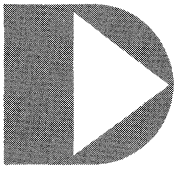
Advanced Techniques in DATASHARE

A Simplified User's Guide

DATAPOINT CORPORATION



Errata Sheet
Advanced Techniques in DATASHARE
A Simplified User's Guide

Page	Revision
6-7	DATASHARE does not allow you to specify an extension to the physical file name in OPEN or PREPARE statements. The extension for a data file is assumed to be, and must be, /TXT.
9	SEQ should be initialized with a FORM statement at the beginning of the program to -1.
11	The third line in the example should be: MATCH "ACE CARDS" TO NAME
12	In both examples, the SEQ INIT "-1" statements should be changed to SEQ FORM "-1"
22	The key used for indexed access must be character data, defined by a DIM or INIT statement, and not numeric data (defined by a FORM statement).
29	The PI instruction cannot be used to prevent interrupts when a KEYIN, DISPLAY, CONSOLE, or PRINT instruction must be executed.

Advanced Techniques in DATASHARE

A Simplified User's Guide

Manual No. 50049-00 November, 1975

Table of Contents

CHAPTER ONE

Introduction

DATASHARE is Datapoint's timesharing system that provides for the simultaneous execution of up to 16 DATABUS programs. The DATABUS language is a very simple and direct high-level business-oriented computing language that is easily suited to a variety of business applications.

All Datapoint languages run under sophisticated, yet easy-to-use operating systems that create standard file formats. Datapoint systems have completely dynamic user-oriented files that eliminate the traditional complexity associated with creating files and managing them. This book explains how to use those files efficiently along with other topics.

This book contains an in-depth coverage of several DATASHARE features, including:

1. Data file access methods (physically sequential, physically random, and indexed).
2. The ANSWER and MASTER control programs.
3. Virtual memory programming considerations.
4. Printing facilities.
5. ROLLOUT procedures and CHAIN files for the execution of DOS commands.

Helpful hints are interjected throughout this book to help you make your DATASHARE system run smoothly, efficiently, and economically.

Prerequisites

This book assumes a basic familiarity with the DATABUS language and the DATASHARE system. But don't be scared off. All of the concepts discussed are carefully illustrated so that even a novice programmer can pick up useful skills that can be incorporated into a DATASHARE operating environment.

This book does not fully explain the DATABUS programming language. For information about the DATABUS language, consult the DATABUS Simplified User's Guide and the DATABUS User's Guide. Appendix A of this book contains an abbreviated listing of all DATABUS instructions.

The details of how to set up and use your DATASHARE system are explained in the DATASHARE User's Guide.

CHAPTER TWO

Effective Data File Utilization

Introduction

All programs that can be executed under DATASHARE must be written in the DATABUS programming language. The DATABUS language lets you create data files on disk and later access the stored data in many ways that will fit your particular application. This chapter will show you how data is organized in those files and how the data can be accessed. Helpful hints will be interjected so you can efficiently use your data files and, consequently, increase the speed of your DATASHARE system.

You should already have an elementary knowledge of how to use data files. However, many DATASHARE users do not know about several simple techniques that can make their programs run faster and make more efficient and effective use of the data in the files. This chapter will show you how to use these techniques. Read on to find out how easy it is to improve the way your DATASHARE programs use data files.

One thing that we're sure you'll discover as you learn how to use data files is just how easy data files are to use. The file access methods are all specifically designed to incorporate speedy access and efficiency of disk space utilization with a minimal amount of programming.

This chapter delves into many details of exactly how the data is written to disk and read from disk. This discussion is essential to a clear understanding of how you, the user, can optimize your file access applications.

The Three Types of Data Files

Data can be arranged in data files in three basic ways:

1. PHYSICALLY SEQUENTIALLY - the data is read in exactly the same order as it is written into the files. This is used in applications where data is to be accessed in chronological (first to last) order.)

2. PHYSICALLY RANDOMLY - the groups of data are numbered, beginning at 0, and any group can be read by specifying its number. This is used in application where a number can easily identify the group of data you want to access.

3. INDEXED SEQUENTIALLY - each group of data is accessed by a unique field of information in that group. This is used when a unique field of information can specify the group of data that you want to access.

With the DATABUS language, it is very easy to effectively use all three of these types of data file structures. The next three sections describe how to use these three types of data

files. If you are unsure of which type of arrangement you want to use for your data, read the beginning of each of these sections.

Physically Sequential Access

The most basic method for storing data is physically sequentially. When data is written to a file physically sequentially, each new record (or group of data) is written immediately after the previous record. When data is read from a file physically sequentially, the records are read in the exact same sequential order that they were written. A special character is inserted between each record to distinguish one record from another.

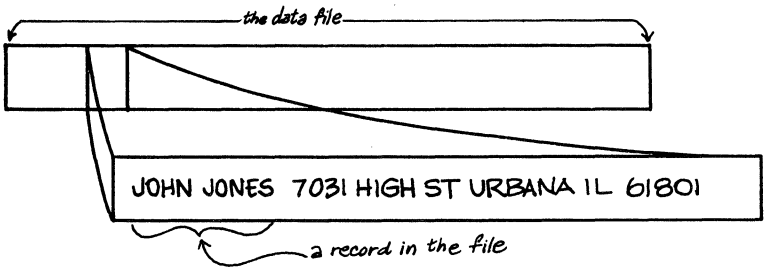
Applications

Physically sequential access is ideal for many applications. You can use it to keep a log file of each day's activities, such as sales orders, in chronological order. Or you can use it to contain a list of names and addresses that are used as labels for regular mailings.

In general, physically sequential access should be used for any data that you want to insert and access in chronological (first to last) order.

What is a Record?

A record is a group of related data. A file is a group of related data records.

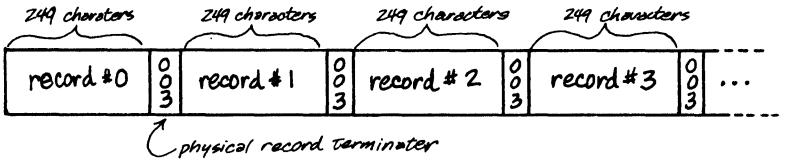


FILES ARE DIVIDED INTO RECORDS

There are two types of records. A physical record is limited to a certain size (249 data characters) and corresponds to a physical sector of the disk where it is stored. A logical record corresponds to a grouping of data that is logically related, regardless of size.

Physical Records

The most basic structure within a file is a physical record. A physical record can contain up to 249 data characters. The end of each physical record is denoted by a 003 character. You do not have to put the 003 character at the end of the physical record; it is automatically put there by DATASHARE.



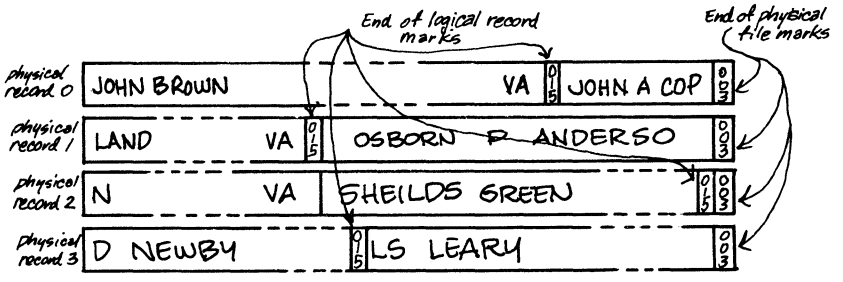
PHYSICAL RECORDS CONTAIN 249 CHARACTERS

With physically sequential access, you never need to pay attention to physical record boundaries. What is important to you are logical records.

Logical Records

The next level of structuring is the logical record. A logical record is a grouping of data that is logically related. WRITES and READS are based on logical record boundaries. There is no limit to the length of a logical record. Logical records are terminated by a 015 character. As with physical records, DATASHARE supplies this character for you.

Physically sequential WRITES use Datapoint's record compressed file structure, where logical records span physical record boundaries, as in the following illustration:

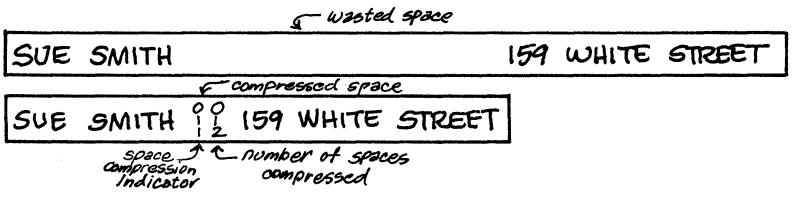


WITH PHYSICALLY SEQUENTIAL WRITES, LOGICAL RECORDS SPAN PHYSICAL RECORD BOUNDARIES

This record compressed structure conserves space by packing the logical records closely together. Datapoint has another technique for conserving space. This technique is called space compression, and is also performed automatically for you.

Space Compression

When DATABUS finds two or more blank spaces in the data that is to be written to a disk file, it uses space compression to save space on the disk. For example, if it encounters "123 JOHN ST.", it converts it to "123(011)(005)JOHN ST.", where (011) is the space compression indicator and (005) is the number of spaces that are compressed. In this way, only two bytes (character positions) are used to store the spaces instead of five.



DON'T WASTE DISK SPACE -- COMPRESS IT

Space compression is always turned on with physically sequential writes unless you are also writing to the same physical file using physically random or indexed insertion writes. If you are mixing access methods within the same physical file, you can make sure that space compression is turned on with the "*+" control signal in the physically random and indexed WRITE statements, as in the following illustration:

```
WRITE FILE1,KEY;*+,NAME,ADDRESS,CITY;
WRITE FILEA,SEQ;STATE,ZIP
```

IF YOU MIX ACCESS METHODS, YOU MUST TURN ON THE SPACE COMPRESSION

You must turn on space compression with the "*+" signal in every physically random or indexed insertion WRITE or space compression will be turned off. All physically sequential READ statements properly expand the space compressed notation.

NOTE: With other access methods, you should not do updates on space compressed records because the total number of characters may not be the same in one space compressed record as in another.

The Physical File Name

The physical file name is the name that corresponds to the actual name the data file has on disk. This name can be up to eight characters long (the first character must be alphabetic) and is followed by a slash and a three-character extension that specifies the type of file that it is.

Data files usually have the extension of /TXT. This is the same extension that is used for any file that has not yet been processed by a compiler. Your DATABUS program source code (the file you created with the DATABUS language instructions) also has the extension of /TXT. When you compile your DATABUS program, however, you get another file that is named the same as the source file but has an extension of /DBC. This /DBC file is the one that is actually executed under DATASHARE. Each time you recompile your program, you wipe out the old /DBC file and create a new one.

THE PHYSICAL FILE NAME

DATABUS assumes that the extension for your physical data file name will be /TXT. This means that if you do not include an extension, /TXT is assumed. In other words, ORDERS/TXT and ORDERS specify the same data file.

Logical File Names

The logical file name is the name that is associated with the physical file name. It is used throughout the program to reference the physical file specified in the OPEN or PREPARE statement.

In every DATABUS program, the logical file name must be declared to be a logical file name through the use of the FILE statement. For example, to use the logical file name FILE1 throughout the program, you must include this FILE statement at the beginning of the program:

```
FILE1 FILE
```

Then the logical file name declared in the FILE statement must be associated with a physical file name through the use of a PREPARE (for new files) or OPEN (for existing files) statement.

For example, to create a new file named ORDERS/TXT and use the logical name FILE1 to access it, you have to use the following two statements:

```
FILE1     FILE
          PREPARE FILE1,"ORDERS"
```

(the /TXT extension specifies a text file and is assumed if not included in the PREPARE or OPEN statement). Once the file is created, it must be used with the OPEN statement. The following statements reference the existing file ORDERS/TXT and associate with it the logical name FILE1:

```
FILE1     FILE
          OPEN FILE1,"ORDERS"
```

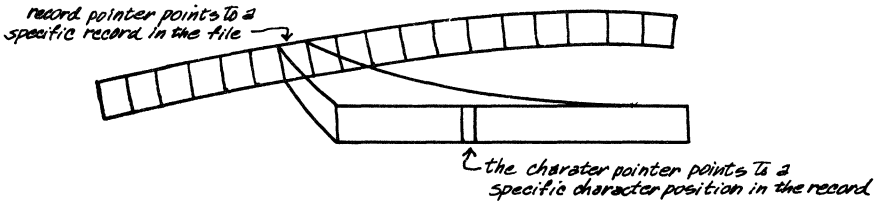
FILE1 is the logical name that is used in all READ and WRITE statements. Because the OPEN or PREPARE statements associate this name with a physical file (ORDERS/TXT), DATASHARE knows where to go to physically read or write the data.

