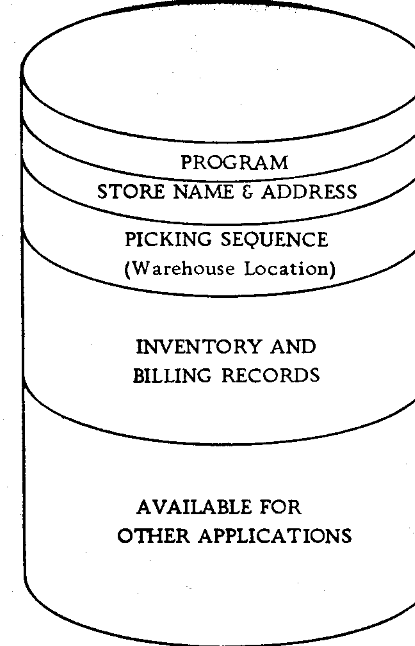

Figure 6. Disk storage organization for the wholesale grocery application described in the text

Store Name and Address File. This file contains the number, name and address of each store customer, along with billing information. As each store order is processed during invoice preparation, the proper store record is selected from the disk file and placed with the order number and the date in core storage. This information is printed on each page of the invoice.

Picking Sequence Table. A picking sequence table is set up in disk storage with an entry for every stock item. This table is in order by page and line number and shows the picking sequence for each item in the warehouse stock.

Billing and Inventory Record. For each item of stock, a billing and inventory record is stored in the disk storage as one 200-character record. These records are arranged in warehouse location order, that is, in picking sequence. The record contains the warehouse location, the picking sequence, catalog page and line number, size, alphabetic description of the item, minimum balance, total sales to date, unit price, balance on hand, and any other information required by the individual customer.

These three files, together with the program for the application, will not completely fill the disk file. The remaining space is available for other applications.

Store orders are processed as they arrive or perhaps in small batches. The store order cards are fed into the 1401 grouped by store and in sequence by page number—the same order in which they are received. An entire order is read and the quantity stored before printing of the invoice begins. This is made necessary by the fact that the catalog sequence and the picking sequence are essentially unrelated. As each order card is read, the picking sequence table for that page is obtained from disk storage. Each line of the order is scanned and whenever a quantity appears, that quantity is stored at a core storage location indicated by the table. If no quantity was punched a zero is stored.

The power of the program organization for this problem depends very much on the use of the core storage picking sequence table. This table must have one character position for each item in the stock. In our case, we assumed 5,000 items and, therefore, 5,000 core storage locations would have to be allocated to the table. (This would, of course, require a larger core storage than that assumed for the rest of this text.) Each item of stock is associated with one character position in this table. The first position in the table is associated with the stock item that should be picked up first if it is present in the order. The second position is associated with the stock item that should be picked up second, and so on through the 5,000 positions.

As each order card is read, the picking sequence table in disk storage is used to determine where in core storage the quantity for that stock item should be stored. When all the order cards have been read, the core storage picking sequence table will contain as many non-zero entries as there are items ordered by the store. This table will contain no identification of the items; this is inherent in the position of each quantity within the table. After all the order cards have been read, it is necessary only to scan through the 5,000-position table looking for non-zero entries and keeping a count of which position of the table is being inspected. Whenever a non-zero character is found, the counter can then be used to compute the address of the corresponding record in the billing and inventory section of the disk storage, which we said was also in picking sequence order.