Why does DATABUS insist on two names, one logical and one physical, for the same file? It is actually done as a convenience to the programmer. If, for example, a different physical file is used every month, yet the same program is used, the programmer only has to change the physical name specified in the OPEN or PREPARE statement. The logical file name can remain

the same. This saves a lot of time because the programmer does not have to change every line in the program that refers to that file.

Positioning and Accessing

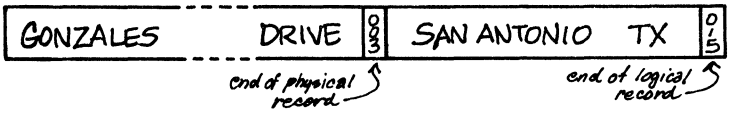
The current position in a data file is defined by two pointers, the physical record pointer (0 through the number of records in the file) and the character pointer (1 through 249). These pointers are kept internally by DATASHARE and are used by all of the access methods.



RECORD AND CHARACTER POINTERS KEEP TRACK OF THE CURRENT LOCATION IN THE FILE

When the file is opened (with an OPEN or PREPARE statement), the physical record pointer is set to 0 and the character pointer is set to 1.

All READ and WRITE operations sequentially increment the character pointer as the individual characters are read or written. If the physical record terminator (003) is reached during a READ or if the 249th character is written during a WRITE, the physical record pointer is incremented by one, the character pointer is reset to 1, and the logical record is continued on to the next physical record.

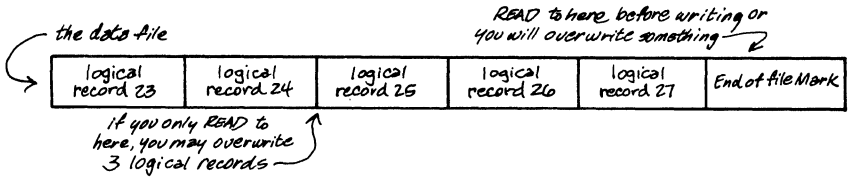


LOGICAL RECORDS SPAN PHYSICAL RECORD BOUNDARIES

Physically Sequential WRITE Statements

You have already learned that you have to declare a logical and physical file name for your data file with the FILE and OPEN or PREPARE statements. You have also learned how space is compressed and records are compressed to save space on the disk. And you have learned how DATABUS keeps track of the physical position in a data file through record and character pointers. Now you will learn how to write data into a file.

When you PREPARE or OPEN your file, you are positioned at the first record. This is fine if it's a new file, but if you want to add data to an existing file, you must read to the end-of-file and then write. Remember to do this or you may overwrite valuable data.



BE CAREFUL NOT TO OVERWRITE VALUABLE INFORMATION

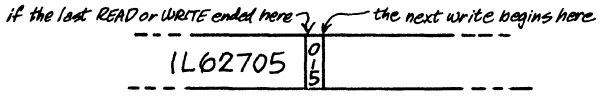
As we've stated before, physically sequential WRITE statements pay no attention to physical record boundaries. Each WRITE writes one logical record to disk unless a semicolon ends the list of variables. In this case, the logical record is not ended with that statement. The format for a physically sequential WRITE is:

WRITE filename,SEQ;variables

where filename is the logical name of the file, SEQ should be initialized at the beginning of the program to -1, and variables are the data that should be written to disk. SEQ is the variable that indicates that this is physically sequential access. For example:

WRITE FILE1,SEQ;NAME,ADDRESS,CITY,STATE

writes the values of NAME, ADDRESS, CITY, and STATE as one logical record in logical FILE1 wherever the previous READ or WRITE left off. After STATE is written, the 015 mark is written denoting the end of the logical record. This logical record will span as many physical records as necessary to hold the data.



PHYSICALLY SEQUENTIAL WRITES SIMPLY PICK UP FROM WHERE THE LAST READ OR WRITE ENDED

To avoid writing the end of logical record mark at the end of the list of variables, conclude the list with a semicolon (;) in the following manner:

WRITE FILE1,SEQ;NAME,ADDRESS,CITY,STATE;

Data from the next WRITE statement will then be included in the same logical record as this data.

Under most circumstances, you should avoid using the semicolon at the end of the WRITE statement. Try to get all of the data for each logical record ready before you write it to disk. More than one WRITE statement for a logical record wastes time because the disk head must be positioned twice for the same

logical record.

You can include a 015 in the list of variables to signify the end of a logical record if you want to write more than one logical record with a single WRITE statement. Other control characters include *ZF, which is used before numeric variable to cause it to be right justified and zero filled on the left, or *MP, to convert a numeric variable to a "minus over-punch" format. For example:

```
WRITE FILE2,SEQ;A,B,C,015,D,E,*ZF,F
```

writes two logical records in as many physical records as necessary. The first logical record consists of the values for the variables A, B, and C. The second logical record contains the values for D, E, and F. F must be a numeric variable because it will be zero filled on the left.

Writing the End-of-File Mark

After you are done writing a physically sequential file, you should write an end-of-file mark. Be sure you are positioned at the end of the file (the last data record has just been written) or you will write the end-of-file mark in the middle of your data. The WEOF instruction, which will write the mark, has the following format:

```
WEOF file,SEQ
```

where file is the name of the logical file.

Then use the CLOSE instruction to release any extra allocated space.

If you later add data to your file, you must READ to end-of-file, then WRITE your data. You should write another end-of-file mark when you are done adding data to your file, and then CLOSE it.

Physically Sequential READ Statements

To read a physically sequential file, the READ statement is used. It works much the same as a WRITE statement. All READ statements for a file must either come before the WRITE statements for the file, or after the end-of-file mark has been written, the file has been closed, and the file has been reopened.

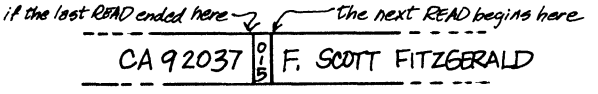
The READ statement reads from where the record and character pointers are positioned to the end of the logical record or until the list of variables is satisfied. Unless a semicolon (;) ends the list of variables, the record and character pointers are left pointing to the beginning of the next logical record. Physical record boundaries are ignored, for all practical purposes. Remember to initialize SEQ to -1 at the beginning of the program to signify that you are using physically sequential access. The following statements give examples of physically sequential READS and explain how they operate.

```
READ FILE1,SEQ;NAME,ADDRESS,CITY,STATE
```

NAME, ADDRESS, CITY, and STATE are read from logical FILE1. The character pointers are left pointing to the following logical record.

```
READ FILE1,SEQ;NAME,ADDRESS,CITY,STATE;
```

This is the same as above example except that the character pointers are left pointing to the next character after STATE.



PHYSICALLY SEQUENTIAL READS SIMPLY PICK UP FROM WHERE THE LAST READ ENDED

A Few Hints About Reading

Physically sequential data files are characterized by the fact that to get any information out of the file, you must start at the beginning and READ until you find it. The COMPARE and MATCH instructions provide useful methods of checking to see if you're reading the record that you need. COMPARE is used to make sure that two numbers are the same and MATCH is used to make sure that two character strings are the same. So, for example, to check to see if you're reading the record for the Ace Cards Co., you could use the following statements:

```
LOOP    READ FILE1,SEQ;NAME,ADR,BILL
        GOTO BAD IF OVER
        MATCH NAME TO "ACE CARDS"
        GOTO GOOD IF EQUAL
        GOTO LOOP
BAD     DISPLAY "**NO SUCH NAME**"
        GOTO START
GOOD    DISPLAY ADR,*N,"OWES:$",BILL
        GOTO START
```

Notice how we checked for the OVER condition. This is a safeguard against not finding a match. If end-of-file is reached, the OVER condition is set. By checking this condition, you can tell if the entire file has been read and checked.

Physically Sequential Program Examples

The following DATASHARE program writes each account number, name, and address to a disk file named ORDERS/TXT:

```
DONE    INIT "00000"
ACNT    DIM 5
NAME    DIM 20
ADR     DIM 20
CITY    DIM 15
STATE   DIM 2
ZIP     DIM 5
SEQ     INIT "-1"
AFILE   FILE
        PREPARE AFILE,"ORDERS"
LOOP    KEYIN *ES,"ACCOUNT NUMBER:",ACNT:
           *N,",",NAME:",NAME,*N,"ADDRESS:",ADR:
           *N,"CITY:",CITY,*N,"STATE:",STATE:
           *N,"ZIP:",ZIP
        MATCH DONE TO ACNT
        GOTO FIN IF EQUAL
        WRITE AFILE,SEQ;ACNT,NAME,ADR,CITY,STATE,ZIP
        GOTO LOOP
FIN     WEOF AFILE,SEQ
        CLOSE AFILE
        STOP
```

This program asks the operator to fill in all the data fields, one at a time. Note how one KEYIN statement is used to collect all the data. It is more efficient to use one long KEYIN or DISPLAY statement than many short ones because of the way DATASHARE handles those instructions. All input and output statements should be written this way, with one long statement rather than several short ones.

The following program reads the SAMPLE data file and prints it. Notice how the OVER condition is checked to see if the end-of-file has been reached.

```
ACNT    DIM 5
NAME    DIM 20
ADR     DIM 20
CITY    DIM 15
STATE   DIM 2
ZIP     DIM 5
SEQ     INIT "-1"
AFILE   FILE
        OPEN AFILE,"SAMPLE"
LOOP    READ AFILE,SEQ;ACNT,NAME,ADR,CITY,STATE,ZIP
        GOTO FIN IF OVER
        PRINT *N,"ACCOUNT NUMBER",ACNT,*N,*10,NAME:
           *N,*10,ADR,*N,*10,CITY,*37,STATE,*40,ZIP
FIN     RELEASE
        STOP
```

Physically Random Access

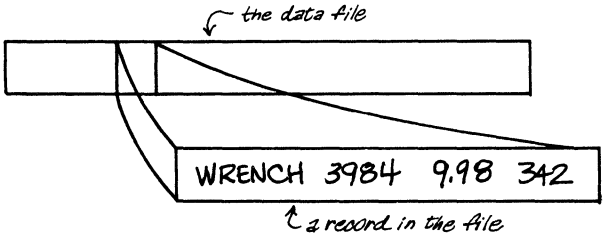
The fastest random access method available under DATASHARE is physically random access. To perform a physically random access, a number is used to point to the specific group of data that you want to access. Physical random access is best if used with groups of data (called records) that contain less than 249 characters.

Applications

Use physically random access for any grouping of data that has a natural sequential numbering scheme to it. For example, if a hardware store has parts that number from 1 to 137, each part can be described in a separate record and accessed by part number. Or, if you have orders that number from 900 to 1000, you can put each order in a separate record and subtract 900 from the order number to get the record number.

What is a Record?

A record is a group of related data. A file is a group of related data records.

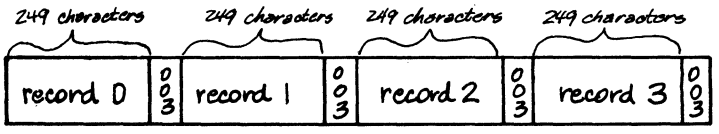


FILES ARE DIVIDED INTO RECORDS

There are two types of records. A physical record is limited to a certain size (249 data characters) and corresponds to a physical sector of the disk where it is stored. A logical record corresponds to a grouping of data that is logically related, regardless of size.

Physical Records

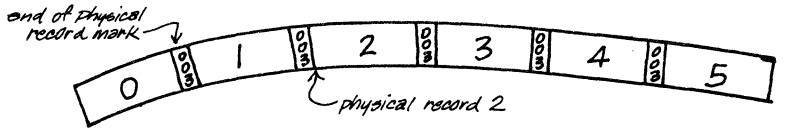
The most basic structure within a file is a physical record. A physical record can contain up to 249 data characters. The end of each physical record is denoted by a 003 character. You do not have to put the 003 character at the end of the physical record; it is automatically put there by DATASHARE.



PHYSICAL RECORDS CONTAIN 249 CHARACTERS

With physically random access, the physical record number (numbering starts at zero) is specified in the READ or WRITE statement to point to a record of data to be read or written.

NOTE: The physical record number never needs to be written in the data file. Because all physical records are written sequentially, DATABUS knows which record corresponds with a particular physical record number.

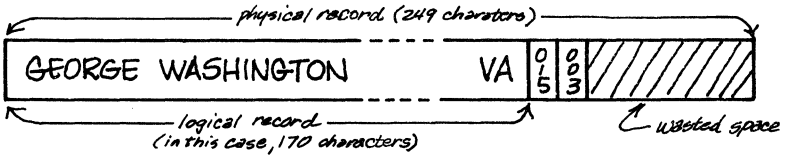


A NUMBER POINTS TO THE PHYSICAL RECORD IN THE FILE

Logical Records

The next level of structuring is the logical record. A logical record is a group of logically related data. For most practical purposes, each logical record should contain 249 or less characters so it will fit in one physical record. Logical records are terminated by a 015 character.

Whatever space in the physical record is not taken up by the logical record is wasted because the physically random access method relies on physical record boundaries. This means that physically random files are NOT record compressed, as are physically sequential files.



EXTRA PHYSICAL RECORD SPACE IS WASTED

Carefully Structure Your Files

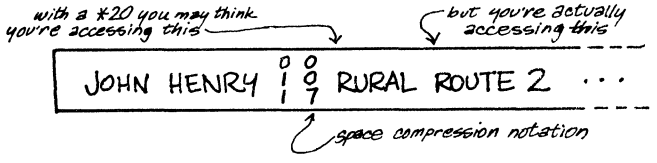
Because the random records are accessed by a number indicating the physical record in the file, it is important to write the data so that logical records begin at physical record boundaries. If you use the physically random WRITE statements, you will automatically start each logical record at a physical record boundary. Do not use the REFORMAT utility on your data file and put more than one logical record per physical record, because when you later access the data it will not be in the proper order and logical records will not begin on physical record boundaries.

If, for some reason, your data file becomes record compressed so that each logical record does not begin on a physical record boundary, you can use the REFORMAT utility to reblock your data (see the REFORMAT section of this chapter). If you edit your physically random file using the DATASHARE

Editor or the DOS Editor, you will have to REFORMAT the file to get it back to the proper format with logical records starting at physical record boundaries.

Space Compression

Space compression (explained fully under Physically Sequential Access) is automatically turned off for physically random WRITE statements. This is because physically random WRITAB and READ statements can use absolute tabbing, which does not properly expand the space compressed notation. See the WRITAB and READ sections for an explanation of absolute tabbing.



ABSOLUTE TABBING OPERATIONS DO NOT PROPERLY EXPAND SPACE COMPRESSION

If you are sure that you will not want to use tabbing, you can begin each WRITE statement with a "*+" control signal to signifiy that space compression for that statement should be turned on. The following example illustrates how space compression can be turned on for a physically random WRITE:

```
WRITE FILE2,NUM;*+,NAME,ADR
```

If you find that your file is space compressed and shouldn't be, you can use the REFORMAT utility to get rid of the space compression (see the REFORMAT section of this chapter).

The Physical File Name

The physical file name is the name that corresponds to the actual name the data file has on disk. This name can be up to eight characters long (the first character must be alphabetic) and is followed by a slash and a three-character extension that specifies the type of file that it is.

All data files usually have the extension of /TXT. This is the same extension that is used for any file that has not yet been processed by a compiler. Your DATABUS program source code (the file you created with the DATABUS language instructions) also has the extension of /TXT. When you compile your DATABUS program, however, you get another file that is named the same as the source file but has an extension of /DBC. This /DBC file is the one that is actually executed under DATASHARE. Each time you recompile your program, you wipe out the old /DBC file and create a new one.

THE PHYSICAL FILE NAME

DATABUS assumes that the extension for your physical data file name will be /TXT. This means that if you do not include an extension, /TXT is assumed. In other words, ORDERS/TXT and ORDERS specify the same data file.

Logical File Names

The logical file name is the name that is associated with the physical file name. It is used throughout the program to reference the physical file specified in the OPEN or PREPARE statement.

In every DATABUS program, the logical file name must be declared to be a logical file name through the use of the FILE statement. For example, to use the logical file name FILE1 throughout the program, you must include this FILE statement at the beginning of the program:

```
FILE1     FILE
```

Then the logical file name declared in the FILE statement must be associated with a physical file name through the use of a PREPARE (for new files) or OPEN (for existing files) statement.

For example, to create a new file named ORDERS/TXT and use the logical name FILE1 to access it, you have to use the following two statements:

```
FILE1     FILE
          PREPARE FILE1,"ORDERS"
```

(the /TXT extension specifies a text file and is assumed if not included in the PREPARE or OPEN statement). Once the file is created, it must be used with the OPEN statement. The following statements reference the existing file ORDERS/TXT and associate with it the logical name FILE1:

```
FILE1     FILE
          OPEN FILE1,"ORDERS"
```

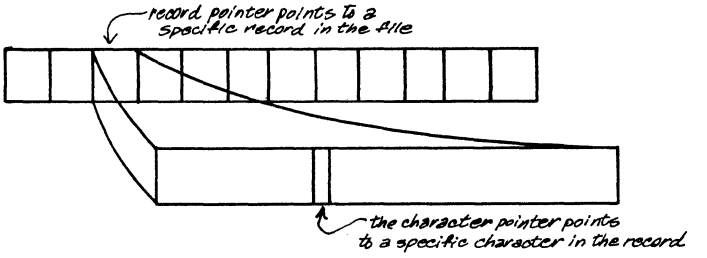
FILE1 is the logical name that is used in all READ and WRITE statements. Because the OPEN or PREPARE statements associate this name with a physical file (ORDERS/TXT), DATASHARE knows where to go to physically READ or WRITE the data.

Why does DATABUS insist on two names, one logical and one physical, for the same file? It is actually done as a convenience to the programmer. If, for example, a different physical file is used every month, yet the same program is used, the programmer only has to change the physical name specified in the OPEN or PREPARE statement. The logical file name can remain

the same. This saves a lot of time because the programmer does not have to change every line in the program that refers to that file.

Positioning and Accessing

The current position in a data file is defined by two pointers, the physical record pointer (0 through the number of records in the file) and the character pointer (1 through 249). With physically random access, the physical record pointer corresponds to the record pointer specified in the READ or WRITE statement. These pointers are kept internally by DATASHARE and are used by the access method.

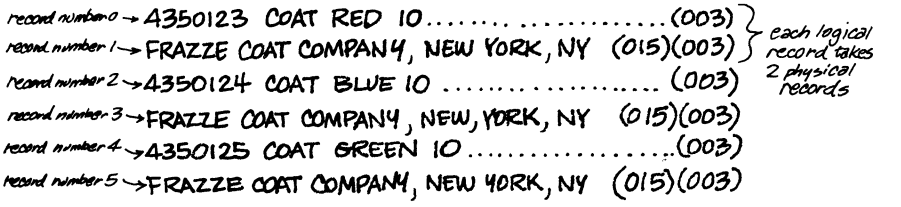


RECORD AND CHARACTER POINTERS KEEP TRACK OF THE CURRENT LOCATION IN THE FILE

When the file is opened (with an OPEN or PREPARE statement), the record pointer is set to 0 and the character pointer is set to 1.

All READ and WRITE operations sequentially increment the pointers. If more than 249 characters are written with a single WRITE statement, the record pointer is incremented by one and the character pointer is reset to 1. All physically random READs and WRITEs begin on physical record boundaries, with the character pointer set at 1.

What if physical random records exceed 249 characters? They are continued on to the next physical record. The following random record will begin on the following physical record boundary. If this is the case for your random records, be sure to recipricate for this by properly incrementing the record number used in READs and WRITEs



SAMPLE RANDOM FILE WITH TWO PHYSICAL RECORDS PER LOGICAL RECORD

For the above example, you would use every other physical record number, starting at zero, to access the logical records.

READ and WRITE Statements

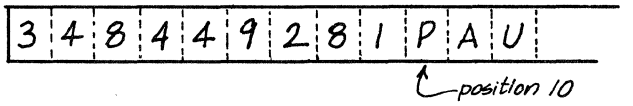
To read or write a physically random access file, use the following READ or WRITE statements:

```
READ filename,number;variables
WRITE filename,number;variables
```

where filename is the logical name of the file, number is a variable containing the number of the physical record to be accessed, and variables are the data to be read or written.

There are two control characters that are used in physically random WRITES. These are *ZF, which is used before a numeric variable to cause it to be right justified and zero filled on the left, or *MP, to convert a numeric variable to a "minus over-punch" format.

Tabbing controls can be added to the list of variables to be read so that selected character positions may be read from a record without having to read all of the positions in the record. To use tabbing, precede the column number or variable containing the column number with an asterisk (*). For example, *10 will tab to the tenth character position in a physical record.



*10 TABS TO THE TENTH CHARACTER POSITION IN A PHYSICAL RECORD

CAUTION: Tabbing will not properly expand space compressed records.

The following examples show how to use the READ and WRITE statements:

```
WRITE FILE3,NUM;NAME,NUMBER,CODE
```

This writes physical record number NUM. The values for NAME, NUMBER, and CODE are written. An end of logical record mark (015) and end of physical record mark (003) is written after CODE.

```
WRITE FILE3,NUM;NAME,NUMBER,CODE;
```

This writes physical record number NUM. The values for NAME, NUMBER, and CODE are written. No end of record marks are written. The next WRITE will continue to fill up the same logical record.

```
READ FILE3,NUM;NAME,*40;CODE
```

This reads physical record number NUM. The values for NAME and CODE are read. Any data between the end of NAME and column 40 is ignored. File pointers then point to the beginning of the

next logical record.

```
READ FILE3,NUM;NAME,NUMBER;
```

This reads NAME and NUMBER from physical record number NUM. File pointers are left pointing to the next character position following NUMBER.

```
READ FILE3,ZERO;;
```

Assume that the numeric variable ZERO is defined to be zero in value. This statement causes the file pointers to be positioned to the physical beginning of the file exactly as if an OPEN statement had been executed.

The WRITAB Instruction

The WRITAB instruction allows you to write characters into any character positions of a physical record without disturbing the rest of the record. The record must already exist. To use tabbing, precede the column number with an asterisk (*). The following variable is written starting at the character position specified. If no positioning is specified, the writing starts at the beginning of the physical record. For example:

```
WRITAB FILE4,NUM;A,*70,B,*10,C,*NAME,"NAME"
```

writes the value for A at the beginning of the record, B at column 70, C at column 10, and "NAME" at the position specified by NAME. Note that WRITAB, like other random access WRITEs, does not use space compression. It will not properly expand the character positions of space compressed records.

Writing the End-of-File Mark

After you have written your random access file, or if you have added data to the file, you must add an end-of-file (EOF) mark after the last record. To do this, increment the record counter variable to one past the last record written, and then use the WEOF instruction. The WEOF instruction has this format:

```
WEOF file,NUM
```

where file is the logical file name and NUM is the record counter variable.

Then use the CLOSE instruction to close the data file.

To add data to the physically random file, increment the record counter variable to one past the last record written (this corresponds to the end-of-file record) and begin to write. Because you've overwritten the end-of-file mark, a new one must be written when you are done adding data to your file.

Random Access Program Examples

In the following DATABUS program, all of the characteristics of a company's parts are inventoried in a

physically random file. Record 0 describes the file. This is a particularly appropriate application for a physically random file because the parts are numbered from 1 to 79.

This program creates a file named PARTS/TXT and writes the first logical and physical record that describes the file. Then the user is requested to keyin a description, price, and quantity for each part in chronological order. Once the 79 part descriptions have been entered, an end-of-file mark is written. The variable PARTNO contains the number of the physical record for each entry.

```
PARTNO    FORM " 1"
ONE       FORM "1"
ZERO      FORM "0"
DESC      DIM 15
PRICE     FORM 3.2
QUAN      FORM 5
NPART     FORM "80"
PARTS     FILE
          PREPARE PARTS,"PARTS"
          WRITE PARTS,ZERO;"PART INVENTORY"
LOOP      DISPLAY *ES,"PART NUMBER",PARTNO
          KEYIN *N,"DESCRIPTION:",DESC,*N,"PRICE:$"
             PRICE, *N,"QUANTITY:",QUAN
          WRITE PARTS,PARTNO;PARTNO,DESC,PRICE,QUAN
          ADD ONE TO PARTNO
          COMPARE PARTNO TO NPART
          GOTO DONE IF EQUAL
          GOTO LOOP
DONE      DISPLAY *ES,"THANKS"
          WEOF PARTS,PARTNO
          CLOSE PARTS
          STOP
```

This DATABUS program, when executed, creates a file named PARTS/TXT. The first nine records might have been filled in to resemble the following:

```
PART INVENTORY
1WRENCH          2.95      300
2WRENCH          3.95      300
3WRENCH          4.95      400
4WRENCH          6.50      100
5WRENCH          7.95      300
6WRENCH          9.95      400
7HAMMER          4.95      300
8HAMMER          6.95      600
```

To see what quantity of a specific part exists, and possibly change that value, the following program was written. This progam asks the user for the number of the part that he wants checked. The program reads that record and displays the quantity inventoried. The user is given the opportunity to change the number of parts inventoried. Then the user is asked

if he is finished changing the file.