An example may help to clarify this procedure. Suppose that the warehouse stocks only ten items, to keep the example simple, and that a certain order lists six items. As each item is processed, its picking sequence number is obtained from the disk file. Assume that the items ordered, their quantities, and their picking sequence numbers are as shown in Figure 7. The essence of the scheme is to store the quantity for each item in the position in the core storage picking sequence table corresponding to its picking sequence number. Assuming that the entire core storage picking sequence table is cleared to zeros before the order is processed, our example would produce a picking sequence table as shown in Figure 8, where the table is taken (arbitrarily) to

Page	Line	Quantity	Picking Sequence Number
1	13	20	4
1	34	5	7
2	02	15	2
6	41	2	9
8	12	30	1
8	13	8	5

Figure 7. Illustrative grocery order

Core Storage Location	Quantity
3001	30
3002	15
3003	0
3004	20
3005	8
3006	0
3007	5
3008	0
3009	2
3010	0

Figure 8. Core storage picking sequence table produced from the order of Figure 7

start at 3001. The 30 in position 3001 is now identified with the item shown on page 8 line 12 only by the relative location of the 30 in the table—but this is enough to identify it, since the billing and inventory records are in the same sequence.

What has been done here amounts to sorting the items in the order into picking sequence, by a method known as distribution sorting. It is not typical of disk file applications to do this, but the programmer should always be alert for unconventional ways to do things, if time and expense can be saved.

We may note briefly how it is possible to store a quantity of up to 79 in one core storage position. This merely requires coding the quantity in terms not only of numerical bits, but also the zone bits and the word mark bit. One possible system would be to specify that a word mark bit of 1 stands for a quantity of 40. The B-bit stands for a quantity of 20, and the A-bit thus stands for a quantity of ten. Numerical bits are used in the normal manner to stand for quantities of zero to nine. The following table shows how a few representative quantities would be coded in this scheme.

Quantity	Coding			
	WM	B	A	Numerical
0	0	0	0	0000
10	0	0	1	0000
15	0	0	1	0101
23	0	1	0	0011
39	0	1	1	1001
61	1	1	0	0001
79	1	1	1	1001

A moderately simple program can be used to create these codes as the quantities are read from the order cards, and another program can convert the codes back to normal two-digit quantities when the invoices are prepared.

This use of the word mark bits is definitely not typical, but there is nothing wrong with it. In this case it brings about a saving of 5,000 characters of storage, which in effect makes the whole approach feasible.

To prepare the invoice, we search through the core storage picking sequence table as outlined above. Each time a non-zero item is detected, the corresponding billing and inventory record is obtained from disk storage. The inventory balance is tested for availability. If stock is available, the inventory and sales-to-date balances are updated by the quantity ordered and the updated billing and inventory record is returned

to disk storage. The quantity ordered is multiplied by the price and a billing line printed on the invoice. If an item is out of stock or if a minimum balance has been reached, an appropriate card is punched for information to the buyers. After all items have been recorded on the invoice, a card is punched for the invoice total.

The items now appear on the invoice in picking sequence. All card-sorting operations required by unit record methods have been eliminated by recording the entire order in core storage in picking sequence as the initial step.

A block diagram of the operations in this application is shown in Figure 9.

Exercises

*1. Write a routine to compute a disk address from a seven-digit key by the method outlined in Section 9.4, then read that record into core storage.

2. Using the routine written for Exercise 1, write a routine to handle chaining. Assume that if the transaction key does not match the key in positions 1-10 of the record, the sector address of the next record in the chain appears in positions 180-184 of the record. (Chains may be any number of records long.)

*3. Set up a routine to read a record from an indexed file, as follows. The input key is nine digits long and purely numerical. Obtain a track address by forming the sum of the left three digits, the middle three digits, and the right three digits, then retaining only the last three digits of the sum. This gives the address of a *track;* obtain sector zero of this track, which contains an index of the records stored in the other nine sectors in that track. The index consists of ten-character groups, each group containing a nine-digit key and a one-digit sector number. Write a loop to search through the index, once it is in storage, to find the key in the index that matches the desired key, then use the corresponding sector number from the index to get the address and to read the desired record.

4. A labor distribution problem begins with a deck of cards, each containing an employee number, a number of hours worked, and a job code. You are required to compute the labor cost for each labor voucher, assuming the existence of a file giving the pay rate for each man and assuming no overtime (for this problem). There is also a file containing a record for each job code. You are required to print a line for each job represented in the input deck, showing the total labor cost for the week, and to update the job record to reflect this week's costs.