Note the use of the OVER condition check. If a record cannot be found, the OVER flag is set. The checking of the OVER flag is an easy way to check if the part number specified is included in the file.

```
PARTNO    FORM 2
QUAN      FORM 3
CHANGE    DIM 1
YES       INIT "Y"
DONE      DIM 1
NQUAN     FORM 3
PARTS     FILE
          OPEN PARTS,"PARTS"
LOOP      KEYIN *ES,"QUANTITY CORRECTION PROGRAM":
             *P1:3,"PART NUMBER:"PARTNO
          READ PARTS,PARTNO;*27,QUAN
          GOTO ERROR IF OVER
          DISPLAY *P1:5,"EXISTING QUANTITY IS",QUAN
          KEYIN *P1:6,"DO YOU WANT TO CHANGE IT?"
             CHANGE
          MATCH CHANGE TO YES
          GOTO FIX IF EQUAL
END       KEYIN *P1:8,"ARE YOU DONE?",DONE
          MATCH DONE TO YES
          GOTO FIN IF EQUAL
          GOTO LOOP
ERROR     DISPLAY *P1:9,"**BAD PART NUMBER**"
          GOTO LOOP
FIX       KEYIN *P1:7,"NEW VALUE:",NQUAN
          WRITAB PARTS,PARTNO;*27,NQUAN;
          GOTO END
FIN       CLOSE PARTS
          DISPLAY *P1:9,"THANKS"
          STOP
```

Indexed Access

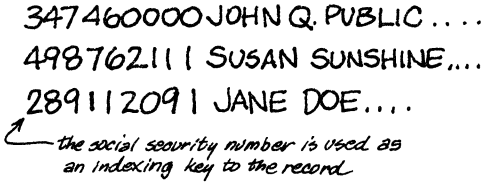
When you use indexed access, you use two files. One file is the actual data file. The other file contains the index structure. Because data files may be accessed in many different ways, a data file can have several index files associated with it.

You can create the data file using physically sequential or random access WRITE statements. Once the file is created, you use the DOS INDEX command to create the index files. Because indexing is based on fields in logical records, the data file can be record and space compressed. In this section, the word record refers to a logical record. The data in each logical record can be space compressed and the fields will be properly interpreted.

Indexing is based on the standard ASCII collating sequence. See Appendix B for a chart of ASCII values.

Indexing is Based on Keys

Indexing is based on a key. A key is a field of data in each record. Each key must be unique. The key can be alphabetic or numeric. In the following example, the first nine digits in each field represent the person's social security number and is used as a key to that record.



A KEY IS A FIELD OF DATA

Before you write the data file, you should consider the best arrangement for the data so the key choice can easily be made. The key must be in the same position in each logical record.

The INDEX Utility

To use the INDEX utility, you must ROLLOUT to DOS and issue the INDEX command at the console or include it in a CHAIN file. See Chapter Six for an explanation of ROLLOUT and examples of its use. The INDEX command has this format:

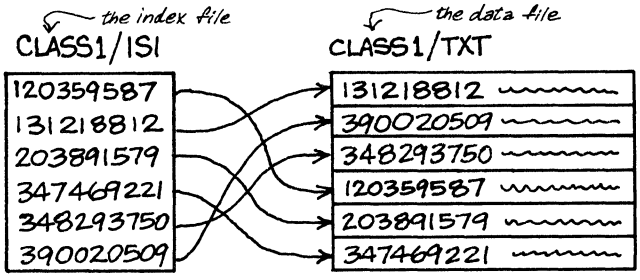
```
INDEX input,output;key
```

where input is the name of the data file, output is the name of the index file that INDEX will create, and key specifies the field that is used as the index. If an output file name is not specified, the input file name will be used with an /ISI extension.

For example, if the data file given in the previous example is named CLASS1/TXT, then this command:

```
INDEX CLASS1;1-9
```

creates CLASS1/ISI, an index file containing the index structure based on the social security numbers at the beginning of each record.



THE INDEX FILE CONTAINS POINTERS TO INDEX ENTRIES IN THE DATA FILE

You can create several index files for the same data file by using the INDEX utility on different fields and specifying different output files. For example, these commands:

```
INDEX CLASS1,CLASS1A;1-9
INDEX CLASS1,CLASS1B;24-28
```

create two index files for the CLASS1 data file. CLASS1A/ISI contains the index structure based on the characters in the first nine positions in the data file. CLASS1B/ISI contains the index structure for the data in positions 24 through 28. When you want to read or write based on the key declared in CLASS1A/ISI, you open CLASS1A/ISI, which automatically points to CLASS1/TXT, the data file. When you want to use the index in CLASS1B/ISI, you open CLASS1B/ISI, which also points to CLASS1/TXT, the data file.

The Five Indexed Operations

Once the index is created for a data file, there are five basic indexed operations you can perform on a file. These operations are:

1. Reading a record of a given key value
2. Reading the record of the next sequential key value
3. Updating the record that was last accessed through the index
4. Inserting a new record of a given unique key value
5. Deleting a record of a given key value

All of these operations are discussed in the following sections.

Logical and Physical Files

For indexed files, the IFILE declaration must be used to declare that a file is an indexed file. The following statement declares that FILEA is the logical name for an indexed file:

```
FILEA     IFILE
```

To associate this logical file name with an existing physical index file, you must use the OPEN statement. The

statement:

```
OPEN FILEA,"CODES"
```

associates the logical name FILEA with the existing index file CODES/ISI. (In this case, the /ISI extension is assumed if an extension is not specified because this is an index file declaration). CODES/ISI is the index structure that points to the CODES data file.

Indexed READ Statements

Indexed READ statements use this format:

```
READ file,key;variables
```

where file is the logical file name (declared as an IFILE), key is a variable containing the characters you are looking for in the index, and variables are the list of data items. The index file is searched for the key given in the string variable key. The key is considered to match an item in the index file if both have exactly the same number of characters and all of them match or if all of the characters up through the length of the index item match and then the rest of the characters in the key variable are spaces.

Remember that there are no trailing spaces in the index key items. This means that even if the INDEX utility is told to index on column 1 through 9, and if that field in a certain record consists of only one character followed by eight spaces, the index file key item would consist of that one character followed by the key terminator character.

If a match is found, the record containing the matched key is read from the beginning, including the key value.

If no match is found, the OVER condition flag is set. It is, therefore, a good idea to include a statement similar to the following, which transfers control to label BAD, where an error message is displayed:

```
GOTO BAD IF OVER
```

The following READ statements show how indexed reads can be performed:

```
READ FILE,KEY;KEY,NAME,ADDRESS
```

This reads the record containing KEY, and since KEY is the first variable in the record, reads KEY along with NAME and ADDRESS.

```
READ FILE,KEY;KEY,NAME,ADDRESS;
```

This is similar to the above operation, but saves time by not scanning to the end of the logical record containing the KEY.

```
READ FILE,NULL;KEY,NAME,ADDRESS
```

Assuming that NULL is given a null value, this is an indexed re-read. This re-reads the last logical record that was accessed.

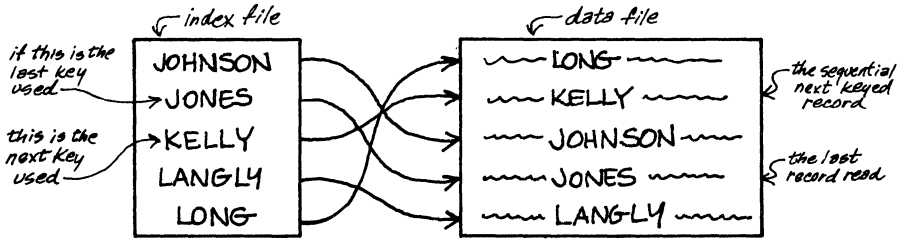
```
READ FILE,KEY;*20,NAME;
```

This reads the record containing KEY and reads data into the variable NAME starting at character position 20. The semicolon is used because it is not necessary to read any other data in the record at this time.

CAUTION: The tabbing facility illustrated above, like all DATASHARE tabbing facilities, will not properly expand space and record compressed files.

The READKS Instruction

All of the key pointers are stored sequentially, in ASCII collating sequence, in the index file (see Appendix B for a list of ASCII values). The READKS instruction is used to read the record pointed to by the next sequential key entry in the index file. The following illustration shows how READKS can be used:



READKS READS THE SEQUENTIALLY NEXT KEY IN THE INDEX FILE

The READKS instruction follows this format:

```
READKS file;variables
```

where file is the name of the logical file and variables are the data that is to be read.

For example, this READKS statement can be used:

```
READKS FILE;*20,NAME
```

This reads the variable NAME from the 20th character position in the next sequential keyed record.

Indexed WRITE Statements

To do indexed insertion writes, use the WRITE statement, which has this format:

```
WRITE file,key;variables
```

where file is the name of the logical file and variables are the

data that is to be written into the file. Key must not be null and must not already exist in the index.

The key is inserted in the index file and the record is written at the end of the data file. A new end-of-file mark is automatically placed after the added record. If your data file has more than one associated index file, use the INSERT instruction to add the key values to the other indexes. See the INSERT section for instructions on how to use this.

When inserting items whose keys fall randomly within the collating sequence, you can usually insert a number of records equal to one-tenth of the total number of records in the file before the insertions will take significantly longer. It generally is a good idea to REFORMAT with record compression and then rerun the INDEX utility as often as practical when many insertions and deletions are being performed. This will keep the speed of insertions and indexed accesses as high as possible.

The following examples show how indexed insertion WRITE statements work:

```
WRITE FILE,KEY;KEY,NAME,ADDRESS
```

This writes a new record containing the values for KEY, NAME, and ADDRESS at the end of the data file. An end-of-file mark is automatically written after ADDRESS. The index is updated.

```
WRITE FILE,KEY;KEY,NAME,ADDRESS;
```

This also writes a new record, but does not write an end-of-file mark at the end of the file. The index is updated. You could use this if you wanted to finish writing the record physically sequentially, and then write an EOF mark at the end of the file yourself. You must be careful, however, that no other DATASHARE user is going to do an insertion in that file before the EOF mark is written, or they will get a RANGE trap error.

If you are going to add a lot of data to your file, it often is a good idea to add it physically sequentially and then create a new index structure by re-running the INDEX utility.

UPDATE Modifies the Most Recent Record

The UPDATE instruction allows you to modify the last record that was accessed with a READ or READKS operation. You do not supply the key because it knows which key to use. UPDATE has the following format:

```
UPDATE file;variables
```

where file is the logical file name and variables are the data that is to be overwritten. For example:

```
UPDATE FILE;*20,ADDRESS
```

reads the last indexed accessed record in FILE and overstores the data in ADDRESS starting at the 20th character.

NOTE: Remember that tabbing instructions do not properly expand space and record compressed notation.

INSERT Updates Other Indexes

Often you will have more than one index for a data file. The indexed insertion WRITE statement only updates one index. To update any other indexes, use the INSERT instruction. When the INSERT operation is performed, the specified key is inserted into the specified index file. This must be performed after the associated indexed insertion WRITE and before another WRITE. The format of the INSERT instruction is:

```
INSERT file,key
```

where file is the name of the logical index file and key is the key that is to be inserted in the index.

DELETE Deletes a Record

DELETE allows a record to be physically deleted from a data file and for its key to be deleted from the specified index. The DELETE instruction is also used to delete keys from any extra index files for that data file. The DELETE instruction has this format:

```
DELETE file,key
```

where file is the name of the logical file and key is to be deleted from the index.

Assuming that the data file INVOICE has three associated logical index files, INVOICEA, INOIVCEB, and INVOICEC, the following instructions will delete the record pointed to by KEY and delete the index entries in the three index files:

```
DELETE INVOICEA,KEY
DELETE INVOICEB,KEY
DELETE INVOICEC,KEY
```

Because the DELETE operation actually just overstores the logical record with 032 delete characters, it does not release any space. Therefore, it is a good idea to REFORMAT and then re-index the file if you are doing a lot of deletions. REFORMAT will release this extra space. See the REFORMAT section of this chapter.

Writing the End-of-File Mark

When you create the data file, you use physically sequential or random access methods and use the WEOF command. When you add data using indexed insertion WRITE statements, the end-of-file mark is automatically written for you.

Indexed Program Example

The following example illustrates just one application suitable for indexed access. A company keeps information about each employee based on that employee's social security number.

A data file has been set up with the social security number in the first nine character positions. An index was created based on that field.

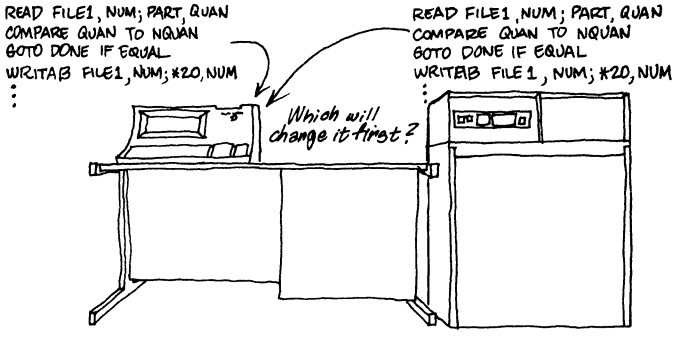
In this program, the user is asked to type in a social security number. If the social security number is found, all information about that person is displayed on the screen. If it isn't found, an error message is displayed. A social security number of nine zeroes ends the program.

```
DONE    INIT "000000000"
SS      DIM 9
NAME    DIM 20
ADR     DIM 20
CITY    DIM 15
STATE   DIM 2
ZIP     DIM 5
AFILE   IFILE
        OPEN AFILE,"CLASS1"
LOOP    KEYIN *ES,"SOCIAL SECURITY NUMBER:",SS
        READ AFILE,SS;SS,NAME,ADR,CITY,STATE,ZIP
        GOTO BAD IF OVER
        DISPLAY *P1:3,NAME,*P1:4,ADR,*P1:5,CITY:
           "  ",STATE,"  ",ZIP
        GOTO LOOP
BAD     MATCH SS TO DONE
        GOTO FIN IF EQUAL
        DISPLAY *P1:3,"**BAD NUMBER**"
        GOTO LOOP
FIN     DISPLAY *P1:3,"THANKS"
        STOP
```

Common File Access Considerations

Since DATASHARE is capable of executing several programs at once, more than one program can access a single file at any given time. There is no problem if these accesses are not modifying the contents of the file or if they are modifying different records in the same file.

However, if a certain record may be modified by more than one program at a time, a lockout mechanism is needed to allow one program to finish its modification before another program can start.



IF MORE THAN ONE USER TRIES TO MODIFY A RECORD, A PROBLEM CAN DEVELOP

The Prevent Interruptions (PI) instruction enables a programmer to prevent certain types of program activities from being interrupted by another program for up to 20 instructions. The number of instructions must always be a fixed decimal number, not a numeric variable. The instructions included cannot include KEYIN, DISPLAY, or CONSOLE. The following example shows an effective use of the PI instruction:

```
PI 4
READ FILE,KEY;NAME,QUAN,STOCK
SUB QTY FROM QUAN
GOTO NOTNUFF IF LESS
UPDATE FILE;NAME,QUAN,STOCK
```

Interruptions will be prevented from the PI instruction through the UPDATE instruction.

If, the user, rather than the program, needs to make a decision before a modification is made, the coding becomes more elaborate. The PI instruction cannot be used to prevent interrupts when a KEYIN, DISPLAY, or CONSOLE instruction must be executed. First the value should be read in and displayed for the user. Before a modification is made, however, the value should be rechecked. The following instructions illustrate this:

```
READ FILE,KEY;NAME,QUAN,STOCK
DISPLAY *ES,NAME,*N,QUAN,*N,STOCK
KEYIN *N,"CHANGE?",CHANGE
MATCH CHANGE TO "YES"
GOTO NO IF NOT EQUAL
KEYIN *N,"NEW QUANTITY:",NQUAN
PI 4
READ FILE,KEY;NAME,QUAN1,STOCK
MATCH QUAN1 TO QUAN
GOTO NE IF NOT EQUAL
UPDATE FILE,KEY;NAME,NQUAN,STOCK
```

If the user wants to change the value, he supplies the new value and the old value is checked to see if it has been changed. If it has been changed, control is switched to another part of the program, which tells the user that the value has been changed by someone else and asks the user if he still wants to change the value.

The REFORMAT Utility

The DOS REFORMAT utility is used to change the internal disk format of a data file. REFORMAT permits you to select essentially three different output file formats:

1. Blocked files that are not space compressed.
2. Record compressed files that are not space compressed.
3. Files that are both record compressed and space compressed.

Record compression and space compression are explained in the Physically Sequential Access section of this chapter.

Blocked files are used for physically random access, where each logical record must be associated with a distinct physical record. Often it is convenient to create a random file through the use of the Editor, which record and space compresses its output. REFORMAT can then reprocess the file into the correct format for random access.

Also, when a file is accessed with the indexed access method, any additions or deletions result in an increase in the physical size of the file. REFORMAT recovers vacated space and releases extra allocated space.

How to Use REFORMAT

REFORMAT must be run under DOS, and not under DATASHARE. ROLLOUT must be used to access DOS (see Chapter Six). The REFORMAT command has this format:

```
REFORMAT file1,file2;parameters
```

where file1 is the input file, file2 is optional and specifies the name output file which will contain a reformatted version of file1 (if this is omitted, file1 will be reformatted in place), and parameters include those listed below:

PARAMETER	DESCRIPTION
Bn	The output file will be blocked with n logical records per physical record and with no space or record compression. (You will usually want to use 1 as the value for n.)
C	The output file will be space and record compressed. The number of logical

records per physical record will be indeterminate.

R The output file will be record compressed. The number of logical records per physical record will be indeterminate.

Ln The length of each logical record will be adjusted to n characters. If the logical record is shorter than n characters, it will be padded with blanks to the proper length. If the logical record is longer than n characters, the action taken depends on the T or S parameter; which also must be specified:

T Truncate the logical record if necessary. All extra characters will be lost.

S Segment the logical record into as many logical records as necessary, with each containing n characters, padded if necessary.

D If reformatting is done in place and this parameter is specified, any disk space vacated by the reformatting process is returned to the operating system for reuse.

REFORMAT Messages

Because there are so many error messages that REFORMAT can display, and because those messages are explained in the DOS. User's Guide, refer to that book for an explanation of the REFORMAT messages.

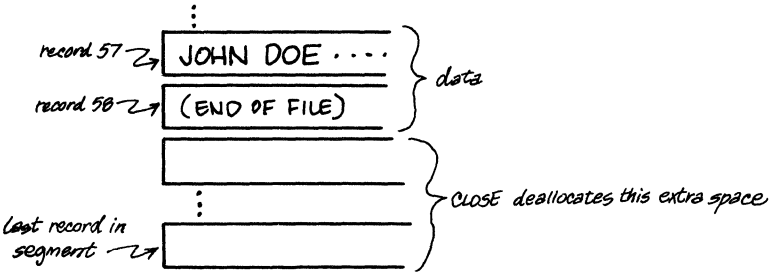
Helpful Hints

Once you understand how to use data files, these hints will help you use them even more effectively. These hints will help your DATASHARE program execute faster and more efficiently.

Close the File Properly

The CLOSE instruction can mean the difference between a lot of wasted space on a disk and the efficient use of the space on a disk. When a file is created, file space is allocated in segments. Under DOS.A each segment contains 192 physical records. Under DOS.B each segment contains 240 physical records. When a CLOSE instruction is executed, all extra physical records are given back to the operating system and can be used to form another segment. Without the CLOSE instruction,

any extra physical records will still be allocated to your data file.



USE CLOSE TO GET RID OF EXTRA SPACE

Group I/O into One Statement

If you group all of your reads or writes for one logical record into one statement, there will only be one physical disk head movement. But because there may be other DATASHARE users on the system and you may be using system routines, if you use several small read or write continuous statements (ending in a semicolon) the disk read/write head will have to reposition several times. Obviously this takes more time, so if you have the data area available to store one entire logical record, read or write it all at once.

The following example requires three disk head movements and wastes execution time:

```
READ AFILE,SEQ;NAME;
DISPLAY *ES,NAME
READ AFILE,SEQ;ADR,CITY,STATE,ZIP;
DISPLAY *N,ADR,*N,CITY,*N,STATE,* N,ZIP
READ AFILE,SEQ;NUM1,NUM2,NUM3
```

This requires only one disk head movement, efficiently uses execution time, and accomplishes the same result.

```
READ AFILE,SEQ;NAME,ADR,CITY,STATE:
   ZIP,NUM1,NUM2,NUM3
DISPLAY *ES,NAME,*N,ADR,*N,CITY,* N,STATE,*N,ZIP
```

Use REFORMAT to Reorganize Your Files

If you are using indexed access, and add or delete records, you should use the REFORMAT utility to release extra allocated space. See the REFORMAT section of this chapter. After REFORMAT, you must re-index your index file.

Write EOF at End-of-File

The WEOF instruction does not automatically space to the end of file to write an end-of-file mark. You must be positioned at the end of the data file, or the EOF mark will be written in the middle of your data area. If the EOF mark is written in the middle of your data area, you will overwrite some of your data. Also, if you are reading physically sequentially, you will not be able to read past the EOF mark.

The WEOF instruction should not be used with indexed insertion WRITES because the end-of-file is automatically written after each write for you.

CHAPTER THREE

PROVIDING SYSTEM SECURITY

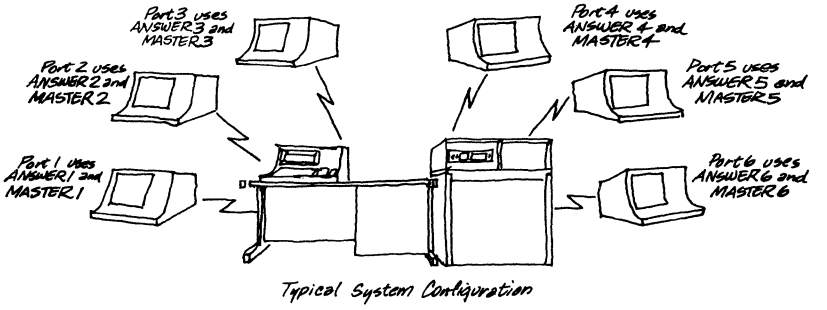
Introduction

Before a DATASHARE system can be used, there must be an ANSWER and a MASTER program for every port. The ANSWER program allows you to force the user to give some sort of identification before he is allowed to use the DATASHARE system. The ANSWER program chains to the MASTER program, when the STOP instruction is executed. The MASTER program usually requests the name of the program that the user wants to execute.

These programs, written in the DATABUS programming language, are created under the DOS Editor and are compiled just like any other DATASHARE program.

Separate Programs for Each Port

Because DATASHARE looks for an ANSWER and a MASTER program for each port, the ANSWER and MASTER programs are named ANSWER1 and MASTER1 for port 1, ANSWER2 and MASTER2 for port 2, etc.



Typical System Configuration

EACH PORT CAN USE ITS OWN ANSWER AND MASTER PROGRAMS

If DATASHARE cannot find the ANSWER or MASTER program for a port, it will look for programs named ANSWER or MASTER, and will use those programs. If those programs do not exist, that port will never become active even if it is configured into the system.

You may want to use separate ANSWER and MASTER programs to individualize the access procedure for each port. For example, you could use a different ANSWER and MASTER program sequence for the DATASHARE port you have in the accounting area than the programs for the port in the shipping area.

Certain programs, including DSREMOT and the DATASHARE Editor, need to know the port number associated with the port from which they are executing. To use these programs, separate MASTER programs must be used for each port and the port number must be the first data character of the MASTER program for that port.

Consider Your System

The ANSWER and MASTER programs must be tailored to fit your operating environment. Careful consideration will allow you to provide personal access restrictions and to enforce file access limitations.

The ANSWER Program

The ANSWER program can provide the opportunity to require that the user identifies himself properly before he uses the system. In an environment where security is important, this feature becomes an essential safeguard.

The User Must Satisfy the ANSWER Program

When a user first signs on to the system and is executing in the ANSWER program, he cannot escape identification requests by striking the INTERRUPT Key. The user must satisfy the requirements of the ANSWER program before he can be granted access to the DATASHARE system.

The ANSWER program can contain whatever identification sequence fits your operating environments. If you do not desire to use any identification whatsoever, your ANSWER program can be very short, possibly containing only an identifying message and a STOP statement. Two examples of ANSWER programs follow.