Outline the method you would follow to carry out these operations, including block diagram.

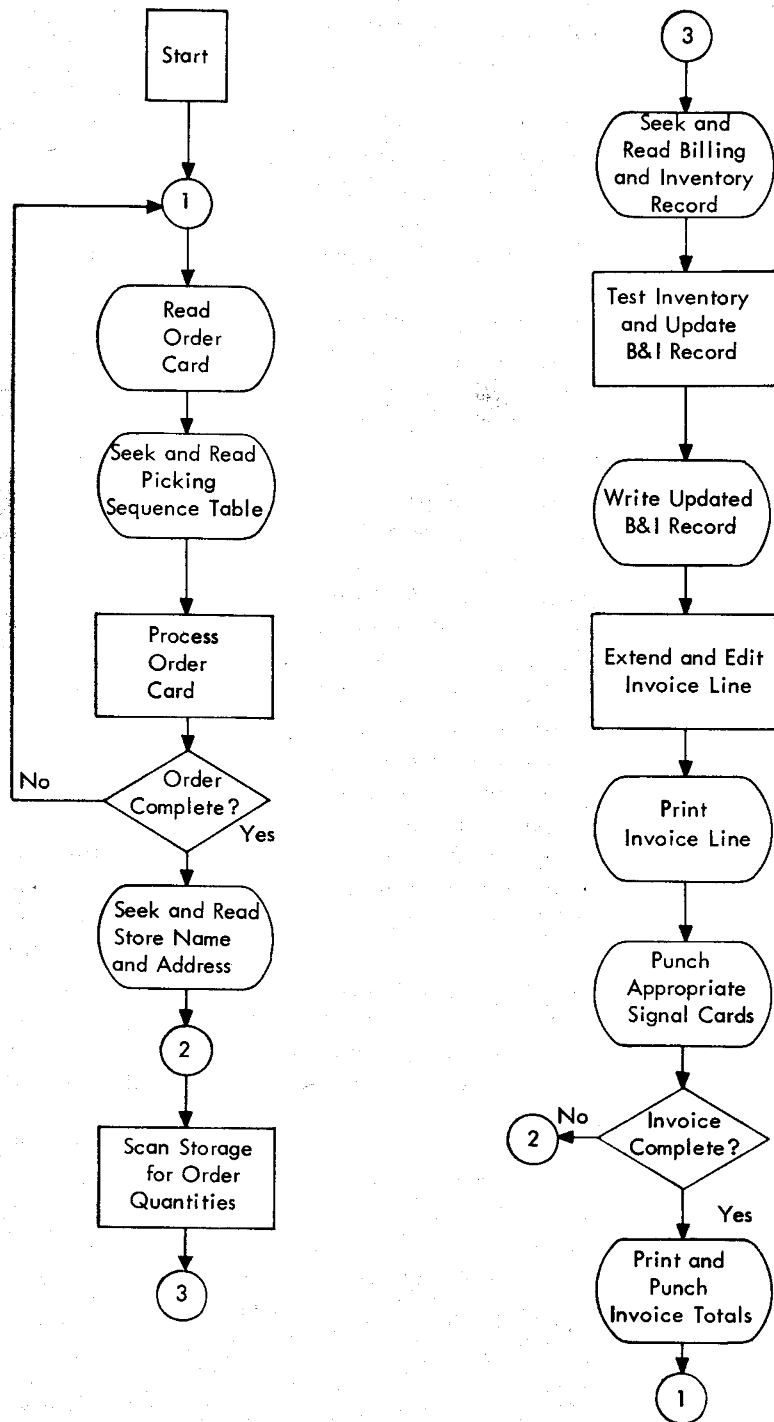

Figure 9. Block diagram of the procedure for the wholesale grocery application