One essential security measure to take for all ANSWER programs is to make a backup copy of each ANSWER program on cassette and then destroy the source file for those programs. This insures that no one will use the Editor to list the program and find the security codes.

A Simple Answer Program

The following example shows a very simple answer program for port 3. First this program displays the port number on the user's display screen and displays its name on the console. Then the program asks the DATASHARE user to supply an identification ("ACCOUNTING"). If the identification matches, a STOP statement is executed, which causes a chain to the MASTER program. If the identification doesn't match, an error message is displayed and the user has to try again.

```
PORTN     FORM "3"
ID        DIM 10
IDCODE    INIT "ACCOUNTING"
          DISPLAY *ES,"DATASHARE PORT",PORTN
          CONSOLE "ANSWER",PORTN
LOOP      KEYIN *EOFF,"ID:",ID
          MATCH ID TO IDCODE
          GOTO GOOD IF EQUAL
          DISPLAY "***INVALID ID ***"
          GOTO LOOP
GOOD      STOP
          STOP
```

If you look closely at the KEYIN statement, you'll notice how a simple security measure was enforced. The *EOFF control inhibits the display of all characters that are typed in by the user. Because the echo is turned on after each KEYIN statement, the *EOFF control must be included at the start of each statement in which no echo is desired.

Because the input characters are not echoed back on the display screen, no one can see the identification code that the user is typing. Therefore, unauthorized people cannot see this identification and use the DATASHARE system.

A More Advanced Example

The simple example used only two simple security measures: an identification was requested and the echo was turned off while the identification was typed in. There are several other convenience and security measures that can be easily built into any ANSWER program.

One added security measure is to use the employee's social security number, or some other unique identifier, for identification instead of just one identifier for the entire group. This insures that only a select group of people can use the system.

Another added security measure is to limit the number of bad identifications allowed, so an unauthorized user can't keep entering identifications to try to break the system's security. After a certain number of tries, the ANSWER program can alert the operator that someone who apparently does not know the identification code is trying to access the system and prohibit the user from entering a valid identification.

The following program incorporates both of these added features. There are only five people that are allowed to use this DATASHARE port. If the user does not enter an acceptable number after three tries, the user is not allowed access to the system even if he does enter a valid identification.

Notice the use of the *W wait statements. Each *W causes a one-second wait. Use the *W rather than a counter or closed loop to cause a wait because the *W uses very little processor time. A counter or a closed loop, on the other hand, uses a lot of processor time and will slow down the entire DATASHARE system.

```
PORTN     FORM "3"
ID        DIM 9
JANE      INIT "342869387"
SALLY     INIT "289374621"
BOB       INIT "293872159"
JOHN      INIT "359123572"
GEORGE    INIT "381298724"
ZERO      FORM "0"
CTR       FORM "3"
ONE       FORM "1"
          DISPLAY *ES,"DATASHARE PORT",PORTN
          CONSOLE "ANSWER",PORTN
LOOP      KEYIN *EOFF,"ID:",ID
```

```
            COMPARE CTR TO ZERO
            GOTO WARN IF EQUAL
            MATCH ID TO JANE
            STOP IF EQUAL
            MATCH ID TO SALLY
            STOP IF EQUAL
            MATCH ID TO BOB
            STOP IF EQUAL
            MATCH ID TO JOHN
            STOP IF EQUAL
            MATCH ID TO GEORGE
            STOP IF EQUAL
            DISPLAY "**BAD IDENTIFICATION**"
            SUB ONE FROM CTR
            GOTO WARN IF EQUAL
            GOTO LOOP
WARN        CONSOLE "NO ID FOR PORT",PORTN
BAD         DISPLAY "**NO VALID ID ENTERED**",*N:
               "**PORT EXECUTION STOPPED**",*W,*W,*W,*W
            GOTO BAD
```

NOTE: A user could try to sign on to the system again after he has been stopped by the above program by turning his port off by the power switch and then turning it on again or redialing (if he is using a dial-up port). If you know that a port is used as a dial-up, you can revise the above program to disconnect the port. A CHAIN "ANSWER" statement will disconnect a dial-up terminal. Thus, the only change necessary to change this program to disconnect a dial-up connection is to change the last line of the program to read CHAIN "ANSWER".

When the terminal is reconnected to the DATASHARE system, the user must again try to satisfy the ANSWER program requests.

The MASTER Program

After the DATASHARE user has satisfied the requirements of the ANSWER program, the MASTER program is automatically executed. The MASTER program usually requests the name of the program that the user wishes to execute.

The DOS directory, which contains the list of programs available, cannot be directly accessed by the MASTER program, so either the user must know which program names he wants or the programmer must set up some way for the user to obtain the names of the programs. This can easily be done by setting up a file that contains the names of all of the programs. The MASTER program can then display the name of the available programs by reading this file.

Every STOP statement is actually considered to be a CHAIN MASTER statement by DATASHARE. Therefore, after each program is executed, control returns to the MASTER program for that port.

A Simple MASTER Program

This simple MASTER program merely requests the name of the program that the user wants to execute. A CHAIN is executed to that program, and if the program does not exist, an error message is displayed. The RELEASE statement is included in case the previously executing program for that port forgot to release the printer.

```
PORTN     FORM "3"
FILNAM    DIM 8
          RELEASE
          CONSOLE "MASTER",PORTN
LOOP      KEYIN *N,*EL,"PROGRAM NAME:",FILNAM
          TRAP NONAME IF CFAIL
          CHAIN FILNAM
NONAME    DISPLAY "**NO SUCH PROGRAM**"
          GOTO LOOP
```

A More Advanced Example

The simple example offers no help to the DATASHARE user who does not remember the name of the program he wants to execute. It also allows any user at any port to execute any DATASHARE program.

This more advanced example uses a file named HELP to contain the names of the files that the user can access. The user can request that these file names be listed for him, and the program checks that the user enters only one of the file names listed in the file.

```
PORTN     FORM "3"
HELP      INIT "HELP"
FILE      DIM 8
HELPF     FILE
NUM       FORM 2
NAME      DIM 8
ZERO      FORM "0"
ONE       FORM "1"
          RELEASE
          OPEN HELPF,"HELP"
          CONSOLE "MASTER",PORTN
          DISPLAY *ES,"MASTER PROGRAM FOR PORT",PORTN
LOOP      KEYIN "TYPE HELP FOR HELP",*N,"PROGRAM NAME:",FILE
          MATCH FILE TO HELP
          GOTO LIST IF EQUAL
          MOVE ZERO TO NUM
LOOP1     READ HELPF,NUM;NAME
          GOTO BAD IF OVER
          MATCH NAME TO FILE
          GOTO OK IF EQUAL
          ADD ONE TO NUM
          GOTO LOOP1
```

```
BAD       DISPLAY "**NO SUCH FILE**"
          GOTO LOOP
LIST      MOVE ZERO TO NUM
LISTLP    READ HELPF,NUM;NAME
          GOTO LOOP IF OVER
          DISPLAY NAME
          ADD ONE TO NUM
          GOTO LISTLP
OK        CHAIN FILE
```

CHAPTER FOUR

Virtual Memory Programming Considerations

Introduction

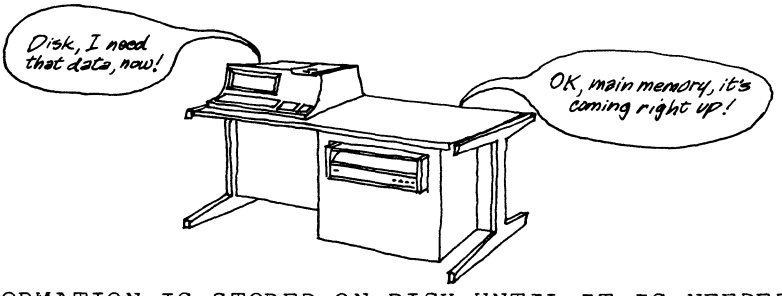
DATASHARE uses virtual memory techniques to give each user the illusion that there is more main memory available to him than actually exists. Virtual memory allows users at several ports to execute large programs that would otherwise overflow the existing memory space.

Once you understand how virtual memory works, you'll find that it may be easy to increase your program's execution speed by carefully structuring your programs. And since each program affects the speed of the entire DATASHARE system, carefully structured programs also help to increase the execution speed of all other executing programs.

What is Virtual Memory?

One of the most important aspects of any computer system is the main memory. Memory holds the information that tells the computer what it is to do at any given time. Memory also holds part of the data that the computer is going to work with.

Main memory space in any computer processor is limited. Disks, cassettes, and magnetic tapes are among the devices that are use to store the data, including information and instructions, when the processor doesn't immediately need it. When the processor needs that information or instruction, it must read it from the storage media into main memory, if it isn't there already.



INFORMATION IS STORED ON DISK UNTIL IT IS NEEDED

DATASHARE Manages the Memory Allocation

The DATASHARE program, with the help of the Disk Operating System (DOS), decides when new data must be read into main memory. To do this, it employs virtual memory techniques that allow DATASHARE to keep track of the memory or storage location of all of the data.

DATASHARE tells the Datapoint processor where the information is stored on the disk and where it should go in main memory. Because it takes time to read data from disk to main memory, DATASHARE tries to minimize the number of reads and writes that are necessary.

DATASHARE Code is Never Modified

Datapoint's DATASHARE system code is reentrant, which means that it is never modified and can be shared by more than one program at a time. Because it is never modified, the system DATASHARE code must only be read from disk, and never written back. This feature really saves the processor time because the processor never needs to write the code back to disk.

Because it is reentrant, more than one DATASHARE port can execute a section of DATASHARE system code at any one time. This permits all the ports to share the memory space allocated to the DATASHARE program itself.

DATASHARE system code is also very compact. A very few instructions are capable of invoking a large amount of processor activity. Therefore, only a small amount of new DATASHARE system code must be read from the disk to main memory.

Program Code is Accessed Often

DATABUS code for user programs is also reentrant, but the code for your DATABUS program is used very differently from the DATASHARE system code because it is accessed at a very high rate. Sections of program code are constantly read into main memory. Unless all the ports share one DATABUS program, several DATABUS programs must often appear to be resident in main memory for execution at various ports.

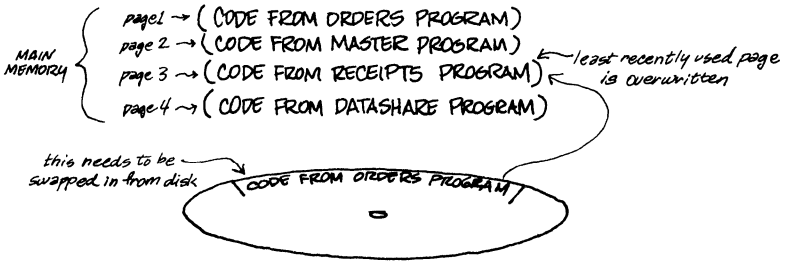
The next section discusses exactly how DATASHARE manages to swap system and program code. There are many effective ways you can structure your DATABUS programs to minimize the amount of swapping that is necessary.

Virtual Memory Implementation

Code is Divided Into Pages

DATABUS program code and DATASHARE system code is read into main memory in chunks that are called pages. Each page contains 250 bytes (a byte is an executable unit, often a character).

Each time a byte of program code needs to be fetched by the DATASHARE interpreter, a check is made to determine if that byte is immediately available in memory or if the page containing the byte must be read in from disk. DATASHARE's virtual memory techniques determine when a page must be swapped from disk to main memory. The page in main memory never needs to be written back to disk because the program code is never modified. The scheme used is based on pure demand, with the least recently used page being overwritten to make space for the new page.

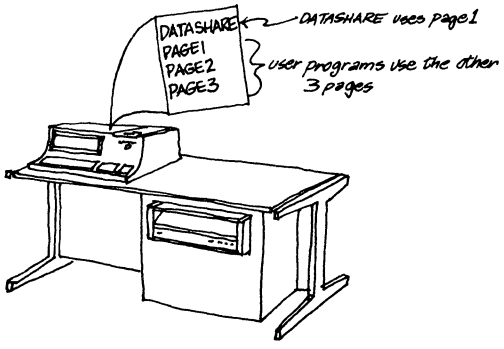


THE LEAST RECENTLY USED PAGE IS OVERWRITTEN WHEN A NEW PAGE NEEDS TO BE READ TO MAIN MEMORY

The Disk Limits the Number of Pages

The number of pages that can be resident in main memory at one time is limited by the disk controller that you are using with your DATASHARE system.

The 9350 series cartridge disk controller allows four pages to be resident in main memory at a time. The DATASHARE system code uses one page, leaving three pages for user programs.



YOU CAN HAVE FOUR PAGES IN MEMORY IF YOU'RE USING A CARTRIDGE DISK SYSTEM

The 9370 series mass storage disk controller allows 16 pages to be resident in main memory at a time. The DATASHARE system code uses four of these pages, leaving 12 pages for user programs.

YOU CAN HAVE 16 PAGES IN MEMORY IF YOU'RE USING A MASS STORAGE DISK SYSTEM

Attaching more disk drives to your DATASHARE system will not increase the number of pages allowed because one disk controller controls all of the disk drives. More disks, however, can increase the amount of supplementary storage available to your system. The number of pages that can be resident in main memory at any one time has serious implications on your DATASHARE system performance. The more pages that are in main memory, the fewer page swaps necessary.

Where are the Page Boundaries?

How can you tell if your code is crossing page boundaries? It's easy. Look carefully at the 5-digit octal number that is printed at the left side of your DATABUS program when it is compiled and listed on the printer. Whenever the middle digit changes to a four, or back again to a zero, a page boundary has been crossed.

In the following example, the page boundaries are marked:

```
01372     W4    SUB C1,X
01405           ADD C2,Y

01772           MOVE BASE,SIX
02005           RESET BASE TO 4

02372           SUB "1",MOVE
02413           GOTO CONT IF NOT ZERO
```

WHEN THE MIDDLE OCTAL DIGIT CHANGES TO A 4 OR A 0, A PAGE BOUNDARY HAS BEEN CROSSED

Programming Hints

Each time your DATABUS program code crosses a page boundary, a new page will have to be read into main memory. There are several effective ways you can minimize the amount of page swapping in your program.

Repeat Code Rather than Call Subroutines

One way to increase execution speed is to repeat small sections of code as much as possible rather than call a subroutine each time that code is needed. It is faster to repeat short sections of instructions rather than to call a subroutine each time those instructions are needed if the subroutine is located in a different page from the location where it is called.

2200 DATASHARE allows each program to use 16,000 bytes of effective memory space. 5500 DATASHARE allows each program to use 32,000 bytes of effective memory space. Because this limit is much larger than most DATASHARE programs ever need to use, repeating code usually will not enlarge a program beyond the limit.

Carefully Structure Loops

Loops are sections of code that are repeated. If a loop spans page boundaries, each page will have to be swapped back and forth between main memory and the disk. Therefore, it is best to keep loops small and compact.

Use the TABPAGE Instruction

The TABPAGE instruction forces sections of a program to start at a new page boundary. This instruction should only be added after the program is completely debugged, and should be used with caution.

You should never liberally scatter TABPAGE instructions throughout your program. This usually will result in an increase in the number of pages that must be read into main memory, which severely decreases execution speed.

A TABPAGE instruction can also cause more harm than help in another way. You may be able to increase the execution speed of one part of your program, but actually decrease it in another part of your program because the TABPAGE causes other sections of instructions to cross page boundaries at different places.

The best use for a TABPAGE instruction is to force often-repeated loops to reside entirely within one or two pages. Look carefully at your program once it works and decide if the TABPAGE instruction can really help you.

A Bad Example

The following section of instructions is a good illustration of how NOT to structure a program:

```
LOOP    KEYIN "NAME",NAME
        MATCH NAME TO "SALLY"
        GOTO L1 IF NOT EQUAL
        DISPLAY "SALLY'S IDENTIFICATION PROGRAM"
        CALL SALLY
        GOTO IDOK
L1      MATCH NAME TO "JANET"
        GOTO L2 IF NOT EQUAL
        DISPLAY "JANET'S IDENTIFICATION PROGRAM"
        CALL JANET
```

```
            GOTO IDOK
L2          MATCH NAME TO "BILL"
            GOT L3 IF NOT EQUAL
            DISPLAY "BILL'S IDENTIFICATION PROGRAM"
            CALL BILL
            GOTO IDOK
L3          DISPLAY "BAD IDENTIFICATION--TRY AGAIN"
            GOTO LOOP
IDOK        KEYIN "PROGRAM NUMBER:",NUM
```

There are two serious problems with this section of instructions. The biggest problem stems from the CALL statements. It's very doubtful if the subroutines are really necessary since control has to be transferred somewhere else anyway. It would be better to just transfer control to a section of code that includes the instructions that would be in the subroutines.

Subroutines should be universal in nature. Code should be written in-line unless it is called from various places in the program. Because it does not appear that these subroutines would be applicable to any other part of the program, it is best to avoid using the subroutines.

The second problem has to do with the size of the main loop. Loops should be kept as small as possible. The DISPLAY statements should be outside of the loop.

A Good Example

Notice how small the main loop is in the following example. No subroutines are called; instead, the instructions are included in the main program. This makes sense because control has to be transferred somewhere else anyway.

```
LOOP        KEYIN "NAME:",NAME
            MATCH NAME TO "SALLY"
            GOTO SALLY IF EQUAL
            MATCH NAME TO "JANET"
            GOTO JANET IF EQUAL
            MATCH NAME TO "BILL"
            GOTO BILL IF EQUAL
            DISPLAY "BAD IDENTIFICATION--TRY AGAIN"
            GOTO LOOP
SALLY       KEYIN "SALLY'S IDENTIFICATION PROGRAM",*N:
               "CODE WORD?",CODE
```

CHAPTER FIVE

Printing

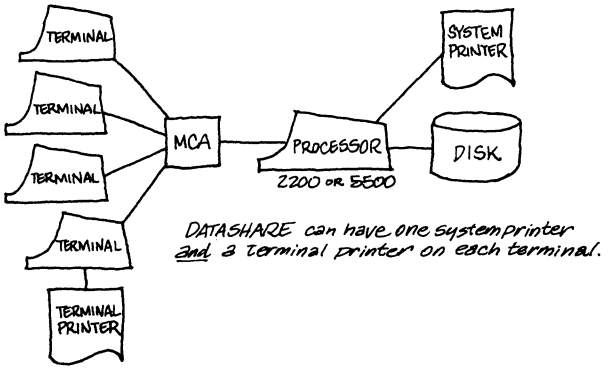
Introduction

Each DATASHARE system can have one system printer attached to it. This means that all ports have to share that one printer. This can create problems if too many users want to print at the same time. These problems can be circumvented, however, by careful program planning.

This chapter discusses the allocation of the system printer, which is directly attached to the central processor. Each port can have a terminal printer attached to it, however. The Datapoint 9292 Belt Printer can be used as a terminal printer.

To use the 9292 Serial Interface Belt Printer at a DATASHARE terminal, all you have to do is include the printer control characters in the DISPLAY statement. To turn on the terminal printer, use the 020 control characters (with a 3360 terminal) or the 032 control characters (with a 3600 terminal). To turn the terminal printer off, use the 024 control characters. All variables listed between the on and off characters will be printed. In the following example, NAME and ADDRESS are listed on the printer from a 3600 terminal:

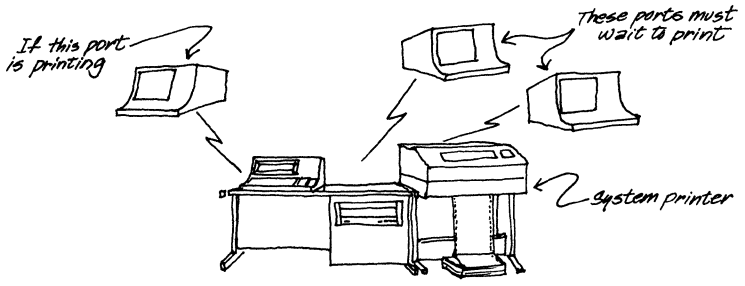
```
DISPLAY 032,NAME,ADDRESS,024
```



TERMINAL AND SYSTEM PRINTER LAYOUTS

How System Printing is Done

When a user executes the first PRINT instruction in a program, DATASHARE checks to see if the printer is available. If the printer is available, the program is given exclusive control over the printer. That program retains control over the printer until it executes a RELEASE instruction. If the printer is not available, execution is halted until it becomes available.



ONLY ONE PORT PRINTS AT A TIME

Remember to RELEASE the System Printer!

Once your program gets control of the printer, it has control until it executes a RELEASE. If you forget to include a RELEASE statement at the end of your printing operations in a program, the printer will hang, waiting for that instruction. Your other programs will not be able to use the printer, and neither will anyone else's programs.

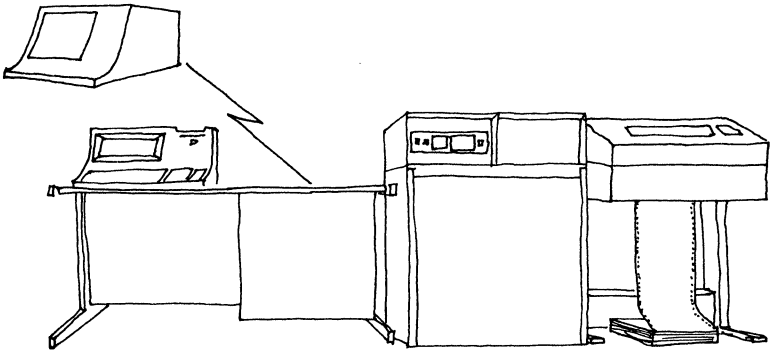
One safeguard against a user forgetting to RELEASE the printer is to include a RELEASE statement as the first instruction in the MASTER program. Because a STOP instruction in a DATASHARE system is actually a CHAIN MASTER statement, control is passed to the MASTER program immediately after a program is executed. Putting a RELEASE instruction at the beginning of any other program will not solve the problem, since DATASHARE ignores all releases except those belonging to the program that is printing.

Managing the Printer

There are a few effective controls that you can implement on your DATASHARE system to effectively utilize the one system printer. Some of these ways are very simple to implement; others are rather complex. Remember that any method should be tailored to your needs and requirements.

Write to a Data File, Then Print

Many users have found that it is more convenient to have their programs write to a data file before it is printed. A utility program can be written to read the data file and display it on the user's display screen. The user can check the file to be sure that the output is correct before it is printed.



Use One Port to Print All System Print Files

If you can afford to dedicate one DATASHARE port to printing, you can write a queuing program for all print files on the system printer that are written to disk. This one port could constantly execute this queuing program.

This is the way this queuing program could work. Anytime a user wants to enter the name of a data file into the queue that is to be printed, he could execute a program that asks for the name of the data file. The name of the data file is entered into the queue file. When the print program, operating on the dedicated port, is done printing one file, it reads the queue file for the next file name.

You can add some special features to each printout this way. You could print a beginning and ending burst page with each file. The burst page could contain the time, date, and name of the file for easy identification.

CHAPTER SIX

ROLLOUT and CHAIN

Introduction

ROLLOUT allows a DATASHARE user to temporarily stop DATASHARE execution at all of the ports and execute DOS commands. Once the DOS commands have been executed, DATASHARE can be restored to its previous status.

The DOS commands that are to be executed are contained in a CHAIN file. The CHAIN file specifies the sequence of DOS commands that are executed. The last instruction in the CHAIN file usually should be DSBACK, which restores the DATASHARE system to its previous status.

ROLLOUT is useful for compiling DATABUS programs, creating an index for a file, or sorting a file. These are operations that cannot be done under ordinary DATASHARE operation.

ROLLOUT Must be Configured

When you configure your DATASHARE system with the DSCON program, you are asked if ROLLOUT is to be configured. If you answer "YES", users will be able to execute ROLLOUT instructions from their DATABUS programs. If you answer "NO", users will not be able to use the ROLLOUT facilities.

If you are not sure if ROLLOUT has been configured for your system, you can run the DSCON program again.

How ROLLOUT Works

ROLLOUT is initiated by the ROLLOUT instruction in a DATABUS program. The ROLLOUT instruction has this format:

```
ROLLOUT string
```

where string specifies the function that is to be initially executed by DOS. Usually the string is a CHAIN command. The string can be initialized as a variable, as in this example:

```
ROLCMD    INIT "CHAIN DOSFILE"
          ROLLOUT ROLCMD
```

Or the string can be a quoted string in the command, as in this example:

```
ROLLOUT "CHAIN DOSFILE"
```

See the CHAIN section of this chapter to see how to set up the CHAIN file.

All Other Programs are Suspended

When a DATABUS program running from any port executes a ROLLOUT instruction, execution is temporarily suspended at all other ports until a DSBACK command is executed. The system status and memory is written out to a disk file named

ROLLFILE/SYS.

At the 2200 or 5500 console, a beep is sounded to alert the operator that a ROLLOUT is initiated. DATASHARE is suspended and DOS is initialized. The DATASHARE time clock is stopped during ROLLOUT unless the DSBACKTD command is executed. The ROLLOUT program supplies the commands for DOS. When the DOS functions are completed, the console is left at DOS level unless a DSBACK command is executed to restore the DATASHARE interpreter system to its previous status.

If the console is left at DOS level, DOS commands can be entered at the console. A DSBACK command will return the system to DATASHARE control.

When to Use ROLLOUT

The ROLLOUT feature is particularly useful when a file needs to be sorted with the DOS SORT command or if an index file needs to be made with the DOS INDEX command. Because DATABUS programs must be compiled under DOS, ROLLOUT can be used for DATABUS compilations. Examples of some of these functions are given in the CHAIN description of this chapter.

ROLLOUT Inconveniences Other Users

ROLLOUT inconveniences other DATASHARE users because it temporarily suspends the execution of their programs. They must wait (hopefully patiently) until the ROLLOUT is over. Therefore, you should use ROLLOUT with discretion and consideration of other users.

Also, unless the other users are informed in some way that a ROLLOUT is occurring, they will not know what is happening when a ROLLOUT is executed. Since their terminals appear inactive, they may think the system has gone down for some other reason.

ROLLOUT Precautions

There are a number of precautions which must be observed during the use of ROLLOUT. The functions performed under DOS must not affect any of the operations that were taking place under DATASHARE. For example, any of the ANSWER or MASTER programs must not be changed and files that are open and in use must not be modified or deleted.

When control returns to DATASHARE, certain items in memory reflecting the state of the DOS file structure are restored. If these items are no longer the same, terrible things can happen to the DATASHARE interpreter system. Operations to be watched in particular include the changing of the object code of any program that is running, the changing of any files that are open, and the re-arrangement of any disks with files in use within a multi-drive system.

Note that changing the DATASHARE configuration will not have effect until the next time the DATASHARE system is initialized. Reinitializing the system after ROLLOUT will not see the configuration change.

The CHAIN File

The CHAIN Command

The CHAIN command tells DOS to look at the disk file specified for the list of DOS commands that are to be executed. Basically, the CHAIN file takes the place of a user entering the commands at the system console under DOS.

To specify the name of the file that contains the DOS commands, specify that file name in the CHAIN command string, in the following manner:

```
CHAIN DOSFILE
```

The CHAIN File Contents

The easiest way to create a CHAIN file is to use the Editor. Each line in the CHAIN file contains an instruction for DOS to execute. What you put in your CHAIN file, of course, depends on your particular application. All CHAIN files should end with the DSBACK command.

The following CHAIN file simply asks DOS to sort a file:

```
SORT AFILE,BFILE
DSBACK
```

This CHAIN file contains INDEX commands for indexed files:

```
INDEX CLASS1,CLASS1A;1-9
INDEX CLASS1,CLASS1B;10-15
DSBACK
```

CHAIN files can contain any DOS commands. Remember not to include commands that change the state of the DOS file structure (see the ROLLOUT Precautions section for a complete list of precautions). See the DOS. User's Guide for an explanation of DOS commands.

APPENDIX A. INSTRUCTION SUMMARY

SYNTACTIC DEFINITIONS

condition	The result of any arithmetic or string operation: OVER, LESS, EQUAL, ZERO, or EOS (EQUAL and ZERO are two names for the same condition).
character string	Any string of printing ASCII characters.
event	The occurrence of a program trap: PARITY, RANGE, FORMAT, CFAIL, or IO.
list	A list of variables or controls appearing in an input/output instruction.
name	Any combination of letters (A-Z) and digits (0-9) starting with a letter (only the first eight characters are used).
label	A name assigned to a statement.
nvar	A name assigned to a statement defining a numeric string variable.
nval	A name assigned to an operand defining a numeric string variable or an immediate numeric value.
nlit	An immediate numeric value.
svar	A name assigned to a statement defining a character string variable.
sval	A name assigned to an operand defining a character string variable or a quoted alphanumeric character.
slit	An immediate character string, enclosed in double quotes (").
nlist	A series of contiguous numeric variables.

slist	A series of contiguous string variables.
rn	A positive record number (>= 0) used to randomly READ or WRITE on a file.
seq	A negative number (< 0) used to READ or WRITE on a file sequentially.
key	A non-null string used as a key to indexed accesses.
null	A null string used as a key to an indexed read.

FOR THE FOLLOWING SUMMARY:

Items enclosed in brackets [ ] are optional.

Items separated by the | symbol are mutually exclusive (one or the other but not both must be used).

```
COMPILER DIRECTIVES

  label     EQU         10
  label     EQUATE      100
            INC         filename[/ext]
            INCLUDE     filename[/ext]

FILE DECLARATIONS

  label     FILE
  label     IFILE

DATA DEFINITIONS

  label     FORM        n.m
  label     FORM        "456.23"
  label     DIM         n
  label     INIT        "character string"
  label     FORM        *n.m
  label     FORM        *"456.23"
  label     DIM         *n
  label     INIT        *"CHARACTER STRING"

CONTROL

            GOTO        (label)
            GOTO        (label) IF (condition)
            GOTO        (label) IF NOT (condition)
            BRANCH      (nvar) OF (label list)
            CALL        (label)
            CALL        (label) IF (condition)
            CALL        (label) IF NOT (condition)
            RETURN
            RETURN      IF (condition)
            RETURN      IF NOT (condition)
            STOP
            STOP        IF (condition)
            STOP        IF NOT (condition)
            CHAIN       (sval)
            CHAIN       (slit)
            TRAP        (label) IF (event)
            TRAPCLR     (event)
            ROLLOUT     (svar)
            ROLLOUT     (slit)
```

CHARACTER STRING HANDLING

```
MATCH     (svar) TO (svar)
MATCH     (slit) TO (svar)
MOVE      (svar) TO (svar)
MOVE      (slit) TO (svar)
MOVE      (svar) TO (nvar)
MOVE      (nlit) TO (nvar)
MOVE      (nvar) TO (svar)
APPEND    (svar) TO (svar)
APPEND    (slit) TO (svar)
APPEND    (nvar) TO (svar)
CMOVE     (sval) TO (svar)
CMATCH    (sval) TO (sval)
BUMP      (svar)
BUMP      (svar) BY (nlit)
RESET     (svar) TO (sval)
RESET     (svar) TO (nvar)
RESET     (svar)
ENDSET    (svar)
LENSET    (svar)
CLEAR     (svar)
EXTEND    (svar)
LOAD      (svar) FROM (nvar) OF (slist)
STORE     (svar) INTO (nvar) OF (slist)
STORE     (slit) INTO (nvar) OF (slist)
CLOCK     TIME TO (svar)
CLOCK     DAY TO (svar)
CLOCK     YEAR TO (svar)
TYPE      (svar)
SEARCH    (nvar) IN (nlist) TO (nvar) WITH (nvar)
SEARCH    (svar) IN (slist) TO (nvar) USING (nvar)
REPLACE   (svar) IN (svar)
REP       (slit) IN (svar)
```

ARITHMETIC

```
ADD         (nvar) TO (nvar)
ADD         (nlit) TO (nvar)
SUB         (nvar) FROM (nvar)
SUB         (nlit) FROM (nvar)
SUBTRACT    (nlit|nvar) FROM (nvar)
MULT        (nvar) BY (nvar)
MULT        (nlit) BY (nvar)
MULTIPLY    (nlit|nvar) BY (nvar)
DIV         (nvar) INTO (nvar)
DIV         (nlit) INTO (nvar)
DIVIDE      (nlit|nvar) INTO (nvar)
MOVE        (nvar) TO (nvar)
MOVE        (nlit) TO (nvar)
COMPARE     (nvar) TO (nvar)
COMPARE     (nlit) TO (nvar)
LOAD        (nvar) FROM (nvar) OF (nlist)
STORE       (nvar) INTO (nvar) OF (nlist)
STORE       (nlit) INTO (nvar) OR (nlist)
CHECK11     (svar) BY (svar)
CK11        (svar) BY (slit)
CHECK10     (svar) BY (svar)
CK10        (svar) BY (slit)
```

INPUT/OUTPUT

```
KEYIN       (list)
DISPLAY     (list)
BEEP
PRINT       (list)
PREPARE     (file),(svar|slit)
PREP        (file),(svar|slit)
OPEN        (file|ifile),(svar|slit)
CLOSE       (file|ifile)
WRITE       (file|ifile),rn|seq|key[;[(list)][;]]
WRITAB      (file),rn|seq;(list)[;]
WEOF        (file|ifile),rn|seq
UPDATE      (ifile)[;[(list)][;]]
READ        (file|ifile),rn|seq|key|nul;(;|(list[;]))
READKS      (ifile);(;|(list[;]))
DELETE      (ifile),(svar)
INSERT      (ifile),(svar)
```

APPENDIX B.

KEYBOARD CODING (ASCII)

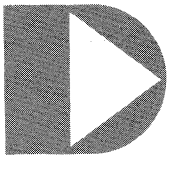
A-101	a -141	0-060	: -072
B-102	b -142	1-061	; -073
C-103	c -143	2-062	< -074
D-104	d -144	3-063	= -075
E-105	e -145	4-064	> -076
F-106	f -146	5-065	? -077
G-107	g -147	6-066	[-133
H-110	h -150	7-067	~ -176
I -111	i -151	8-070	] -135
J -112	j -152	9-071	^ -136
		Space-040	_ -137
K-113	k -153		
L-114	l -154	! -041	@ -100
M-115	m-155	" -042	{ -173
N-116	n -156	# -043	\ -134
O-117	o -157	$-044	' -140
P-120	p -160	% -045	| -174
Q-121	q -161	& -046	} -175
R-122	r -162	' -047	Enter-015
S-123	s -163	(-050	Cancel-030
T-124	t -164	) -051	Backspace-010
U-125	u -165	* -052	Del-177
V-126	v -166	+ -053	
W-127	w-167	, -054	
X-130	x -170	- -055	
Y-131	y -171	. -056	
Z-132	z -172	/ -057	

HOME OFFICE:
9725 Datapoint Drive
San Antonio, Texas 78284

SALES OFFICES:
Atlanta/(404) 458-6423
Boston/(617) 890-0440
Chicago/(312) 298-1240
Cincinnati/(513) 481-2600
Cleveland/(216) 831-0550
Dallas/(214) 661-5536
Denver/(303) 770-3921
Des Moines/(515) 225-9070
Detroit/(313) 478-6070
Greensboro/(919) 299-8401
Hartford/(203) 677-4551
Houston/(713) 688-5791
Kansas City/(913) 321-5802
Los Angeles/(213) 645-5400
Milwaukee/(414) 453-1425
Minneapolis/(612) 854-4054
New Orleans/(504) 522-5457
New York/(212) 736-3710
Orlando/(305) 896-1940
Philadelphia/(215) 667-9477
Phoenix/(602) 265-3909
Pittsburgh/(412) 931-3663
Portland/(503) 223-2411
San Diego/(714) 460-2020
San Francisco/(415) 968-7020
Seattle/(206) 455-2044
St. Louis/(314) 878-6595
Stamford/(203) 359-4175
Tulsa/(918) 664-2295
Union/(201) 964-8761
Washington, D.C./(703) 790-0555

INTERNATIONAL:
Australia/Sydney/(2) 922-3100
Austria/Vienna/0222/36 21 41
Belgium/Brussels/3762030
Brazil/Rio de Janeiro/222-4611
Canada/Toronto/(416) 438-9800
Denmark/Copenhagen/(02)96 53 66
Ecuador/Guayaquil/394 844
England/London/(1) 903-6261
Finland/Helsinki/(90) 661 991
France/Paris/(1) 657-13-31
Germany/Hannover/(0511) 634-011
Holland/Rotterdam/(10) 216244
Hong Kong/(5) 243-121
Iran/Tehran/8538857
Israel/Tel-Aviv/(03) 410565
Italy/Milan/316 333
Japan/Tokyo/(264) 6131
Lebanon/Beirut/(348) 340/1/2
Norway/Oslo/153490
The Philippines/Makati Rizal/877 294
Singapore/Singapore/911788
South Africa/Johannesburg/724-9301
Spain/Las Arenas/63 64 00
Sweden/Stockholm/(8) 188295
Switzerland/Lyss Berne/(32) 844240
Taiwan/Taipei/768-1114
USA/Los Angeles, Calif./(213) 475-6777

DATAPOINT CORPORATION



The Leader in Dispersed Data Processing

The text in this catalog was entered, edited and typeset using
a Datapoint 2200 with the SCRIBE text processing program
and a phototypesetter.